

# THE FOUNDATIONS FOR A SCALEABLE METHODOLOGY FOR SYSTEMS DESIGN

by

Toby Myers

*B.Eng(hons), B.InfTech*

SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY  
SCIENCE, ENVIRONMENT, ENGINEERING AND TECHNOLOGY  
GRIFFITH UNIVERSITY

Submitted in fulfillment of the requirements of the degree of

DOCTOR OF PHILOSOPHY

MARCH 2010



# Abstract

Marketplace demand is driving the need to develop software systems of ever increasing scale. Managing the complexity created by this increasing scale is crucial. Failure to adequately address the complexity that emerges with increasing scale can play havoc with even the most simple of tasks. Mainstream software & systems engineering approaches are struggling to manage the complexity of building large-scale software-intensive systems which is resulting in the widespread failure of projects. These failures are the result of two deficiencies in mainstream approaches. Firstly, these approaches utilise abstraction to manage complexity. Abstraction is a temporary solution which just delays the re-emergence of complexity until the approach is applied to larger systems. Secondly, these approaches do not provide a clear path from the requirements of a system to a final work product. It is common instead for a miraculous leap of intuition to occur from the initial requirements to a specification, a design or a deployed system. To ensure requirements are met, the resulting work product then must be iteratively re-evaluated against the requirements and corrected until it achieves acceptable quality. This construct-by-correction approach results in unnecessary rework, and can be overwhelmed by the complexity of large-scale systems.

The objective of this dissertation is to address the issue of scalability in software & systems engineering by providing the foundations for a scalable, widely applicable, end-to-end methodology. To achieve this we have extended Behavior Engineering (BE), which is an integrated approach to systems development that supports the engineering of large-scale dependable software intensive systems at both the systems and software engineering level. BE uses a bottom-up process that enables each requirement to be modeled independently

and integrated one at a time to form a complete view of the system specification that is built out of the requirements. Current research involving BE focuses primarily on using BE models as a formal specification, which can then be further analysed using techniques such as model-checking. This dissertation extends BE by providing a new design stage developed within a model driven engineering framework.

The resulting end-to-end methodology is demonstrated using three case studies intended to show a wide cross-cutting of applications. In the first case study, the extended BE approach is demonstrated by deploying a BE design on an embedded controller.

In the second case study, the BE approach is combined with Modelica, a mathematical modeling language. Together, BE and Modelica are used to develop a new approach called Co-Modeling, which involves the development of systems composed of integrated software and hardware components. Co-modeling is demonstrated with a case study involving the development of an automated train protection system which monitors a train driver and takes control of the train if a dangerous situation is not responded to. The developed co-model is used to investigate co-modeling scenarios and to determine the effect that various combinations of sensors, actuators and hardware platforms have on the behavior of the integrated system.

The third case study introduces a migration approach to dealing with legacy systems which uses a BE model as an intermediary. The approach is demonstrated using a real-world case study from industry. The outdated circuitry is captured in a BE model using a domain-specific extension and used to generate VHDL, a hardware description language. To demonstrate the benefit of using BE as an intermediary, the BE model is also used to perform failure mode and effects analysis, a procedure more commonly applied to BE models developed from software requirements.

Together these case studies demonstrate the potential of using a scaleable methodology to manage the complexity of designing the software-intensive systems of software & systems engineering.



# Declaration

This work has not previously been submitted for a degree or diploma in any university. To the best of my knowledge and belief, the thesis contains no material previously published or written by another person except where due reference is made in the thesis itself.

---

Toby Myers



# Acknowledgements

I would like to gratefully acknowledge the support I have received from many people during the preparation and writing of this dissertation. First and foremost, I would like to convey my sincere gratitude to my supervisor, Professor Geoff Dromey, for his unwavering support and encouragement throughout this project. Your recent passing away has saddened many people and I feel honoured for the time you gave to teach me, even in your failing health. This work could not have come to fruition without the opportunities you provided. I would also like to thank my associate supervisor, Professor Vladimir Estiville-Castro, for taking over supervision duties towards the end of this dissertation.

I extend my thanks to my colleagues working on the Building Dependability into Complex Computer-Based Systems project including Nisansala Yatapanage, Kirsten Winter, Robert Colvin, Lars Grunske, Ian J. Hayes, Peter Lindsay, Lian Wen, Diana Kirk, John Seagrott and Saad Zafar. The contributions each you have made created a strong foundation on which to base this work. I also thank you all for the robust discussions at our weekly meetings, and for your advice which helped shape this dissertation. In addition I would like to thank Joern Guy Suss for introducing me to the Eclipse Integrated Development Environment.

I would like to thank my family for their patience and understanding during the substantial time it took to complete this project. I particularly want to thank my wife, Toni and my son, David for providing me with such a loving and supportive environment whilst I undertook this work.

Finally, I would like to acknowledge the financial assistance provided by Griffith University and Raytheon Australia during the course of this project.



# List of Publications

- Myers, T., Fritzson, P., Dromey, R.G. *Co-Modeling: From Requirements to an Integrated Software/Hardware Model*. (submitted to IEEE Computer Magazine)
- Myers, T., Dromey, R. G. (2009), *From Requirements to Embedded Software - Formalising the Key Steps*, in Proceedings of the 2009 Australian Software Engineering Conference (ASWEC'09), Gold Coast, Australia, 14-17 April, pp. 23-33.
- Myers, T., Fritzson, P., Dromey, R. G. (2008), *Seamlessly Integrating Software & Hardware Modelling for Large-Scale Systems*, in Proceedings of the 2nd International Workshop on Equation-Based Object-Oriented Languages and Tools, Paphos, Cyprus, July 8, pp. 5-15.
- <sup>1</sup> Myers, T. J., Noel, T., Parent, M., Vlacic, L. (2005), *Autonomous Motion of a Driverless Vehicle operating among Dynamic Obstacles*, in Proceedings of the 44th IEEE Conference on Decision and Control, and the European Control Conference 2005 (CDC-ECC'05), 12-15 December, Seville, Spain, pp. 5071- 5076.
- <sup>1</sup> Myers, T. J., Vlacic, L., Noel, T., Parent, M. (2005), *Autonomous driving in a time-varying environment*, in the 2005 IEEE Workshop on Advanced Robotics and its Social Impacts, Nagoya, Japan, 12-15 June, pp. 53-58.

---

<sup>1</sup>These publications are unrelated to the topic of this dissertation but were completed during candidature.



## Dedication

*I dedicate this work to my father, may you rest forever peaceful,  
frolicking through the elysian fields.*





# Contents

<b>Abstract</b>	<b>iii</b>
<b>Declaration</b>	<b>v</b>
<b>Acknowledgements</b>	<b>vii</b>
<b>List of Publications</b>	<b>ix</b>
<b>Dedication</b>	<b>xi</b>
<b>List of Figures</b>	<b>xix</b>
<b>List of Tables</b>	<b>xxv</b>
<b>Supplementary Material</b>	<b>xxvii</b>
<b>I Introduction &amp; Background</b>	<b>1</b>
<b>1 Introduction</b>	<b>3</b>
1.1 Objective and Approach . . . . .	7
1.2 Thesis Hypotheses . . . . .	11
1.3 Contribution . . . . .	13
1.4 Thesis Structure . . . . .	14

<b>2</b>	<b>An Overview of Model Driven Engineering</b>	<b>17</b>
2.1	Origins and Goals of MDE . . . . .	18
2.2	Model Driven Engineering Approaches . . . . .	21
2.2.1	Meta-Modeling and the Meta-Object Facility . . . . .	23
2.2.2	The UML Platform-Independent Model . . . . .	26
2.2.3	Executable UML . . . . .	30
2.3	A New MDE GPL for System Behavior . . . . .	32
2.3.1	Representing System Behavior . . . . .	33
2.3.2	Using a Scaleable Methodology . . . . .	46
2.3.3	Extension Mechanism . . . . .	51
2.4	Conclusion . . . . .	54
<b>3</b>	<b>Introduction to Behavior Engineering</b>	<b>57</b>
3.1	Overview of the Behavior Modeling Language . . . . .	59
3.1.1	Behavior Trees . . . . .	59
3.1.2	Composition Trees . . . . .	66
3.2	Overview of the Behavior Modeling Process . . . . .	67
3.2.1	Formalisation . . . . .	68
3.2.2	Testing Fitness for Purpose . . . . .	71
3.2.3	Specification . . . . .	73
3.3	Modeling the Simple Microwave Oven . . . . .	75
3.3.1	Formalisation . . . . .	76
3.3.2	Testing Fitness for Purpose . . . . .	81
3.3.3	Specification . . . . .	84
3.4	Related Research . . . . .	90
3.5	Discussion . . . . .	91
3.6	Conclusion . . . . .	91
<b>4</b>	<b>Design with Behavior Engineering</b>	<b>93</b>
4.1	Why Design With Behavior Engineering . . . . .	94
4.2	Deploying a BE Model . . . . .	97

4.2.1	Replicating Hardware Success with Software Components . . . . .	97
4.2.2	The BE Component Model . . . . .	100
4.3	An Extension Mechanism for BE . . . . .	103
4.4	The Design Stage of the BMP . . . . .	105
4.4.1	Resolving the System/Environment Boundary . . . . .	105
4.4.2	Resolving the System/Component Boundary . . . . .	107
4.4.3	The Deployment Composition Tree . . . . .	108
4.5	Designing the Simple Microwave Oven . . . . .	108
4.6	A Previous Approach to Design with BE . . . . .	112
4.7	Discussion . . . . .	115
4.8	Conclusion . . . . .	116
<b>II</b>	<b>Forward Modeling</b>	<b>119</b>
<b>5</b>	<b>Behavior Engineering of Embedded Systems</b>	<b>121</b>
5.1	The BE EMF Editor . . . . .	125
5.2	Model-to-Model Transformations . . . . .	128
5.2.1	The BT M2M Transformation Language . . . . .	130
5.2.2	Implementing M2M Transformations in ATL . . . . .	134
5.3	Model-to-Text Transformations . . . . .	137
5.3.1	M2T transformation of the Behavior Tree . . . . .	137
5.3.2	M2T transformation of the Composition Tree . . . . .	138
5.4	The Process Control Model . . . . .	140
5.5	Deployment of the One-Minute Microwaver . . . . .	144
5.6	Discussion . . . . .	147
5.7	Conclusion . . . . .	147
<b>III</b>	<b>Integrated Modeling</b>	<b>151</b>
<b>6</b>	<b>Co-modeling</b>	<b>153</b>

6.1	From Co-design to Co-modeling . . . . .	155
6.2	Constructing a Co-Model . . . . .	159
6.2.1	Modeling the Requirements . . . . .	159
6.2.2	Building a Virtual Environment . . . . .	175
6.2.3	Implementing Software/Hardware Interactions . . . . .	179
6.3	Exploring the Possibilities . . . . .	181
6.4	Towards Co-modeling of Cyber-Physical Systems . . . . .	185
6.5	Discussion . . . . .	186
6.6	Conclusion . . . . .	186

## **IV Reverse Modeling 189**

### **7 Behavior Engineering of Digital Circuits 191**

7.1	Capturing Digital Circuits with BTs . . . . .	194
7.2	Replicating Legacy Hardware with BE . . . . .	200
7.2.1	Translating a Hardware Requirement . . . . .	202
7.2.2	Testing Fitness for Purpose . . . . .	207
7.2.3	Specification . . . . .	210
7.2.4	Design . . . . .	212
7.3	Performing Failure Modes and Effects Analysis . . . . .	215
7.3.1	Modifying the MBT for Performing Model-Checking . . . . .	215
7.3.2	Results of Performing Failure Modes and Effects Analysis . . . . .	218
7.4	Simulation and Synthesis . . . . .	219
7.4.1	Generating VHDL Code . . . . .	220
7.4.2	Simulation . . . . .	224
7.4.3	Synthesis . . . . .	225
7.5	Discussion . . . . .	227
7.6	Conclusion . . . . .	228

<b>V</b>	<b>Discussion</b>	<b>231</b>
<b>8</b>	<b>Discussion</b>	<b>233</b>
8.1	A Scaleable Methodology for System Design . . . . .	234
8.2	Comparison with Related Work . . . . .	235
8.3	Contribution . . . . .	237
8.4	Future Work . . . . .	239
<b>VI</b>	<b>Appendices</b>	<b>243</b>
<b>A</b>	<b>Contrasting BTs with other visual languages</b>	<b>245</b>
A.1	Case Study I: State Machines . . . . .	246
A.1.1	Transitions . . . . .	248
A.1.2	Segmentation . . . . .	249
A.1.3	Methodology . . . . .	249
A.2	Case Study II: StateCharts . . . . .	250
A.2.1	Transitions . . . . .	252
A.2.2	Segmentation . . . . .	253
A.2.3	Methodology . . . . .	253
A.3	Case Study III: Exogenous Connectors . . . . .	260
A.3.1	Transitions . . . . .	261
A.3.2	Segmentation . . . . .	263
A.3.3	Methodology . . . . .	263
<b>B</b>	<b>Behavior Tree Language</b>	<b>267</b>
B.1	Naming Conventions . . . . .	268
B.1.1	Variable Naming Conventions . . . . .	268
B.1.2	Node Naming Conventions . . . . .	269
B.1.3	Relation Naming Conventions . . . . .	270
B.1.4	Tree Naming Conventions . . . . .	271
B.1.5	Tree Branch Naming Convention . . . . .	272

B.2 Behavior Tree Notation & Syntax . . . . .	273
B.2.1 Node Tags . . . . .	273
B.2.2 Basic Nodes . . . . .	274
B.2.3 Behavior Tree Composition . . . . .	275
B.2.4 Node Operators . . . . .	276
<b>C Investigating the Hardware Component Model</b>	<b>277</b>
<b>D Behavior Tree Model-to-Model Transformations</b>	<b>285</b>
D.1 Conjunction Transformation Rule . . . . .	286
D.2 Disjunction Transformation Rule . . . . .	287
D.3 Exclusive OR Transformation Rule . . . . .	288
D.4 Reversion Transformation Rule . . . . .	289
D.5 Reference Transformation Rule . . . . .	289
D.6 Branch-Kill Transformation Rule . . . . .	290
D.7 Synchronisation Transformation Rule . . . . .	290
<b>References</b>	<b>291</b>
<b>Glossary</b>	<b>309</b>
<b>Acronyms</b>	<b>315</b>
<b>Index</b>	<b>320</b>

# List of Figures

2.1	An Overview of the elements of MDE . . . . .	20
2.2	MDA Approaches . . . . .	23
2.3	Three visual languages for capturing system behavior [Sow00] . . . . .	34
2.4	Network Structure versus Tree-Like Structure . . . . .	36
2.5	Segmentation and Transitions in a Petri Net . . . . .	38
3.1	An Outline of Behavior Engineering . . . . .	59
3.2	Summary of the Core Elements of the Behavior Tree Notation . . . . .	61
3.3	Behavior Tree Node Naming Conventions . . . . .	62
3.4	Summary of the Core Elements of the Composition Tree Notation . . . . .	66
3.5	Example Composition Tree . . . . .	68
3.6	Initial Translation of Requirement #1 of the Microwave Oven . . . . .	77
3.7	The Requirement Behavior Trees of the Microwave Oven . . . . .	80
3.8	Integrating RBT2 and RBT5 and integrating the result with RBT7 . . . . .	81
3.9	The Integrated Behavior Tree of the Microwave Oven . . . . .	82
3.10	The Integrated Composition Tree of the One-minute Microwaver . . . . .	83
3.11	Partial Model Behavior Tree #1 . . . . .	85
3.12	Partial Model Behavior Tree #2 . . . . .	87
3.13	Model Behavior Tree of the One-minute Microwaver . . . . .	89
4.1	Exponential Growth of Transistors per Integrated Circuit . . . . .	98

4.2	The structure of a deployed BE system . . . . .	100
4.3	Extending BT Expression Syntax to include a Factorial Operation . . . . .	104
4.4	Partial Design Behavior Tree #1 . . . . .	110
4.5	Design Behavior Tree (with deleted nodes hidden) . . . . .	111
4.6	An Example Component Interaction Network (CIN) . . . . .	112
4.7	The Component Behavior Tree (CBT) of the Light Component . . . . .	113
4.8	The Component Interface Diagram (CID) of the Light Component . . . . .	114
5.1	The workflow of the embedded Behavior Runtime Environment . . . . .	125
5.2	The Behavior Engineering Metamodel . . . . .	127
5.3	The BT and CT of the Microwave Oven in the BE EMF Editor . . . . .	129
5.4	Textual Terms for Matching BT Nodes . . . . .	131
5.5	Graphical Notation for Matching BT Nodes . . . . .	132
5.6	Matching Groups of Nodes . . . . .	133
5.7	Synchronisation Transformation Rule . . . . .	135
5.8	Implementing the Synchronisation Rule . . . . .	136
5.9	Generating Code with Java Emitter Templates (JET) . . . . .	138
5.10	Code Representation of a Behavior Tree . . . . .	139
5.11	Code Representation of a Composition Tree . . . . .	139
5.12	The Five-State Process Control Model . . . . .	140
5.13	The two possible paths for a BT Node through the process control model . .	141
5.14	The BE One-Minute Microwaver . . . . .	144
5.15	BE Boundaries of the One Minute Microwaver . . . . .	146
6.1	From Co-design to Co-modeling . . . . .	158
6.2	The Requirement Behavior Trees of the ATP System (R1,R2,R4,R6,R7,R8) .	161
6.3	The Requirement Behavior Trees of the ATP System (R3,R5,R9) . . . . .	162
6.4	The Integrated Behavior Tree of the ATP System . . . . .	164
6.5	The Integrated Composition Tree of the ATP System . . . . .	165
6.6	Partial Model Behavior Tree #1 of the ATP System . . . . .	167
6.7	Partial Model Behavior Tree #2 of the ATP System . . . . .	169



6.8	Model Behavior Tree of the ATP System . . . . .	171
6.9	Partial Design Behavior Tree of the ATP System . . . . .	173
6.10	Design Behavior Tree of the ATP System (with deleted nodes hidden) . . . . .	176
6.11	Component Diagram of the Modelica model of the ATP System . . . . .	178
6.12	Interactions between Modelica and BE Models . . . . .	179
6.13	Simulation of several co-modeling scenarios using the co-model of the ATP System . . . . .	184
7.1	Mapping for Generating a Rising and Falling Signal State . . . . .	197
7.2	Mapping for Receiving a Signal from Multiple Sources . . . . .	198
7.3	Two Mappings for Inverting a Signal . . . . .	199
7.4	The Sub-System of the A2 Controller Card . . . . .	201
7.5	Translating a Hardware Requirement . . . . .	202
7.6	Comparison of INTCLK and 32MHz signals . . . . .	205
7.7	The A2 sub-system partitioned into the 17 hardware requirements . . . . .	206
7.8	The Requirement Behavior Trees of the Sub-System (R1-R9) . . . . .	207
7.9	The Requirement Behavior Trees of the Sub-System (R10-R17) . . . . .	208
7.10	Integrating RBT1 with RBT2 followed by integration with RBT14 . . . . .	208
7.11	The Integrated Behavior Tree of the Sub-System . . . . .	209
7.12	The Model Behavior Tree of the Sub-System . . . . .	213
7.13	The Design Behavior Tree of the Sub-System (with deleted nodes hidden) . . . . .	214
7.14	The Modified MBT used for Model-Checking . . . . .	216
7.15	VHDL Code generated from the DBT . . . . .	221
7.16	Component Behavior Tree of Divide By 2 Component . . . . .	223
7.17	Simulation of the Clock Select System . . . . .	224
7.18	VHDL of TTL Version of the Clock Select System . . . . .	226
7.19	Synthesised Gate-Level design of Clock Select System . . . . .	227
8.1	Summary of Contribution . . . . .	238
A.1	State Transition Diagram . . . . .	247

A.2	StateChart of an Aircraft . . . . .	251
A.3	Aircraft StateChart modified to show interdependencies . . . . .	252
A.4	Partial Class Collaboration Diagram . . . . .	255
A.5	Partial Class Diagram . . . . .	256
A.6	Partial Building StateChart . . . . .	257
A.7	Partial Bank Operations in Action Language . . . . .	257
A.8	Partial Shaft StateChart . . . . .	258
A.9	Partial Cab StateChart . . . . .	258
A.10	Partial Door StateChart . . . . .	259
A.11	Exogenous Connectors Composite Component Alarm_Brakes_Control . . . . .	261
A.12	Interface Definition of ALRM_BRKS_Control Composite Component . . . . .	262
A.13	Exogenous Connectors Composite Component Reset_Alarm_Brakes_Control . . . . .	265
A.14	Exogenous Connectors model of the Automated Train Protection System . . . . .	265
B.1	Behavior Tree Node Naming Conventions . . . . .	269
B.2	Behavior Tree Relation Naming Conventions . . . . .	270
B.3	Behavior Tree Tree Naming Conventions . . . . .	271
B.4	Tree Branch Naming Convention . . . . .	272
C.1	Overview of a simple Digital Clock . . . . .	278
C.2	Modifying the Encapsulated Functionality of a Hardware Component . . . . .	279
C.3	Internal and External Views of the 7490 IC . . . . .	280
C.4	Units of the Seconds Section . . . . .	280
C.5	Reconfiguring a component using integration . . . . .	282
C.6	The Hours Module of the Simple Digital Clock . . . . .	283
C.7	Reuse in Digital Clock Case Study . . . . .	284
D.1	Conjunction Transformation Rule . . . . .	286
D.2	Disjunction Transformation Rule (Alternate Branching) . . . . .	287
D.3	Disjunction Transformation Rule (Parallel Branching) . . . . .	287
D.4	Exclusive OR Transformation Rule (Alternate Branching) . . . . .	288

---

D.5 Exclusive OR Transformation Rule (Parallel Branching) . . . . .	288
D.6 Reversion Transformation Rule . . . . .	289
D.7 Reference Transformation Rule . . . . .	289
D.8 Branch-Kill Transformation Rule . . . . .	290
D.9 Synchronisation Transformation Rule . . . . .	290



# List of Tables

2.1	Summary of Behavior-Based Approaches . . . . .	41
3.1	Summary of the first three stages of the Behavior Modeling Process . . . . .	69
3.2	Requirements of the Microwave Oven . . . . .	76
3.3	Issues found during translation of requirements of the One-minute Microwaver	79
4.1	Comparison of Potential Applications of the BE specification . . . . .	95
4.2	Hardware Components versus BE Components . . . . .	99
5.1	2003 and 2008 Embedded Systems Development Survey Results . . . . .	122
5.2	The Behavior of BT nodes through the Process Control Model . . . . .	142
5.3	Order of operation of the process control model . . . . .	143
6.1	Requirements of the ATP system . . . . .	160
6.2	Issues found during translation of requirements of the ATP System . . . . .	163
7.1	Three approaches to dealing with Legacy systems . . . . .	193
7.2	Mapping VHDL functionality to Behavior Trees . . . . .	195
7.3	Results of Performing Failure Modes and Effects Analysis . . . . .	219
A.1	State Transition Table . . . . .	248
A.2	Requirements of the ATP system . . . . .	260
B.1	Variable Naming Conventions . . . . .	268

B.2 Elements of a Behavior Tree Node . . . . . 269

B.3 Elements of a Behavior Tree Relation . . . . . 270

B.4 Nodes of a Behavior Tree . . . . . 271

B.5 Branches of a Behavior Tree . . . . . 272

# Supplementary Material

This dissertation is accompanied by supplementary material in the form of electronic resources included on an accompanying compact disc. Inserting the compact disc into your computer should automatically open a page in your web browser with links to access the electronic resources. If the page fails to open, the folder structure can be browsed manually. The folder structure and contents of the electronics resources are as follows:

- **Case Studies:** Electronic versions of all work products of the three case studies in the dissertation in Portable Document Format (PDF).
  - Microwave Oven
  - Automated Train Protection System
  - Clock Selector
- **Behavior Tree Traces:** Animated traces demonstrating the semantics of the Behavior Tree language in the Scaleable Vector Graphics (SVG) format. The SVG files should display in most web browsers.
  - Basic Nodes
  - Composition
  - Operators
- **Author's Publications:** The publications of the author in Portable Document Format (PDF).





# Part I

## Introduction & Background



# 1

## Introduction

Marketplace demand is driving the need to develop software-intensive systems of ever increasing scale and to deliver products in shorter time frames, for less cost, and of a higher quality. Managing the complexity created by the increasing scale of these systems is crucial, as it impacts upon the related factors of time, cost and quality. Despite this, mainstream approaches to software and systems engineering (Sw&SE) do not adequately address the issue of scalability.

If scalability is not adequately addressed, it can play havoc with even the most simple tasks. This is easily demonstrated by contrasting a mental arithmetic approach to solving an arithmetic problems with a paper-and-pencil approach. The sum of two 2-digit numbers (e.g.  $39 + 54$ ) can be easily computed using either of the two approaches. The sum of two 7-digit numbers (e.g.  $3452938 + 1295729$ ), however, is much more difficult to solve using

mental arithmetic than using a paper-and-pencil approach. Finally, it is almost impossible for most people to compute the sum of five 7-digit numbers mentally whilst it still remains easily solvable using a paper-and-pencil approach.

The larger scale summations remain solvable with paper-and-pencil because partial results can be recorded on paper. This means the final result can be calculated by repeatedly adding two digits at a time. The paper-and-pencil approach is scaleable in essence because the *local problem space* (adding two digits) remains constant regardless of the *global problem space* (adding two or more numbers). Mental arithmetic does not scale because partial results form part of the local problem space, thereby causing an increase in the global problem space to also result in an increase in the local problem space. The increasing size of the local problem space leads to an overload of the short-term memory because the partial results have to be remembered whilst also trying to calculate the addition of two digits.

Approaches to Sw&SE that do not adequately address scaleability are analogous to mental arithmetic in the previous example. It is possible to proceed directly to a formal specification, a design or even a deployed system if one attempts only a small-scale textbook example with only a few requirements as it is possible to resolve many of the issues informally and intuitively. As the scale increases, however, an informal approach to resolving these issues is no longer feasible as the short-term memory is overloaded by the interdependencies of the requirements.

One result of the struggle of mainstream Sw&SE approaches to manage the complexity of large-scale software-intensive systems is widespread failure of projects. Charette [Cha05] estimates that 5-15% of software projects conducted in 2005 failed as a result of either being abandoned before they were finished or just after delivery due to not meeting the needs of the client. Charette also found that large-scale software projects are three to five times more likely to fail than smaller projects. Unless dealt with appropriately, the exponential rise of complexity created by the need to develop software systems of increasing scale will make the failure of large-scale software projects even more commonplace.

There is therefore an obvious need for a Sw&SE methodology which can handle the complexities of developing large-scale systems. A scaleable methodology for Sw&SE must maintain a local problem space of constant size regardless of the global problem space (e.g.

---

thousands of requirements). This methodology must therefore build a system out of its requirements, using a rigorous approach to translate each requirement individually in a formal and repeatable way until a complete system specification is built. A design that results in the satisfaction of the specification requires a similar approach that individually applies design decisions to the specification until a completed design is built. The resulting design can then be used as the basis for a variety of deployed systems. This end-to-end approach of building a system from out of its requirements differs to mainstream Sw&SE approaches which use a miraculous intuitive leap to proceed from the requirements to a specification, a design or a deployed system.

We propose that the reason that mainstream Sw&SE approaches do not adequately address scalability is that they use representations that prohibit scaleable methodologies. Consider again a paper-and-pencil approach to arithmetic, this time using roman numerals instead of arabic numerals. The task of adding two numbers is now complicated by a mismatch between the representation and a scaleable methodology; since roman numerals are not positional, it is no longer possible to deal with two digits at a time. This mismatch between representation and a scaleable methodology in arithmetic is analogous to mainstream Sw&SE approaches which use representations that do not allow one requirement to be dealt with at a time.

A Sw&SE approach, however, with a representation that enables a scaleable methodology, can dramatically improve our ability to manage the complexity of large-scale systems. By managing this complexity, a scaleable methodology enables systems to be built in shorter time frames, for less cost and to be made at a higher quality.

A scaleable methodology enables shorter timeframes for system delivery by using a divide and conquer approach. Revisiting our arithmetic analogy, if we have 100 numbers to add, one person could perform 99 additions to add the 100 numbers. Alternatively, if ten people add ten numbers each, 90 of these additions can be performed concurrently with the results used to perform the remaining nine additions. Applying this to system design, rather than one person translating 100 requirements it would be possible for ten people to concurrently translate ten requirements each and integrate the results <sup>1</sup>.

---

<sup>1</sup>While this approach would result in a significantly faster translation of the requirements, it requires

A scaleable methodology reduces project cost by supporting the task of risk analysis to mitigate the possibility of schedule slippage and project failure. Risk analysis is improved because a scaleable methodology provides a clear understanding of the status of the project at all stages of development. This clarity is provided by metrics such as the number of requirements translated and the number of defects found, which provide concrete indicators of the status of project and the projected cost of further development.

A scaleable methodology increases project quality by building a system from out of its requirements. This creates work products that have direct traceability to the original requirements and that preserve the original intent and vocabulary of the system analyst. The formalisation process also increases defect detection, allowing defects to be addressed earlier which results in a higher quality final product.

Although mainstream approaches can also enable shorter timeframes using a divide and conquer approach, the divisions are based upon an informal top-down analysis performed at the early stages of development. A lack of understanding of the system being built at these early stages may result in the system being poorly divided. This in turn can cause problems during integration of the parts or can result in the creation of a substandard design.

The miraculous intuitive leap which is used by conventional approaches to proceed from the requirements to a final work product also has detrimental impacts on the cost and quality of a project. Traceability to the requirements is lost which increases the risk that some requirements will not be addressed. Determining whether requirements are addressed necessitates testing which can only be performed during the later stages of development. Quick iterative cycles of development are then performed to mitigate the risk created by the uncertainty of the project status prior to testing. This iterative approach addresses the risk at the expense of an added cost of development and testing. When defects are eventually located using testing, they are usually discovered later in development when they are more costly to fix.

---

additional support for the information that is gathered concurrently to be consistent.

## 1.1 Objective and Approach

The objective of this dissertation is to address the issue of scalability in Sw&SE by providing the foundations for a scaleable, widely applicable, end-to-end methodology. This dissertation will use these foundations to demonstrate the applicability of the approach and the potential benefits it can provide. It is hoped that demonstrating these potential benefits will encourage further research into this area, providing the critical mass required to create a finalised methodology. This approach of driving methodology development from practical experience is also favored by other researchers that study design ([Ale64], Preface):

“... I think it is absurd to separate the study of designing from the practice of design. In fact, people who study design methods without practicing design are almost always frustrated designers who have no sap in them, who have lost, or never had, the urge to shape things. Such a person will never be able to say anything sensible about “how” to shape things either.”

Satisfying the objective of providing the foundations for a scaleable, widely applicable, end-to-end methodology, requires the approach described in this dissertation to have the following characteristics. Firstly, satisfying the objective of providing the foundations for a scaleable methodology requires the approach to begin at the requirements. Then, the local problem space must remain minimised and constant regardless of the global problem size throughout the whole process from the requirements to the final work product. Secondly, to satisfy the objective of an end-to-end methodology, the approach must proceed all the way from the requirements to result in a final work product of a deployed system. This ensures that the approach provides the largest productivity benefits. It also allows the results of the approach to be compared against existing mainstream Sw&SE approaches which result in deployed systems. Thirdly, to satisfy the objective of a widely applicable methodology, the approach must be demonstrated to apply to a wide area of both problem domains and solution domains. This ensures that the effort of creating a scaleable methodology is justified as the benefits of the approach can be widely achieved in numerous areas.

We propose utilizing the Behavior Engineering approach to achieve these objectives rather than independently developing a new approach. Behavior engineering (BE) [Dro06c] is an

integrated approach to systems development initially proposed by Dromey in 2001 [Dro01]. It supports the engineering of large-scale dependable software intensive systems at both the systems and software engineering level. BE has had significant success in industry where it is currently used primarily for requirements analysis of large specifications [Pow07, Bos08].

The success of BE in requirements analysis is primarily due to using a scaleable methodology that deals with one requirement at a time. BE uses a bottom-up process that enables each requirement to be modeled independently and integrated one at a time to form a complete view of the system specification that is built out of the requirements. Combining this scaleable methodology with a rigorous approach to translation results in a model complete with traceability back to the original natural language requirements. This scaleable methodology is enabled by BE's graphical representation that visually captures the information in natural-language requirements as the behavior of a system composed of a set of integrated components. The component interactions are formally defined in a multi-threaded language which co-ordinates the individual behaviors associated with each of the components. The separation of computation from integration using components is an approach gaining traction in the research community, as evidenced by several component-based representations [Par00, Arb04, LLW06].

Current research involving BE focuses primarily on using BE models as a formal specification, which can then be further analysed using techniques such as model-checking. An approach to design with BE has been outlined previously by Dromey [Dro06c], but this muddled the creation of a specification with the task of adding real design decisions beyond the specification. This dissertation provides a new approach to design using BE which separates specification and design into separate stages and clarifies the types of design decisions that need to be made to go from a specification to a design in BE.

Another central part of our approach is to develop the new design stage for BE within a model driven engineering (MDE) framework. MDE<sup>2</sup> is a software engineering approach that advocates a shift from code-centric development to model-centric development, in which models are the primary artifacts of all stages of development. Models help manage

---

<sup>2</sup>For the purposes of this dissertation we will use Model-Driven Engineering as a generic term intended to encompass all model-driven approaches. When discussing a model-driven approach individually, its particular term will be used.



complexity by focusing on one aspect of a system, ignoring other irrelevant details which may hinder the understanding or manipulation of the aspect of interest.

BE can be integrated into an MDE framework by considering the system behavior captured by BE's graphical representation as a modeling dimension. MDE captures the information of a modeling dimension by creating domain specific languages (DSLs) that utilise the notation and abstractions common to the domain. By using the terminology of the domain, the usage of DSLs often encourages domain experts to play a greater role in software development. This benefit, however, must be weighed against the cost of designing, implementing, and maintaining a DSL and the tools to support its use.

Integrating the BE approach into an MDE framework provides benefits for both BE and MDE. BE gains the productivity benefits provided by MDE's leveraging of the increased abstraction provided by models to improve the development and maintenance of software systems. MDE benefits by gaining a language for defining system behavior with an end-to-end scaleable methodology that proceeds all the way from requirements to a deployed system without any miraculous intuitive leaps. This is useful because the use of abstraction in MDE only decreases the likelihood that scaleability will be an issue. Creating larger models in a DSL can still cause increased complexity which can only be prevented if a scaleable methodology is used to build the model.

The choices for development of the BE DSL for system behavior (and any MDE DSL) can be broadly categorised into two approaches [MHS05]: *language exploitation* and *language invention*. Language invention involves developing a new language from scratch. Language exploitation creates a DSL by extending an existing general purpose language (GPL), a language that uses generalised constructs to cut across multiple modeling dimensions.

Language invention is used to develop a DSL when existing languages cannot be used to capture domain-specific concepts. Language invention creates DSLs that provide the most benefit to DSL users but they also require the development and maintenance of completely new tools such as an editor, compiler, debugger and simulator. Language invention is most suited to DSLs with a focus on syntax over semantics, with semantics often implicitly defined by model compilers or generators [HM07]. A DSL for capturing system behavior is not particularly suited to language invention, as there is a greater emphasis on the semantics of

domain concepts in addition to their syntax and structure.

Language exploitation is the most cost effective approach to developing a DSL for a number of reasons. DSLs created by language exploitation can leverage any existing technology developed for the chosen GPL to simplify the implementation and mapping between multiple DSLs. It also allows the utilization of existing editors and the potential to benefit from a user's familiarity with the existing GPL. The ultimate success of language exploitation, however, depends on the effectiveness of the chosen GPL at capturing the concepts of the DSL.

Our approach creates the BE DSL for system behavior using language invention because the concepts of BE are unsuited to be captured by existing GPLs. Whilst using language invention to create the BE DSL, however, we also develop BE into a GPL for creating other system behavior DSLs using language exploitation. This enables a family of languages, all based upon the same BE GPL, which allows existing tool support and user familiarity with the BE GPL to be exploited.

This concept of a family of languages is also one of the goals of the unified modeling language (UML), an object-modeling notation that is widely regarded as [Rat05], "... an industry-standard language that allows us to clearly communicate requirements, architectures and designs". In contradiction to its wide adoption as an object-modeling notation, however, UML is widely criticised as an unsuitable GPL for MDE [LCM06, FS06, Tho03]. The wider adoption of UML as a host GPL for creating DSLs by language exploitation is hindered by problems with its representation, its methodology and its extension mechanism. These problems, which we discuss in detail in the next chapter, justify the need for a BE GPL for MDE.

Developing BE into a GPL for use in an MDE framework primarily requires an extension mechanism for defining DSLs using language exploitation. To gain the most benefit from MDE, however, a BE model resulting from the new design stage should also be executable. This requires the BE component model to be specified, as previous BE research has not required components with executable behavior, but have instead represented components as variables with an enumerated state [GLYW05]. A toolset consisting of an editor, model transformations, code generation and a virtual machine is also required to execute the BE

models.

After laying these foundations, the remainder of the dissertation is used to demonstrate the wide applicability of this approach in order to establish the suitability of BE as a GPL. We ensure the coverage of several modeling dimensions by utilizing Bézivin’s categorisation of three types of MDE applications, namely forward modeling, interactive modeling, and reverse modeling<sup>3</sup> [BBJ07]. Forward modeling involves the most common form of modeling in MDE, which is to create a system from a model. Interactive modeling consists of a system and a model co-existing with each other, with any changes in one influencing the other. Reverse modeling involves the creation of a model from an existing system.

Three case studies<sup>4</sup> were selected within this framework to show a wide cross section of concerns in the addressed modeling dimensions. The three case studies cover the subject area modeling dimensions of consumer electronics, transportation and avionics. They also cover the system aspect modeling dimensions of embedded systems, software/hardware integration and development of field programmable gate arrays from legacy hardware.

## 1.2 Thesis Hypotheses

This dissertation defends the following theses:

1. Mainstream Sw&SE approaches, generally, do not adequately address scalability.
  - (a) These approaches commonly use abstraction to decrease the likelihood that scalability will be an issue.
  - (b) These approaches do not build a system from out of its requirements and it is common for a miraculous intuitive leap to occur between the requirements and the specification, design or deployed system.

---

<sup>3</sup>Bézivin used the term forward engineering and reverse engineering instead of forward modeling and reverse modeling. He also used the term models-at-runtime as opposed to interactive modeling, though the central concepts of these terms remain unchanged.

<sup>4</sup>The term ‘case study’ as used throughout this thesis, is not intended in the sense of the research methodology of the same name (the case study research methodology [Yin09]). Instead, each case study in this dissertation should be considered as a worked example that demonstrates the principles upon which it is based.

2. Scaleability can be addressed better by:
  - (a) Using an approach with a minimised local problem space that remains constant regardless of the size of the global problem space.
  - (b) Using an end-to-end approach that begins at the initial requirements and proceeds all the way to a deployed system without any miraculous intuitive leaps. An end-to-end approach provides the maximum productivity benefit and discourages the need for intuition to be used in the approach.
3. Extending BE to have a design stage which is developed within an MDE framework will leverage the scaleability of BE and the productivity benefits of MDE to:
  - (a) Create an end-to-end methodology.
    - i. The system behavior modeling dimension of BE is suited to describing a system at the stages of requirements, specification, design and deployment.
  - (b) Create a widely applicable approach.
    - i. The system behavior modeling dimension of BE is already widely applicable to a multitude of problem domains.
    - ii. The applicability of the BE approach can be improved further by using an extension mechanism to create domain-specific languages defined by BE.
    - iii. The MDE concept of platform-independence and transformations can improve the applicability of the approach to a larger area of solution domains.
  - (c) Create a scaleable methodology.
    - i. BE is already proven to be scaleable for requirements analysis of large-scale systems in industry.
    - ii. A scaleable methodology for design with BE can be developed by extrapolating the principles that make BE scaleable for proceeding from requirements to a specification and adhering to them whilst proceeding from a specification to a design.

## 1.3 Contribution

The contribution made by this dissertation is summarised below:

1. The existing BE approach has been extended to support design and deployment.
  - (a) The stages of specification and design have been clearly separated and the design decisions required to proceed from a specification to a design are outlined.
  - (b) A component model for BE has been defined that describes how executable components can be integrated by a deployed BE model.
  - (c) An extension mechanism is described that enables DSLs to be created that use BE as a host GPL.
  - (d) A toolset that consists of an editor, model transformations, code generation and a virtual machine has been developed to support the deployment of BE models.
2. The resulting approach provides a new GPL for MDE with the following characteristics:
  - (a) It uses a visual language with a small core notation with formal, well-defined semantics.
  - (b) It is linked to a scaleable methodology that is applicable all the way from requirements, through specification and design to a deployed system.
  - (c) It can be applied to Bezivin's [BBJ07] three types of modeling within MDE:
    - i. Forward modeling is demonstrated by deploying a BE model on an embedded controller.
    - ii. Integrated modeling is demonstrated by investigating early-stage hardware/-software design by combining this approach with Modelica, a mathematical modeling language.
    - iii. Reverse modeling is demonstrated by building a BE model from an existing legacy system that is then deployed on a new platform of a Field Programmable Gate Array.

3. A new approach to integrated hardware/software development is introduced called co-modeling. Co-modeling is intended to supplement the existing practice of co-design which is applied later in the later stages of development. Co-modeling has the following characteristics:
  - (a) It provides a framework to investigate the suitability of design decisions applied to a specification consisting of a mixture of hardware and software components.
  - (b) It helps locate and resolve defects across the software/hardware boundary early in development when they are cheaper and easier to resolve.
  - (c) Simulation of co-modeling scenarios provides a graphical and documentable result that can be used to determine the specifications of sensors and actuators operating in a range of environmental conditions.

## 1.4 Thesis Structure

This dissertation is organised into five parts with a total of eight chapters. **Part I - Introduction and Background** is comprised of four chapters, beginning with this chapter, *Chapter 1: Introduction*.

*Chapter 2: An Overview of Model-Driven Engineering* provides a more in-depth discussion of the state of the art in MDE that forms the foundation for the objectives and approach of this dissertation. This discussion begins by defining the goals of MDE, and exploring their origins. Existing approaches to MDE are then investigated with a focus on their ability to capture system behavior. The appropriate representation, methodology and extension mechanism of a new MDE GPL for system behavior is then discussed.

*Chapter 3: Introduction to Behavior Engineering* provides an introduction to BE for readers not already familiar with the approach. Readers who are familiar with BE may prefer to omit this chapter. The chapter details the representation and process of BE by means of a case study involving modeling the requirements of a simple microwave oven. The case study is used to demonstrate how BE is used to formalise natural language requirements by translation; perform a fitness for purpose test by integrating the formalised requirements;

and develop a specification by locating and correcting defects in the integrated requirements. Related research in the area of Behavior Engineering is also discussed.

**Chapter 4: Design with Behavior Engineering** describes an extension to BE which provides better support for design. This chapter begins by addressing why BE requires a design stage as opposed to using a BE specification with other conventional design approaches. Following this the BE component model and extension mechanism are described. The chapter concludes by describing the process of the design stage which is illustrated by continuing the design of the microwave oven from the specification produced in Chapter 3.

**Part II - Forward Modeling** discusses the creation of a system from a model, which is perhaps the most common use of MDE. **Chapter 5: Behavior Engineering of Embedded Systems** concludes the microwave oven case study by illustrating its deployment on an embedded controller. This illustrates that BE can be utilized for the system aspect of embedded systems. This chapter also introduces the toolset which has been created to support design with BE. The toolset consists of an editor for describing BE models; model transformations to make BE models executable; and a virtual run-time environment for executing BE models tailored specifically for embedded systems.

**Part III - Integrated Modeling** deals with the difficult case of a model and a system co-existing and influencing each other. This form of modeling often gives rise to wicked problems: problems with complex interdependencies such that the solution of one aspect of the problem often creates or reveals previously unidentified problems in other aspects. **Chapter 6: Co-Modeling with Behavior Engineering and Modelica** introduces an example of integrated modeling that occurs when developing a system composed of integrated software and hardware components. This is often addressed by co-design, which deals with partitioning functionality onto a mixed architecture of software executing on a central-processing unit and specialised hardware implemented on a field programmable gate array. This dissertation introduces co-modeling, which is a new approach that also deals with integrated software/hardware design but at an earlier stage of development than co-design. Our approach to co-modeling is introduced which integrates BE with Modelica, a mathematical modeling language. Co-modeling is demonstrated with a case study involving the development of an automated train protection system which monitors a train driver and

takes control of the train if a dangerous situation is not met with an appropriate response. The developed co-model is used to investigate co-modeling scenarios and to determine the effect various combinations of sensors, actuators and hardware platforms have on the system's behavior. The case study also demonstrates the Behavior Run-time Environment, which differs from the embedded version by utilizing attributes, an expression parser and parameterized messages. The chapter concludes with a discussion of how co-modeling can be a useful tool in building large-scale systems.

**Part IV - Reverse Modeling** deals with creating a model from an existing system. A common application of this form of modeling is to model legacy systems which can then be migrated to a new platform. ***Chapter 7: Behavior Engineering of Hardware Systems*** introduces a migration approach to legacy systems in which a BE model is used as an intermediary. This chapter uses a real case study from industry. The case study deals with a small portion of outdated circuitry built with transistor-transistor logic (TTL) integrated circuits (ICs) that need to be replicated in a field programmable gate array (FPGA). The circuit schematic is captured in a BE model by using a domain-specific extension. The BE model is then used to generate VHSIC hardware description language (VHDL), a hardware description language that is used to define a FPGA. To demonstrate the advantage of using BE as an intermediary (as opposed to a direct mapping from a domain-specific language) the BE model is used to perform failure mode and effects analysis, a procedure commonly applied to BE models developed from software requirements.

**Part V - Discussion** comprises ***Chapter 8: Discussion*** which concludes this dissertation. This chapter gives an overview of the material presented to elicit the contribution provided by this dissertation. Ideas for future work are also discussed.



# 2

## An Overview of Model Driven Engineering

In this chapter we review existing model driven engineering (MDE) approaches for developing software-intensive systems. We analyse the suitability of the languages used by these approaches, particularly for capturing semantically-rich system behavior.

This chapter is organised into three sections: The first section of this chapter investigates the origins and goals of MDE in more detail. The core concepts of MDE and the reasons for their emergence are described, and whether MDE has the potential to live up to its claims. The second section is a review of the state of the art approaches to MDE. The review focuses on the model driven architecture (MDA), a prevalent view of MDE. Three different realisations of MDA are discussed in conjunction with other related approaches. The third section argues that there is a need for a new general-purpose language for capturing the modeling dimension of system behavior. The section elicits the special characteristics of

this modeling dimension that warrant the development of a new general-purpose language in an MDE framework. It addresses the deficiencies of the current MDE approaches at capturing this modeling dimension by investigating what representations and methodologies are required for a new MDE GPL for capturing system behavior.

## 2.1 Origins and Goals of MDE

In the Introduction, we defined MDE as a model-centric approach to software development. Rothenberg vividly describes the utility of modeling [Rot89],

“Modeling, in the broadest sense, is the cost-effective use of something in place of something else for some cognitive purpose. It allows us to use something that is simpler, safer or cheaper than reality instead of reality for some purpose. A model represents reality for the given purpose; the model is an abstraction of reality in the sense that it cannot represent all aspects of reality. This allows us to deal with the world in a simplified manner, avoid the complexity, danger and irresistibility of reality.”

The concept of making models central to software development is relatively recent compared to the concept of modeling itself. The use of models can be traced back as far as the ancient Greeks, in writings by Plato describing the function of blood vessels by means of an analogy to irrigation channels<sup>1</sup>([Jow53], p729 originally in [Béz05], p175).

Gitzel [Git05] traced the origins of MDE concepts as far back as Zachmann [Zac87], who advocated a separation between the technical and domain-specific elements of software architecture. A more commonly recognised starting point, however, is computer aided software engineering (CASE) which gained popularity in the 1980’s [Git05, HT06]. CASE [Fug93] involved the development of applications to support and partially automate software development. CASE used graphical representations such as state machines, statecharts, data-flow diagrams and entity-relation diagrams to perform systems analysis. CASE tools then used the diagrams created during analysis to generate implementations.

---

<sup>1</sup>“... these courses, detained as in a vast river, neither overcame nor were overcome; but were hurrying and hurried to and fro, so that the whole animal was moved and progressed, ...”

Many researchers are concerned that CASE is considered to be a forerunner of MDE [HT06, Sch06], because CASE is widely regarded as having failed to meet its goals. Hailpern and Tarr [HT06] attribute CASE's inability to adequately address the silo problem as the central reason for its failure. The silo problem involved the creation of model silos, where each silo would contain models that represented a particular stage of software development. Rather than being able to use a single model throughout development, a model became disjointed across several silos created by the poor integration of the different CASE tools being used. The silo problem was common across different CASE vendors, but even occurred in environments with several tools from the same vendor.

Other reasons for the failure of CASE include [Sch06]: the models mapped poorly onto the underlying platforms; generating code that was unnecessarily complex and verbose instead of simplified by using frameworks; CASE tools targeted proprietary execution environments; the tools were unable to scale to larger systems, and; the CASE tools were hard to develop, debug and evolve. Many of these issues are no longer a problem with MDE. Standards increase the chance of interoperability between different tool vendors, and technology has improved since the failure of CASE [HT06]. A key example of this is Eclipse [CR06], an open development platform which provides extensible frameworks, tools and runtimes for software development.

The principles of MDE also differ slightly from CASE. While many MDE approaches use graphical representations, it is not a fundamental principle of MDE. Bezivin states the central principle of MDE is that [BBJ07], "A system lies in the real world while a model lies in the modeling world.". This principle relies on two axioms. The first axiom is representation: Any kind of system can be represented by models. The second axiom is conformance: Any model syntactically conforms to a metamodel, a model that describes the concepts of other models. Bezivin uses these two axioms also as the basis for the two basic relations of MDE [BBJ07]: representation (*repOf*) and conforms to (*c2*). Barbero et al. [BJGB07] add a third relation, extension, where new models are derived from base models using an extension mechanism. Thomas [Tho03] suggests that there are three MDE relations: *isBasedOn*, *isLike* and *isRepresentedBy*. These relations appear to conform to Bezivin's relations, with *isBasedOn* and *isLike* being aliases for the conforms to relation and

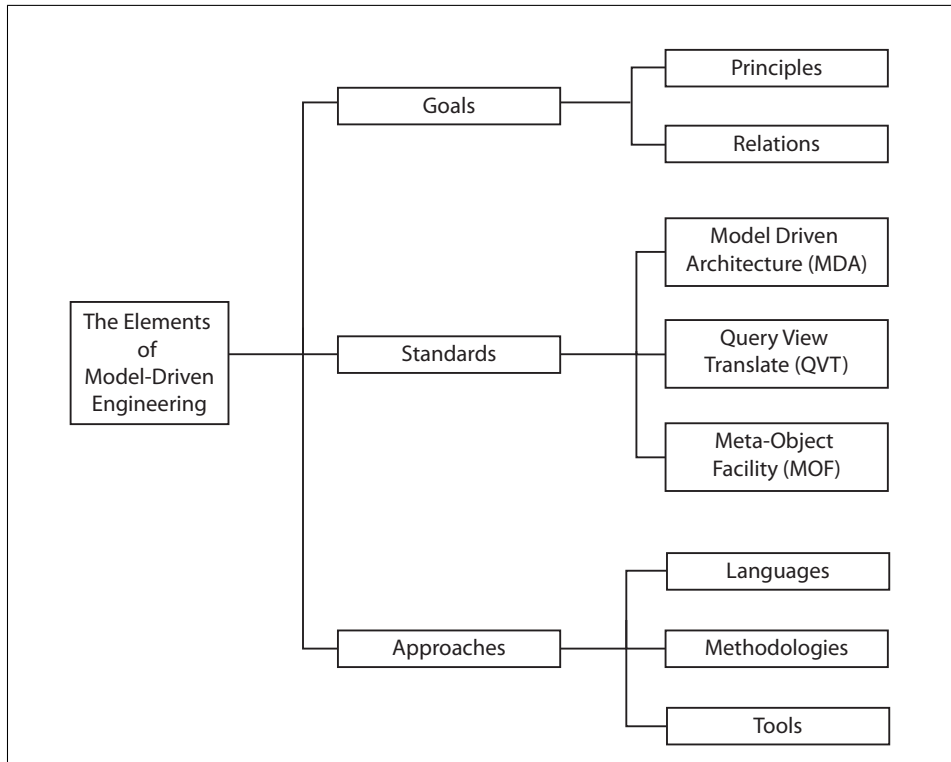


FIGURE 2.1: An Overview of the elements of MDE

isRepresentedBy matching repOf.

These MDE principles and relations are used to support the goals of MDE. In the Introduction, we stated the goal of MDE is to utilize models to increase abstraction thereby improving the development and maintenance of software systems. This goal is widely recognised in the literature. Gitzel [Git05] states the goal of MDE is to reduce the cost of software by dealing with complexity using DSLs and by reducing the impact of change using automation and model transformations. Hailpern and Tarr [HT06] rank raising abstraction as the central goal of MDE, with the sub-goals being to improve software quality and software reuse.

MDE is made up of several elements which work together to achieve these goals. Giudice [Giu07] organises these elements into models, standards, tools, architecture, and process. Schmidt [Sch06] focuses on DSLs and transformation engines and generators. In the MDA manifesto, Booch et al. [BBI<sup>+</sup>04] state the three foundations of MDE are direct representation, automation and open standards. Bezivin and Barbero [BBJ07] use the categories of principles, standards and tools.

Figure 2.1 provides an overview of our categorisation of the elements of MDE. We separate MDE into three areas: Goals, Standards and Approaches. The goals of MDE are enabled by the principles and relations of MDE. These, in turn, give rise to standards which describe generalised strategies for performing various portions of MDE. The standards are implemented by approaches which consist of a language, methodology and tool support for performing MDE.

For the purposes of this dissertation we will focus primarily on MDE approaches. The next section discusses existing approaches to MDE.

## 2.2 Model Driven Engineering Approaches

The broad principles of MDE give rise to many approaches. Some notable approaches<sup>2</sup> include Agile MDE [Amb07, Mel04], Model-Driven Software Development [Bet04, SVC06], Model-Integrated Computing [Dav02], Model-Driven Architecture [Fra02, MKUW04], and Software Factories [GSC<sup>+</sup>04]. Disputes about the direction each of these approaches takes MDE is also common. For example [Gut04a, Co04b, Gut04b], advocates of MDA (a MDE standard managed by the object modeling group (OMG)) and advocates of Software Factories (an MDE approach based on DSLs by Microsoft) disagree on how central the role of standards should be in MDE.<sup>3</sup>

Out of these many approaches, we have chosen to focus on approaches based on the MDA standard. While MDA is only one of the many visions of MDE, it is also currently the most prevalent. The vision of MDA is to create a standard engineering approach to automate the task of describing business processes in software. It is proposed that this will be achieved by separating the business or application logic from technological details about the platform on which it is implemented. By doing this MDA claims it can increase tool

---

<sup>2</sup> A more complete but still non-exhaustive list of model-driven acronyms (with or without associated processes) includes [Béz06]: Model Engineering (ME), Model Driven Architecture (MDA), Model Driven Development (MDD), Model Driven Software Development (MDSD), Model Driven Software Engineering (MDSE), Model Management (MM), Model Driven Data Engineering (MDDE), Architecture Driven Modernisation (ADM), Model Driven Reverse Engineering (MDRE), Domain Specific Language (DSL) and Domain Specific Modeling (DSM).

<sup>3</sup>This dispute may have been resolved recently in favor of MDA with Microsoft researchers returning to work with the OMG on revising UML [Coo08].

and platform interoperability, thereby increasing software reusability [Git05]. Siegel [Sie02] claims a wider set of benefits including: automating the transformation from a business model to an implementation; guaranteeing that the requirements in the model will be in the final implementation; and generated code is derived from patterns designed by the industry's best developers.

The MDA has two types of models<sup>4</sup>: a platform independent model (PIM) and a platform specific model (PSM) ([MKUW04], p34). A PIM is a model of the subject matter of concern described independently of any implementation details. A PSM adds technical details to the PIM making it suitable to be implemented on a specific platform. An MDA process then consists of modeling a PIM, transforming it to a PSM and deploying the system. Another way of viewing this is linking a set of modeling standards such as the UML, meta object facility (MOF), XML metadata interchange (XMI), common warehouse metamodel (CWM), software process engineering metamodel (SPEM) to a set of middleware standards such as extensible markup language (XML), java platform enterprise edition (J2EE), enterprise javabeans (EJB), microsoft .NET framework (.NET), and common object request broker architecture (CORBA).

Transformations defining how to convert a PIM into a PSM are central to MDA and can be categorised as being either elaborative or translative [MKUW04]. Elaborative approaches describe static aspects in a PIM which are then annotated with information to make a PSM. These approaches typically generate about 50-60% of the total code, with the rest of the code created by the developer. Round-trip problems are common with elaborative approaches if generated code is changed, and then regenerated from the PIM. Translationist approaches provide full code generation by avoiding using a PSM, translating directly from a PIM to code. Because these approaches avoid using PSMs, they currently require a model compiler for each platform.

According to Cook [Coo04a], MDA is realised by three different approaches:

---

<sup>4</sup>A third type of model does exist in MDA called a computational independent model (CIM). A CIM describes the business context and the business requirements independent of how these details will be computationally realised in software [Obj03]. CIMs have received little attention by researchers in comparison to PIMs and PSMs because many researchers do not believe automatic transformation from a CIM to a PIM is possible ([KWB03], p19).

- **The UML Platform-Independent Model:** The UML is used as a general-purpose language for defining a PIM, and a language exploitation technique called UML Profiles is used to create a family of languages.
- **Meta-modeling and the Meta-Object Facility:** Domain-specific languages are defined using metamodels, which themselves are defined by a meta-metamodel conforming to the MOF standard.
- **Executable and Translatable UML:** A subset of the UML with executable semantics and an action language that is used to create executable models.

Figure 2.2 categorises these three approaches based upon how they utilize the MDE concept of a domain specific language. We will now discuss each of these three realisations of the MDA in detail.

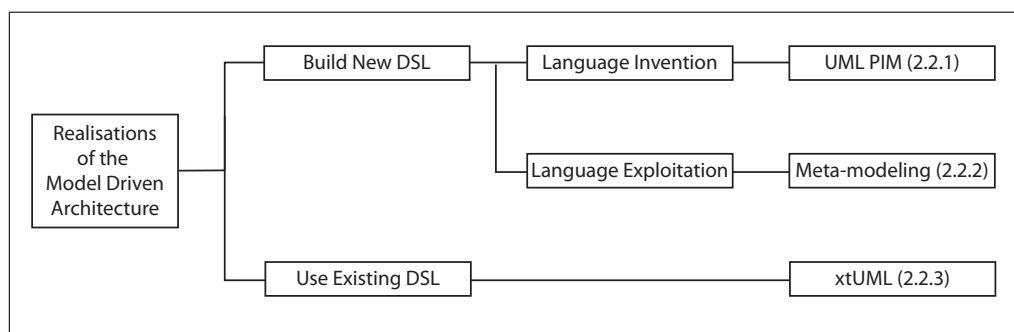


FIGURE 2.2: MDA Approaches

### 2.2.1 Meta-Modeling and the Meta-Object Facility

This approach to MDE uses metamodels to define DSLs. As defined earlier, a metamodel is a model that defines the concepts of other models. A meta-model can be considered analogous to a legend on a map. Much like a legend describes the meanings of any symbols used on the map, a metamodel defines the concepts of any models which conform to it.

The idea of a metamodel becomes more complex when we consider what defines the concepts of a meta-model. This role is fulfilled by a meta-metamodel, which conforms to the MOF. The MOF is a standard for defining meta-metamodels, that can then be used

to define metamodels, such as the UML metamodel. The MOF standard is conformed to by metamodeling implementations such as the Eclipse Modeling Framework [Ecl09] and the Graphical Modeling Environment [Ins08].

The MOF defines metamodeling using four levels. Level M0 consists of actual instances of the model, or the real world system that is captured by the model. Level M1 is the classes of the model, or the model itself. Level M2 is the metamodel, which describes the concepts of the model. Level M3 is the meta-metamodel used to define all M2 metamodels.

Defining a model completely requires a metamodel (Level M2) consisting of five parts: an abstract syntax, well-formedness rules, a concrete syntax, semantics, and mappings. MOF only defines the abstract syntax of the metamodel by defining the concepts of the model and their relations to each other. Well-formedness rules are defined by placing constraints using the object constraint language (OCL): a formal textual language for describing constraints and queries on models that is compatible with the MOF. A concrete syntax defines what a model looks like, which is undefined by MOF. Semantics are normally defined informally in transformations which convert the model into an executable format. Mappings can be defined internally between parts of a metamodel, and; externally between metamodels. Internal elements of a metamodel requiring mappings to each other include the concrete and abstract syntax and the abstract syntax and the domain in which the semantics of the concepts are defined. External mappings between metamodels are supported by the query view transformation (QVT) standard [Obj08a], allowing information between different metamodels to be shared. External mappings do not just map from a PIM to PSM, but map between the metamodels of many different modeling dimensions. This is because many advocates of metamodeling find that the platform modeling dimension in MDA has a too central role ([CSW08], p13).

It is common for most metamodels to be incomplete, only providing some of the five parts of a complete metamodel. Clark et al. ([CSW08], p25) assess the quality of a metamodel on a five level scale. At level 1, the abstract syntax is defined but has not been tested; the semantics are informal and incomplete, and; no well-formedness rules are defined. At level 2, the abstract syntax is complete and partially tested; well-formedness rules are also defined. At level 3, the abstract syntax is completely tested and the concrete syntax is described



informally. At level 4, the concrete syntax is formalised and tested; semantic models are created. At level 5, all aspects of the metamodel are complete including an executable semantic model.

A more recent development in metamodeling is language driven development (LDD). The concept of LDD arose from language oriented programming (LOP), a term first used by Ward [War94] to define the development of domain-oriented and formally specified very high level languages which are suited to a particular programming problem. Ward advocates creating a DSL as a two stage process. Firstly the language is designed by defining both its syntax and its formal semantics. Secondly, the language is implemented in an existing programming language<sup>5</sup>. Advocates of LOP, such as Dmitrev [Dmi04], consider the language exploitation approach to building DSLs using domain-specific extensions of a GPL problematic. Dmitrev justifies this as being due to the complicated mapping of domain-specific concepts and operations into an existing language, the lack of support for domain-specific extensions by tools made for the GPL, and the verbosity of the programs when defined in the GPL. This is evidenced by the examples of graphical user interface libraries and database libraries, such as the Swing GUI library in Java. Of course, the validity of this criticism depends upon the degree of mismatch between the syntax and semantics of the DSL and those of the GPL.

LDD is supported by integrated development environments called Language Workbenches [Fow05], which implement the concepts of LOP to simplify the design and integration of DSLs. Language workbenches define a DSL in three parts: schema, an editor and a generator. The schema defines the abstract syntax of the language, projections of which are used in the editor. The generator is used to define the semantics of the language. Language workbenches remove the need to write a parser because editing is done using the abstract syntax, and they provide the DSL with its own integrated development environment if an editor is defined. Examples of Language workbenches include Intentional Software[Int09c], JetBrains Meta-Programming System [Jet09], and Microsoft's Software Factories [GSC<sup>+</sup>04].

---

<sup>5</sup>Ward's definition of LOP is a language invention approach, however LOP is also sometimes used more broadly to encompass both language invention and language exploitation (For example, [Fow05])

### 2.2.2 The UML Platform-Independent Model

This approach to MDA uses UML to define a PIM. The UML originated from the Rational Modeling Language made by combining the competing object modeling notations of the three amigos of Rational Software: Grady Booch, Jim Rumbaugh and Ivar Jacobson [Tho03]. In 1997, UML became an open standard for software development managed by the Object Management Group (OMG) [LCM06]. The UML is now widely regarded as, “... an industry-standard language that allows us to clearly communicate requirements, architectures and designs” [Rat05]. For UML to define a PIM, however, it can no longer be considered as just a visual object modeling notation. For UML to be a suitable representation for defining a PIM requires: changes to the UML; a means of mapping from a PIM to a PSM, and; an extension mechanism to support creating DSLs.

The first requirement of changing the UML was started in 2004, when the OMG released UML2, a major revision of the UML. The main reasons for the UML2 revision was to update UML to [Sel06]: support MDA by handling different views and abstraction levels; support domain-specific extensions; increase the precision of UML, and; improve the organisation of the language itself.

UML2 separates the UML specification into two parts: the Infrastructure and the Superstructure. The infrastructure defines the UML metamodel which forms the conceptual foundation of the UML. The superstructure defines the notations and semantics of the UML concepts. In the current version, UML2.2, the infrastructure specification is 226 pages and the superstructure specification is 740 pages [Obj09d]. In addition to the UML specification, there are the related specifications of the Object Constraint Language [Obj06] which is required to specify constraints on UML models, and the UML Diagram Interchange [Obj07] which describes means for communicating UML models for achieving tool interoperability. Finally, executable UML models require an action language. The UML specification defines an action semantics for UML, but not a concrete semantics. Several academic and proprietary action languages exist but lack formal semantics [CD08],

“UML foundation must be accompanied by an action language, preferably one which has been formally specified. To the best of our knowledge, no formal

semantics for any UML 2 action language yet exists.”

The UML2 superstructure [Sco04] consists of thirteen diagrams which define the structure, behavior and interaction of software systems. Structure is defined using the class diagram, component diagram, composite structure diagram, deployment diagram, object diagram and package diagram. Behavior is defined using the activity diagram, state machine diagram and use case diagram. Interaction is defined using the communication diagram, interaction overview diagram, sequence diagram and timing diagram.

Due to the large size of UML2 and to better enable tool interoperability, UML2 has four types of compliance and four compliance levels. The four types of compliance are [Sel06]: compliance to the abstract syntax; compliance to the concrete syntax; compliance to both the concrete and abstract syntax, and; compliance to the concrete syntax, abstract syntax and the diagram interchange standard. The compliance levels are defined using language units, which are sub-languages of UML suited to modeling specific aspects [Sel06]. The compliance levels are also hierarchical, compliance with a higher level requires compliance to all the levels below. The four compliance levels are [Obj05]: Level 0 consisting of class-based structures; Level 1 with the addition of uses cases, interaction, structure, actions and activity language units; Level 2 with the addition of state machines and profiles; and level 3 the addition of Information Flows, Templates, and Model Packaging language units. Despite these changes to improve tool interoperability, no tools currently conform to the complete UML specification and no reference implementation exists to check for conformance. In fact, Suss contends that [SFP08], “During the closing panel of the last UML conference, the experts agreed that no tool was ever going to implement the UML 2.0 completely”.

The changes of UML2 still have not provided a complete formal semantics of UML or caused a high adoption of formal usage. The UML specification remains defined by a mixture of an MOF metamodel, semi-formal OCL and informal textual descriptions [SFP08]. The precise UML (pUML) group [Pre09] was originally formed in 1997 to deal with the imprecision of UML [SFP08],

“The precise UML group, a think-tank of computer scientists, has worked for ten years on defining the precise meaning and consistency of UML and has yet to

deliver a joint and clear statement. ”

More recent attempts regarding UML semantics focus on enabling executable models by defining precise subsets of the UML. An example is Executable UML, which is discussed in the next section. Another is a proposal by the OMG for a foundational subset for executable UML models (fUML) [Obj08b]. When finished, the fUML subset is planned to be used to define higher-level formalisms of the complete UML [CD08].

Even if UML eventually is defined by a precise and formal semantics, most usage of UML is informal [LCM06] which is not suited to MDE. A reason for informal usage may be due to the size and complexity of UML2.0 itself. France et al. [FS06] discuss this accidental complexity of UML, often referred to as UML bloat. UML bloat was meant to be addressed in the UML2 revision by reducing the set of modeling concepts required for the wide application of UML. UML2 attempted to address this with levels of conformance, but in France et al.’s opinion [FS06]:

“The relative novelty of UML modeling can require users to understand most if not the entire language before determining what models are appropriate for their environment” .

UML2 actually introduced a new form of UML bloat for model transformation designers who must operate at a meta-model level. The UML “MetaMuddle” of the UML2.0 metamodel causes two problems [FS06]:

1. It is hard to comprehend the entire UML2.0 metamodel: “The complexity of the current UML 2.0 metamodel can make understanding, using, extending, and evolving the meta model difficult” [FS06].
2. It is hard to use only a part of the UML2.0 metamodel: “The task of identifying and extracting the required subset of concepts from the UML meta model must currently be performed by manually navigating through the ‘metamuddle’” [FS06].

For UML to be used as a PIM, it requires an extension mechanism for creating DSLs. According to Selic [Sel06], “From its inception, UML was conceived as a platform for a

family of related modeling languages - languages that share a common semantics framework and possibly, a common notation.”. UML2 supports creating domain-specific languages by language exploitation using UML Profiles. UML Profiles are a light-weight extension mechanism and therefore cannot modify the existing UML metamodel but can only adapt concepts that are already present for a particular domain or platform. Concepts can be adapted to use new terminology or new syntax, to add different notation for existing symbols, to add semantics that do not exist in the meta-model or are unspecified, and to add constraints to how to use the metamodel ([Obj09c], p117).

The primary extension mechanism of UML profiles is stereotypes. Stereotypes are a type of class used to extend metaclasses in the UML metamodel. Stereotypes also have tagged values which are attributes that are required to be added to the extended metaclass. A stereotype is shown graphically as a name in double angled brackets (<<>>) above the name of the element. UML Profiles can also include filtering rules to hide metamodel elements, but it is not possible to remove the elements from the metamodel. Examples of UML Profiles include the OMG systems modeling language (SysML), the UML Profile for CORBA and the UML Profile for modeling and analysis of real-time and embedded systems (MARTE) [Obj09a].

Selic [Sel07] suggests a two-stage process of developing UML profiles by firstly creating a domain metamodel (as per the previous section) and then mapping this metamodel to the UML metamodel to create a UML profile. Mapping the domain concepts in the domain model to the UML metamodel is performed as follows [Sel07]:

1. Select a base UML metaclass with semantics similar to the domain concept.
2. Check the constraints of the base metaclass (and any superclasses) to ensure no conflicts exist.
3. If needed, refine the attributes of the base metaclass.
4. Check the associations of the base metaclass to ensure no conflicts exist. If they do, check if it is possible to remove the associations with a constraint.

UML profiles can theoretically be used to map a UML PIM to a PSM, by marking the

UML PIM with stereotypes of a UML profile. Marking models, however, are still an active area of research [Sco04],

“Today, there’s no ‘theory’ of marking models that states exactly what should go into a marking model or how to create one. Indeed, it’s not completely cut-and-dried what should be thought of as a mapping, a mark, or a mapping function.”

Lano ([Lan05], Section 5.10) discusses informal rules for transformation of a UML PIM of a scrabble system resulting in a PSM for generating Java code. Thomas [Tho04] states that while transforming a PIM into a PSM is possible for restricted behavior such as state machines, it is non-trivial or impossible to make PSMs for complex platforms such as J2EE and .NET.

### 2.2.3 Executable UML

Executable UML (xtUML) [MB02, SVC06, RFW04] is a descendant of the Shlaer-Mellor object-oriented methodology [SM92]. xtUML is a subset of UML that supports executable models by making small changes to the semantics of some diagrams and by the addition of an action semantics. Mellor [Mel04] claims xtUML is a profile for the UML that defines execution semantics for a subset of UML, suitable to be used as a PIM. Mellor [Mel04] also refutes the need for a PSM, using a translational approach to map directly from an xtUML PIM to code. The translational approach of xtUML uses a model compiler to create code suited to a target platform. Multiple platforms are supported by associating each platform with its own model compiler. Model compilers describe how to implement the xtUML PIM in the target platform [Mel07a],

“We base the concept of a model compiler on the observation that to understand the fundamental architecture of a system, we need only to understand the structure of each of its prototypical elements and how they interrelate ... We don’t need to understand the details of the semantics of an application - in fact they are a hindrance.”

The complete list of UML diagrams that can be used with xtUML are ([Mel07a], Chap. 2): Use Cases, Sequence Diagrams, Domain Charts, Class Diagrams, Statechart Diagrams,

Class Collaboration Diagrams and Action Language. It is more common, however, to create xtUML models with a minimal set of diagrams consisting of a Class Diagram, Statecharts, and an Action Language [Mel07a]. Use Case diagrams and Sequence Diagrams are used in xtUML, but because they are informal, they do not form part of the executable model. The Domain Chart used in xtUML is similar to the Package Diagram of UML. Class Diagrams are used to define the structure on which the executable behavior takes place. Statechart Diagrams are used by xtUML to define behavior. Statechart Diagrams are based upon a subset of Statecharts by Harel [Har92], and are similar to the UML State Machine diagram. Activity Diagrams are not used to define behavior because it is necessary to define the behavior required in each action state ([RFW04], p13). Finally, class collaboration diagrams are used to view how classes interact with each other.

The previous diagrams are used by xtUML to create executable models as follows [Mel07a]. Each class with dynamic behavior has a state machine defined by a statechart that describes the behavior in a number of states. Each state machine can be in one state at a time. The semantics of execution is a set of communicating state machines that are executing concurrently. A state machine can communicate with another state machine by sending a signal, which causes a transition. State transitions trigger a change of state, which triggers an entry action defined in action language.

The methodology associated with xtUML consists of two parts ([RFW04], Chap. 3): System Level Modeling and Domain Modeling. During system level modeling the system is partitioned into domains. Excluding the application domain, domains provide services to other domains such as security, logging and user interfaces. During domain modeling, each of these domains is modeled in detail.

System level modeling ([RFW04], Chap. 3) is performed with use cases to gain an initial understanding of the system. This understanding helps determine the vocabulary of the system and the domains that the system is composed of. The domains are captured in the Domain chart and their interfaces, called bridges in xtUML, are defined. Each of the domains is then modeled, potentially concurrently. It is important to choose carefully which domains to model first, because modeling the domain may make it necessary to revise the domain chart and the bridges between the domains.

Domain modeling ([RFW04], Chap. 3) is an iterative process which is performed for each domain identified during system modeling. Each iteration of the domain proceeds by considering the use cases created during system modeling. The following tasks are then performed:

- Scenarios of the use cases are investigated by adding or refining sequence diagrams
- The class diagram is added or refined
- A statechart is added or refined for classes with dynamic behavior
- A class collaboration diagram is added or refined
- Any operations and state actions are specified in action language

At the end of each iteration, the domain model is executed to test conformance to the use cases it is based upon.

Mellor claims that just as programming languages hide information about the hardware platform by abstraction such as registers, action languages hide information about the software platform such as distribution strategies, list structure, remote procedure calls, and cardinality of data structures [Mel04]. This is refuted by France and Solberg [FS06], “Writing a method body using an action language that does not provide a level of abstraction above primitive actions can require as much effort as writing the method body in some programming language”. Another issue is that xtUML deals with informal knowledge by iteration, this is discussed in detail in Section 2.3.2. Frankel [Fra04a] argues that model compilers have trouble dealing with the multitude of variation present in platforms consisting of a mixture of standards (e.g. J2EE), well-known proprietary systems (e.g. SAP) and lesser-known proprietary systems that may be produced in-house.

## **2.3 A New MDE GPL for System Behavior**

Given the previously reviewed MDE approaches, we propose there is a need for a new MDE GPL for capturing system behavior. Existing meta-modeling approaches have difficulties



capturing the rich semantics of languages which model system behavior. UML is widely regarded as unsuitable to be a GPL for MDE in general, due to its lack of formal usage, bloat of features, and poorly defined extension mechanism. Using a subset of UML with well defined formal semantics, such as Executable UML, is also problematic. Executable UML, with the use of an action language, essentially becomes a graphical programming language which is too low-level to benefit much from abstraction.

Based upon our review of current MDE approaches, this new GPL for system behavior should have three central features. Firstly, it should have an appropriate representation for capturing system behavior. Secondly, this representation should be compatible with a scaleable methodology that encourages formal usage. Finally, the representation and methodology should be accompanied by an extension mechanism that can capture the specific semantics of new domains involving system behavior.

### 2.3.1 Representing System Behavior

The choice of an appropriate representation for a GPL for modeling system behavior should be made based upon the nature of the software being captured. Brooks considered capturing software to be a complex task [Bro87], “The essence of a software entity is a construct of interlocking concepts: data sets, relationships among data items, algorithms, and invocations of functions”. In addition to the essential complexity arising from the inherent nature of software, Brooks also believed that representations could give rise to accidental complexity causing difficulties not inherent to the task of software development. Despite this, Brooks did not consider representation to be a central concern, “I believe the hard part of building software to be the specification, design and testing of this conceptual construct [software], not the labor of representing it and testing the fidelity of the representation”.

Dromey contests this view, giving representation a more central role [Dro06b], “... the choice of representation seriously impacts the complexity and relative difficulty of a task, the ease of understanding and changing what is represented, and the likelihood of making mistakes.”. Alexander also considers diagrams a powerful tool providing similar benefits as MDE through abstraction ([Ale64], Preface),

“The idea of a diagram, or pattern is very simple. It is an abstract pattern of physical relationships which resolves a small system of interacting and conflicting forces and is independent of all other forces, and of all other possible diagrams ... it is possible to create such abstract relationships one at a time, and to create designs which are whole by fusing these relationships ...”

The representation of system behavior has a long history of being defined by visual languages. Since these languages are all very similar, it is important to consider their advantages and disadvantages. Another interesting issue is to determine the impact of the underlying representation of these visual languages.

### Visual Languages

Sowa ([Sow00], pp.217-218) demonstrates the similarity of visual languages designed to capture system behavior using the visual languages of flowcharts, finite state machines, and Petri nets. Figure 2.3 shows the example used by Sowa to compare these three visual languages. Flowcharts [GvN47] represent computational events as boxes and decisions as diamonds. A finite state machine (FSM) [WW03] (a popular

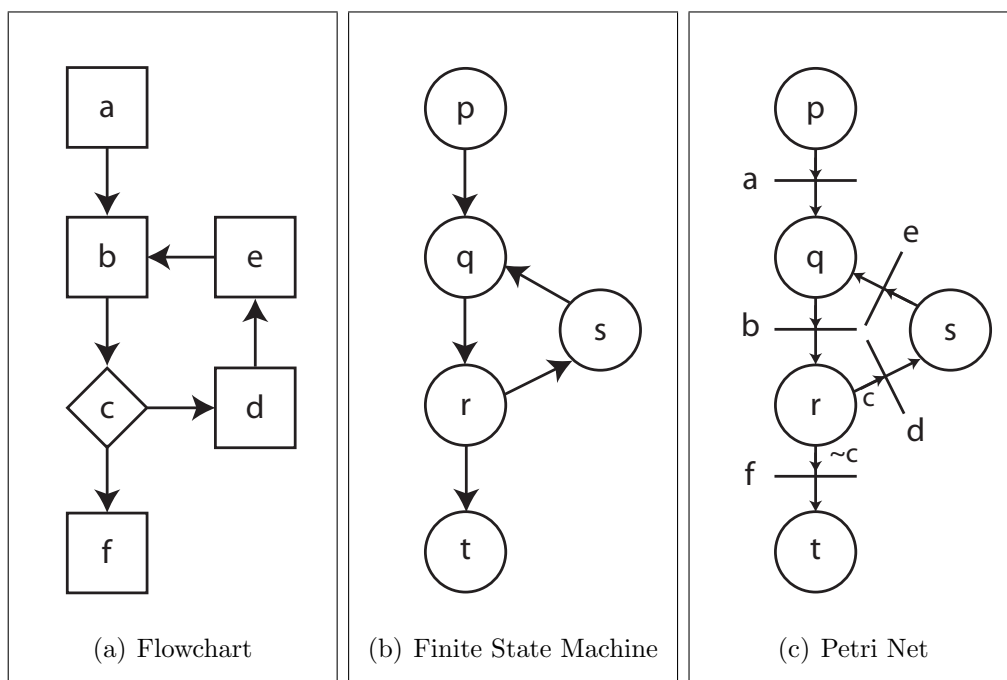


FIGURE 2.3: Three visual languages for capturing system behavior [Sow00]

form of State-transition diagrams) represents states as circles and transitions between the states as directed arrows. Place transition (PT) nets [JKW07] (The simplest form of Petri nets) consist of places and transitions, which are connected together by directed arcs. Figure 2.3 shows how a Petri net (PN) can be constructed that uses the events of a flow chart as the basis for the transitions and the states in a FSM as a basis for the places.

In addition to capturing both events and states in a single diagram, PNs [JKW07] are also particularly suited to modeling system behavior because they can capture concurrency. PNs model concurrency using tokens, which are located inside the places of the PN. The transition of a PN can fire whenever a token is located at the place connected by an input arc. When a transition fires, the token is moved in an atomic action to the place at the output arc. PNs have non-deterministic execution semantics; when multiple transitions are enabled it is possible for any transition to fire, or for none of the transitions to fire.

The task of defining PNs can be simplified by using extensions to create a high-level PN. High-level PNs include Colored PNs [JKW07], Hierarchical PNs [JKW07], Timed PNs [Wan98] and Stochastic PNs [Haa02]. In Colored PNs, colors can represent a type of state for each place, a type of event for each transition, and a type of object associated with tokens. Hierarchical PNs organise small Petri nets into hierarchical models which can then be combined to form a larger Petri net. Timed PNs and Stochastic PNs provide additional information regarding the behavior of transitions by adding timing and randomness to create non-deterministic behavior respectively.

PNs also have similarities to other visual languages. Activities are the main concept of activity diagrams in UML2 and use a Petri-like semantics [Sto04]. PNs are also similar to Behavior Trees (BTs), the visual language used in Behavior Engineering to capture system behavior. BTs also can model both states and events in a single diagram and can capture concurrency. Beyond these similarities, however, there remain some subtle but important differences that distinguish BTs from PNs<sup>6</sup>.

The first of these differences is that PNs are directed biparte graphs which allow cycles. The presence of cycles in a PN can create a network structure with several intersecting lines.

---

<sup>6</sup>In the following discussion we will use notation based on PNs to simplify the comparison between PNs and BTs. The BT language is discussed in detail in Chapter 3.

This network structure has the potential to obscure information in the PN, particularly as the number of places and transitions in the PN increases.

BTs remove the cycles present in a PN by using a reversion operator that forms an implicit link with an earlier part of the tree. This allows BTs to use a tree-like structure and avoid the potential for obfuscation of information present in a network structure. We use the term tree-like because while a BT has the structure of a tree, it implicitly remains a graph as the reversion operator forms semantic links that break the tree structure. Figure 2.4 shows how the network structure of a PN with a cycle can be represented in a tree-like structure with a reversion operator.

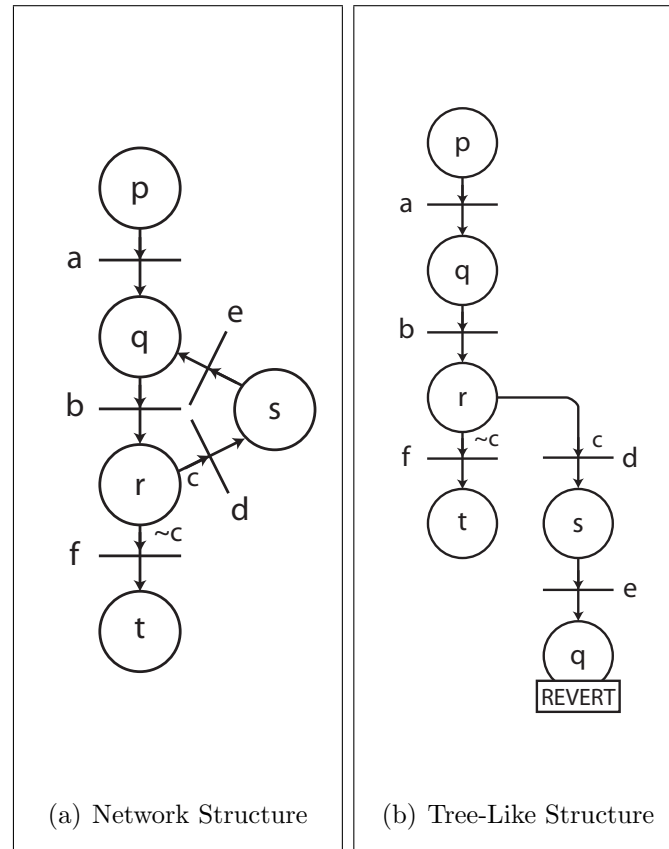


FIGURE 2.4: Network Structure versus Tree-Like Structure

Another difference between a PN and a BT is how behavior is modularised. An example of Hierarchical PNs is shown in Figure 2.5(a). The Hierarchical PN in the figure has been simplified for this example, since it would actually consist of ports which are used to connect the modules, rather than keeping the transitions. Even from this small example, it is evident

that while Hierarchical PNs simplify the construction of a single module, the complexity is shifted to the interactions of these modules. BTs offer a different approach to modularisation in which places are tagged with the module they belong to, as is shown in Figure 2.5(b). This has the benefit of allowing a tree structure to be maintained, which is not possible if similar behaviors are graphically segmented into individual modules.

A third difference between a PN and a BT involves how transitions are defined. Transitions are described in PNs using a separate modeling language such as the coloured Petri net modeling language (CPNML) ([JKW07],p5). BTs differ from this by formally defining transitions graphically using a small subset of formalised transition types which are tagged to each place. Transitions with expressions are now defined in places, so other transitions no longer need to be labeled and are used solely to define the flow of control. Figure 2.5(c) shows how a Petri net can define transitions graphically using a *state* and *if* transition type.

Simon offered the following advice with regard to representation ([Sim79], p472), “To encode something is one matter, to encode it in such a way that it is useable is another.”. The three differences we have discussed between a BT and PN appear to make BTs more useable by improving the speed at which the diagram can be comprehended and increasing the information available in the diagram itself. Though these benefits have not been proven, research by Simon into the nature of human visual memory appears to show a structure particularly suited to parsing trees. Simon ([Sim88], pp 84-89) found that human visual memory is captured as relations between adjacent elements which are stored as hierarchical list structures consisting of descriptive elements and lists of three to four elements. This means of capturing and storing visual memory matches well with a tree structure which keeps related elements spatially close and organised in a fashion easily compatible with a hierarchical list structure. Tree structures may also benefit from being quickly parsed. Search trees are commonly used in search strategies ([Sim88], pp 66-72), which in addition to their computational benefits, also provide benefits to comprehension such as allowing related children to be quickly identified and compared.

According to Petre [Pet95] some of the differences we introduced between PNs and BTs are known as secondary notation. Based on several surveys, Petre found that many of

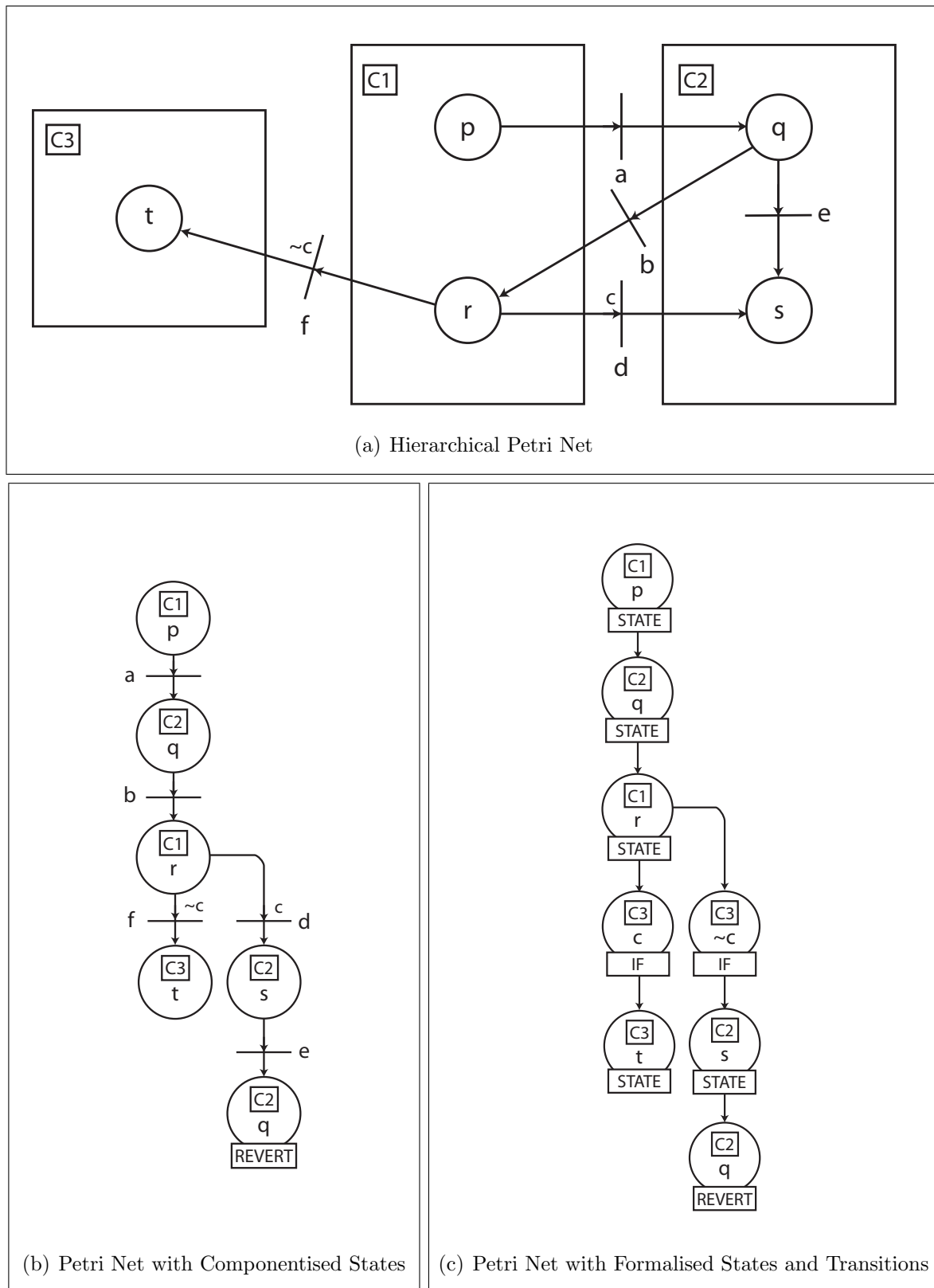


FIGURE 2.5: Segmentation and Transitions in a Petri Net

the benefits of graphical representations did not come from the notation itself, but from a secondary notation involving layout, cues, and graphical enhancements. He found that this secondary notation was an acquired skill, which experts could both represent and comprehend better. The effect of secondary notation may explain some of the reasons programmers give for preferring graphical notations including [Pet95]: they are richer (more information in less space); they can provide a ‘Gestalt’ effect giving a complete overview; they operate at a higher-level of abstraction, and; present information in a way that is quicker and easier to comprehend. By using a tree representation in BTs, secondary notation is restricted allowing experienced modelers to quickly identify concurrency and flow of control.

Appendix A illustrates these points using three visual languages. In the next section, we discuss the underlying object-oriented representation used by most of these visual languages and present an alternative behavior-based representation.

### **Underlying Representation**

The history of software development consists of continually raising the level of abstraction to manage complexity and enable reuse. Mellor et al. ([MKUW04], Chap. 1) describes the path of increasing abstraction beginning with machine code sequences of ones and zeros, to assembly languages, then early programming languages such as FORTRAN (formula translation) and COBOL, to the current programming languages such as Java and C++. This rise in abstraction is also attributable to reuse. Reuse was common in earlier systems to preserve memory, with assembly code being reused wherever possible and flags used to distinguish small differences. As flags became increasingly difficult to maintain, functions were developed which operated on shared data structures. These shared data structures also caused maintenance problems, leading to objects with local data and methods. Finally, components were created to allow the reuse of a group of closely related objects.

Most software development approaches currently remain primarily object-oriented. Object orientation (OO) was developed initially in the 1960’s, yet complete consensus about the fundamental concepts of OO still does not exist. One attempt at reaching this consensus was performed by Armstrong [Arm06], who conducted a literature review of OO development publications from 1966-2005 and found eight central concepts: Inheritance

(81%), Object (78%), Class (62%), Encapsulation (63%), Method (57%), Message Passing (56%), Polymorphism (53%), and Abstraction (51%). The survey identified 31 other concepts which Armstrong found were similar to or identify with the eight central concepts.

There is a concern amongst some researchers that OO's fundamental concepts are not suited to MDE. Some researchers also contend that objects do not scale due to collaboration problems ([Tri06], p3),

“Since by definition objects are simple to design and understand, complexity in an object-oriented system is well known to be in the *collaboration* between objects, and large systems cannot be understood at the level of classes and objects.”

This is significant because [Fra04b], “Most software architects and developers view Model Driven Architecture (MDA) as fundamentally based on object-oriented technology (OO) because of its connection with the UML”. Frankel [Fra04b] describes the problem with using OO concepts in MDE as object technology barriers, stemming from issues with the tight coupling of data and behavior in objects. Frankel identified several object-technology barriers including impedance mismatch between business modeling and software modeling and frequent process change. The impedance mismatch between business and software modeling is due to the modeling of different concerns. Business modeling is interested in information which is acted on by processes, which is hard to capture in terms of the data and behavior framework of OO. Frankel suggests components may be more suited to capture this relation between information and process. Another object-technology barrier is frequent process change. In business processes, it is more common for processes to change than the information structure they operate on. In OO, a change in process has a big impact on the software because the data and behavior are tightly bound together in objects.

Behavior based (BB) approaches offer an alternative to OO representations. BB approaches operate on the principle of representing system behavior by separating individual behaviors from their means of integration. This principle has been applied in disciplines such as robotics, artificial intelligence and component-based software engineering. Table 2.1 provides a summary of some of these BB approaches.

One of the earliest examples of a BB approach is W. Grey Walter's *Machina Speculartrix*



Field	Approach	Ref
Robotics	Machina Speculartrix	[Wal61]
	Subsumption Architecture	[Bro87]
Artificial Intelligence	Behavior Directed Acyclical Graphs	[DHJ <sup>+</sup> 01][Isl05]
Component-Based Software Engineering	VICON	[Par00]
	REO	[Arb04]
	Exogenous Connectors	[LEW05]
	Behavior Engineering	[Dro01]

TABLE 2.1: Summary of Behavior-Based Approaches

([Wal61], pp. 113-118), in which tortoise-like robots named after their speculative desire to explore their environment. During the time when the Machina Speculartrix was created, most robots utilized fixed, preprogrammed patterns of behavior. These robots could not operate in a new environment without being totally redesigned and were not robust enough to continue operating even after a minor change in the environment. Walter used a new approach where individual behaviors were activated by the current environmental conditions. This meant Walter's robots could operate in multiple scenarios, requiring only small constraints to be placed on the layout of the environment.

The Machina Speculartrix have the mechanical structure of a front-wheel drive tricycle, equipped with: a light sensor; a touch sensor; motors for propulsion and steering, and; an analogue computer comprised of two vacuum tubes. The robot exhibited one of three behaviors, determined by the state of the environment as perceived by the light sensor and the touch sensor:

- **Search:** Steering motor is on full speed and the propulsion motor is on half speed
- **Move:** Steering motor is off and and propulsion motor is full speed
- **Dazzle:** Steering motor is on half speed and propulsion motor is on full speed

If the light sensor determines the environment is dark, search is active. A moderate light reading activates move and a bright light reading activates the dazzle behavior. If the touch sensor makes a detection, then the active behavior toggles between search and dazzle. The integration of these individual behaviors introduced emergent system-level behaviors such as seeking out light, avoiding obstacles, alternating between light sources and interaction

between two robots. Walter's robot introduced a novel architecture in which the behavior that is selected is determined by current environmental conditions. This can be considered an early BB approach because the means of integrating the individual behaviors of the robot is separate from the implementation of the behaviors themselves.

A more commonly recognised starting point for BB approaches occurred when robotics researchers began investigating real world examples of intelligence taken from ethology (study of animal behavior), entomology (study of insect behavior), neurobiology, psychophysics (study of the relation between stimulus and sensation) and sociology. In 1986, Brooks [Bro86] proposed a new method of decomposing a mobile robot control system by using task-achieving behaviors rather than traditional decomposition into functional modules, which became known as the behavior-based approach. When proposing the behavior-based approach, Brooks also described a method for combining these behaviors called the Subsumption Architecture. The Subsumption Architecture consists of a number of layers of behavior that are directly coupled between sensors and actuators. Fixed priority arbitration is used to resolve competition between different layers, allowing a higher-level behavior to subsume a lower-level behavior and inhibit its output to actuators. Brooks created a number of robots with the Subsumption Architecture which autonomously achieved goals such as [Bro90]: visiting interesting locations; following moving objects; walking a six-legged robot over rough terrain; retrieving soda cans from offices, and; opening doors in a corridor. The Subsumption Architecture provides an approach to integrating individual task oriented behaviors that is adaptable enough to create systems suited to a number of different goals. The simple integration mechanism of the subsumption architecture, however, has trouble coping with co-ordinating large numbers of behaviors, particularly if they are co-operative or operating independently in parallel.

In the field of artificial intelligence, behavior directed acyclical graphs (behavior DAGs) were developed as a means of describing the behavior of AI actors, replacing customised state machines which were previously used for the same purpose. They were first used by the NOAI team [DHJ<sup>+</sup>01] to develop behaviors for robotic agents playing in the simulated league of robocup. The approach has since been improved and used in various commercial computer games such as Halo 2 [Isl05] and Spore [Hec09]. Behavior DAGs consist of nodes

which describe behaviors and terminals which describe skills. Each behavior is defined by a group of lower-level behaviors (or skills) and a relevancy value. The relevancy value indicates the suitability of the behavior to the agent's current environment and state. One approach of determining relevancy is to use predicate expressions, resulting in a fuzzy boolean value ranging between 0 and 1. Arbitration then involves selecting the behavior with the highest fuzzy boolean value.

For larger systems, using a binary boolean value (true or false) for relevancy of a behavior is necessary for scalability [Isl05]. More complex arbitration strategies can then be used for determining which behavior is activated including prioritisation (similar to the Subsumption Architecture), sequential lists, sequentially looping lists, probabilistic choice, and one-off lists. Time spent determining relevancy can be reduced by extracting basic relevancy conditions from the relevancy expressions and tagging them to the behaviors. When a tag is not relevant, large portions of the tree are skipped for relevancy checking. When the tag is relevant, the relevancy expression of the behavior is evaluated.

Other behavior DAG issues include behavior impulses, stimulus behaviors and memory requirements. Behavior impulses contain a relevancy expression that references another behavior in the tree. If the behavior impulse is executed, the referenced behavior is run in place of the behavior impulse. Stimulus behaviors dynamically add behavior impulses to the tree when an event occurs. This allows event-driven behavior to be dealt with economically, ensuring behavior impulses based on events do not have to be continually evaluated. Memory needs for Behavior DAGs include per-behavior (short-term and persistent), per-object and per-object per-behavior. Short-term per-behavior memory is only required while the behavior is executing, persistent per-behavior memory is required to be stored after execution as well. Per-object and per-object per-behavior memory store the agent's state information regarding objects it has interacted with.

Similarly to BB approaches in robotics and AI, BB approaches are also used with components in software engineering. BB component based software engineering (CBSE) approaches include visually integrated component (VICON) [Par00], REO [Arb04], exogenous connector (EC) [LEW05] and Behavior Engineering [Dro02]. In CBSE, the mainstream view is that a component has several ports with provided and required services which are linked to other

components [LW07]. This differs from the behavior-based approach to CBSE proposed by Parr ([Par00], p. ii), "... systems integration (dataflow and control flow between components) should be separated from, and implemented differently to, the computational concerns that are encapsulated and implemented in components".

Parr [Par00] developed the VICON system using black-box components; self-contained, self-sufficient applications that can run in isolation. VICON components, however, do not communicate directly by invoking each other's methods, but communicate through an integration server. The integration server controls all the data and control flow between the components. VICON component inputs and outputs are linked by connectors. There are five output connector types which determine how to handle output data: no memory buffering, shared memory buffer, shared priority buffer, unique memory buffers, and unique priority buffers. There are six input connector types which deal with component triggering: auto-start, standard, mutually exclusive, data triggering, data termination triggering, and optional signals.

The REO framework [Arb04] was developed by Arbab based on his earlier concept of exogenous coordination [Arb98]. REO uses channels as the primitives for connectors to 'glue' together components. REO channels consist of a source and a sink end. The source end accepts data into the channel and the sink end dispenses data out of the channel. Channels support data transfer using input/output operations, but do not support message passing with method call semantics. Components can control channels they are connected to using operations such as create, forget, move, connect, disconnect, wait, read, take and write. REO Connectors are formed by composing together channels, which can be either synchronous or asynchronous. Example synchronous connectors include sync, filter, syncdrain, syncspout and lossysync. Example asynchronous connectors include FIFO, FIFO<sub>n</sub>, asyncdrain, asyncspout, shiftFIFO<sub>n</sub> and lossyFIFO<sub>n</sub>.

Lau et al.'s ECs [LEW05] encapsulate and manage all control and data flow between components. Lau distinguishes EC from REO because control flow is also handled [LLUE07], "In Reo, components only perform I/O operations, and connectors are data channels.". Similarly to VICON, components of exogenous connectors do not invoke methods on other components. In the EC model, however, this is done by the connectors themselves rather

than an integration server. EC are also composable, as in REO, and consist of multiple connector levels. EC consists of one level 1 connector, called the method invocation connector. The method invocation connector can connect to a single component, run any method, and provides a single result. EC level 2 connectors connect to several invocation connectors. Examples include a Pipe connector which passes values from the execution of one component to the invocation of the next component and a Selector component which selects one connector to execute. EC also consists of level  $m > 2$  connectors which connect to connectors one level lower than  $m$ . These connectors are typically used to capture design patterns. An example composite connector is an Observer, formed by linking a level-2 Pipe and a level-1 Sequencer [LLUE07].

The dynamic view of Behavior Engineering's Behavior Modeling Language, Behavior Trees (BT), is also a Behavior-Based CBSE language. Behavior Engineering is based upon the principle that [Dro06b], "Systems are built out of a network of interacting components". BTs are a formally defined multi-threaded graphical language that describe component interactions that coordinate the activation of individual behaviors encapsulated within the components. Rather than group behaviors that are associated with the same component, several instances of components and their associated behavior are interleaved throughout the tree.

BTs also manage both data and control flow, similarly to ECs. The primary difference between BTs and ECs, is that BTs as part of BE comprise a means to rigorously formalise informal knowledge. Another difference is that ECs generally group behaviors that are associated with a component, and define them in one place in the diagram. A result of this is that when modeling with ECs, it is hard to determine the most suitable hierarchy of components, similar to problems in OO with defining a type hierarchy of classes. This problem is avoided in BTs because components and their associated behaviors are interleaved and the integration of behavior naturally emerges from the translation of the requirements. These issues are discussed further in Appendix A, where a small case study is used to contrast Exogenous Connectors with Behavior Trees.

The benefits of behavior based approaches over OO arise primarily from OO encapsulating interactions inside components together with data and computation. BB approaches

separate the task of integration from the definition of computation. This separation of concerns allows interactions to be modeled first. A by-product of first focusing on interaction with a BB approach is that the overall system behavior is naturally decomposed into several smaller units of self-contained behavior. This has benefits for development, testing, re-use and maintenance of components. In the next section we will discuss how a methodology using a bottom-up process such as this is necessary for a new GPL for MDE.

### 2.3.2 Using a Scaleable Methodology

For a methodology to be scaleable it must be closely linked to an associated representation and it must encourage formal usage. These two characteristics are hard to achieve because it is common in many fields for designers to resist formalised processes to protect their creativity. Even Brooks saw software construction as a creative process that requires great designers so he was not confident of the role of methodology in helping less talented designers [Bro87], “Sound methodology can empower and liberate the mind; it cannot inflame or inspire the drudge”.

Other researchers see formal design methods as a central issue. Alexander [Ale64] argues for the need for formal design methods in architecture, but the principles apply equally well to software development. Alexander justifies using formal methods by means of a parable ([Ale64], Epilogue) about an imaginary civilisation with no formalised arithmetic. When deciding how many seeds are required to sow a crop, citizens determine the area of the field (e.g. 6m by 10m) by pacing both sides. They then mull over the two numbers and come up with a result; maybe 60, or 61, or 58. They found this process accurate enough for the task, and found formal arithmetic unnecessary. Men who were particularly good at producing appropriate answers were honored as ‘seers’. In failing to adopt a formalised process, Alexander argues that the civilisation was neglecting the numerous advances that formal arithmetic could provide them.

The principles of this parable are identifiable in software development in two areas: not linking methodology and representation closely and modeling languages not being supported by a methodology that encourages formal use. We will now discuss each of these areas in

turn, using UML as an example.

### **Linking Representation and Methodology**

We propose that for a representation to be successful, it must have a closely associated methodology. Representations are built on (sometimes implicit) assumptions as to how they will be used. These assumptions suit a particular methodology or set of methodologies. Creating good software requires three aspects: a good representation; a good methodology; and that both the representation and methodology are compatible with each other. The easiest way to make a representation and methodology compatible is to design them together, thereby ensuring that they interact well.

Closely linking representation with methodology is at odds with current software development representations such as UML. A foundational goal of UML is to provide visual notation for specifications which can then be used by any methodology or process. The OMG's website states [Obj09b], "OMG, as a vendor-neutral organization, does not have an opinion about any methodology. To help you get started selecting the one that's best for you, we've collected links to methodology resources here.". One of these links lists over 40 different methodologies which are almost exclusively object-oriented. Many, such as the Rational Unified Process, also assume a prior understanding of object-oriented development techniques. Because UML is created from the unification of three object-oriented notations, it is understandable that UML is more suited to object-oriented methodologies; this, however, is not congruent with the principle that a representation can be used by any methodology.

One reason for the wariness about linking methodology and representation may be traced back to the failure of CASE. Recall that one of the reasons for the failure of CASE was linking proprietary methodologies to a representation. Another reason, as witnessed by Alexander in architecture, may be due to the desire to protect creativity and resist formal processes.

As an alternative to UML's principle of a representation which can be used by any methodology, we propose it is the responsibility of a methodology to support a representation towards the goal of making good software. Meyer [Mey97] addressed how UML contributes to making good software in a satirical essay from the point of view of a student requesting a grade change. The essay states that good software should be correct, robust, easy to change,

reusable and efficient. Meyer’s fictitious student suggests that: UML does not help make software that is correct or robust; making software that is easy to change is hindered by UML bloat; UML does not address reusability, and; UML is not efficient because bubbles and arrows never crash. Whilst UML has improved dramatically since this essay was written, many of these issues still remain.

Having a representation that is not linked to a particular methodology also causes identifiable problems with UML. Bell cataloged some of these problems into a group of UML fevers with the caveat that, “UML itself is not the direct cause of any maladies described herein. Instead, UML is largely an innocent victim caught in the midst of poor process, no process, or sheer incompetence of its users” [Bel04]. Bell [Bel04] documented 18 fevers in total arranged into four metafevers: Delusional, Emotional, Pollyanna, and Procedural. Several of Bell’s procedural UML fevers can be attributed to a lack of methodology linked to representation. Circled wagon fever involves using UML use cases to perform a fine grained functional decomposition of the domain in an attempt to perform object-oriented domain analysis. Gnat’s eyebrow fever involves creating UML diagrams that are extremely detailed to increase the probability that the code that is based on them will be correct. Kitchen sink fever involves building massive UML models that are very detailed [Bel04], “Victims of kitchen sink fever typically spend significant amounts of time recovering from the effects of crashes of their modeling tools.”. Round-trip fever involves creating a UML implementation model reverse engineered from code, rather than creating a UML design model first.

### **Encouraging formal usage**

In addition to linking a methodology to a representation, the associated methodology must also encourage formal use of the representation. This is needed, because representations must have formal semantics to gain the benefits of MDE. Another evidence of the disadvantage of UML not being associated with a particular process is a survey by Lange of UML usage in industry. Over 60% of participants in the survey self-assessed themselves as informal users of the UML specification with loose, very loose or no adherence [LCM06]. The survey also found that criteria for stopping UML modeling and beginning development of the software included [LCM06]: passing a review (33.8%), completeness (52.5%), amount of effort (17%)



and deadline (32.8%). Using completeness as a stopping criteria is hard because a definition of completeness for UML models does not exist. Lange also notes that using a deadline as a stopping criteria is a strong indicator of a bad quality product.

Scaleable methodologies can support formal use by helping modelers overcome the informal/formal barrier. The informal/formal barrier occurs due to the semantic gap between informal representations such as natural language with imperfect knowledge and formal representations suited to software development. To overcome the informal/formal barrier, imperfect knowledge consisting of inaccurate, inadequate, unnecessary, incomplete, inconsistent and redundant information must be systematically dealt with [Dro06a]. This information must then also be analysed to determine how the system will deliver what is specified. According to Dromey, dealing with the informal/formal barrier is crucial [Dro06c],

“It does not matter how good or how formal the representations are that we use for design/modeling, unless the first step that crosses the informal-formal barrier is as rigorous, intent-preserving, and as close to repeatable as possible, all subsequent steps will be undermined because they are not built on a firm foundation.”

The most common approach to deal with the informal/formal barrier is to use quick, successive cycles of verification until the developed product meets the initial requirements. If this approach is used, then success depends on how quickly the current design can be verified, essentially replacing the informal/formal barrier with a verification barrier. Mellor links the success of a modeling language in overcoming the verification barrier to the level of formal usage. He categorises three levels of formal usage of modeling languages [Mel07a]: model as a sketch, model as a blueprint, and model as a programming language.

Using a model as a sketch is common usage of UML, where UML models are used to discuss abstractions before commencing coding. The model produced using this process forms documentation of the agreement reached about what is going to be built. Modeling as a sketch causes a significant gap of time before the documentation can be verified against the actual software, leading to the possibility of building the wrong system for several months.

Using a model as a blueprint, is analogous to how a blueprint is used to make a building

except that a UML model is used to as a blueprint for software architecture. This approach uses elaboration to automatically generate a code framework, but relies on developers to manually develop the remaining code.

A model as a programming language uses UML as programming language (e.g. Executable UML) with well-defined executable semantics. By defining all the behavior in the model using an action language, translation can be used to achieve full code generation from the model. Mellor [Mel07a] argues this closes the verification gap by allowing models to be quickly verified as they are being developed, and enables platform-independent modeling.

Mellor's model as a programming language does provide a means to overcome the informal/formal barrier by iteration, translation and verification. Despite removing the verification barrier, however, resistance to using a model as a programming language still exists. As discussed in Section 2.2.3, this approach relies on an OO representation where interactions must be specified at the lowest-level in an action language. This makes it hard to capture subject area modeling dimensions, which may not map well directly onto a low-level OO representation.

Use of models as a sketch and models as a blueprint may be due to the failure of methodologies to support bridging of the informal/formal barrier. Three gaps exist in crossing the informal/formal barrier: moving from requirements to a specification, from specification to a design, and from design to code. The requirements-specification gap involves resolving problems with informal knowledge represented in natural language, such as ambiguous, incomplete, inaccurate and conflicting requirements. The specification-design gap involves transitioning from a model describing *what* the system must do to *how* the system will do it. The third gap, design-to-code, is the most commonly addressed, using approaches such as Mellor's model as a programming language [Mel07a].

Schmidt [Sch06] classifies approaches to overcoming the design-to-code gap as either correct-by-construction or construct-by-correction. Correct-by-construction approaches use automated transformation processes to translate a design into code (or one formal model into another formal model). Alternatively, construct-by-correction approaches use handcrafted conventional software design processes. The automation used by correct-by-construction approaches provides many of the productivity benefits of MDE. It also minimises or

eliminates the introduction of defects when translating a design into code.

Unfortunately, whilst the design-to-code gap can use automated transformation processes, the creativity required for the specification-design and requirements-specification gaps due to the abundant interpretations and choices in these processes precludes the possibility of automated transformations. Consequently, we suggest redefining the correct-by-construction approach for these two gaps to a scaleable methodology that guides the representation to ensure a higher likelihood of a correct result. Similarly, a construct-by-correction approach to these two gaps has shortcomings since it uses either a poor methodology to deal with these gaps or it does not deal with them at all. A common result of applying construct-by-correction approaches to the specification-design and requirements-specification is a large degree of iteration. On the whole, a correct-by-construction approach deals with the requirements-specification and specification-design gaps better, resulting in less iteration.

Based upon this redefinition, xtUML is successful at bridging the design-code gap but ignores the rest of the informal/formal barrier. xtUML closes the verification gap, allowing quick iterations which are performed until the right system is delivered. Some of this iteration is unnecessary, because Use Cases are not suited to solving problems necessary to bridge the Requirements-Specification gap. Another cause of unnecessary iteration is that xtUML is too low-level to describe a design-independent specification. This results in essentially omitting the specification stage altogether and proceeding to a design which is developed using a top-down iterative process for defining class diagrams called ‘object blitzing’ ([RFW04], p154).

### 2.3.3 Extension Mechanism

Developing a scaleable methodology is a complex task because the system behavior modeling dimension contains semantically rich concepts. Ideally, a representation that uses a scaleable methodology should be widely applicable so as to gain as much benefit as possible. This necessitates any extension mechanism for creating DSLs that represent system behavior to support mapping semantics in addition to abstract syntax. Mapping semantics using an extension mechanism, however, is not currently supported by any of the existing approaches to MDE that we reviewed in Section 2.2.

Meta-modeling approaches generally use language invention to create a completely new language, which has problems for the rich execution semantics required to capture system behavior ([CSW08], p24),

“The task of creating a metamodel for a language is not a trivial one. It will closely match the complexity of the language being defined, so for example, a language containing rich executable capabilities will be much more complex to define than a simple static language.”

Barbero et al. [BJGB07] discuss a different approach to metamodeling extension, where new models are derived from base models. This extension approach extends an existing base metamodel using an extension metamodel which defines new concepts with reference to concepts in the existing metamodel. These two metamodels are then composed using automated model transformations, with a resulting metamodel containing both the concepts of the base metamodel and the extension metamodel. Barbero et al. demonstrate this approach using a simple Petri net metamodel which is extended to include token markings. A limitation of this approach, however, is that it only extends the abstract syntax [BJGB07],

“From a model-based point of view, the abstract syntax of a DSL is a metamodel. Thus, DSL concepts can be extended with the mechanism described in this paper. The study of how to extend the concrete syntax model and how it is related to grammar extension represents areas of future work.”

UML uses language exploitation to create a family of languages using UML Profiles. Though the situation has improved somewhat in recent revisions of the UML specification, Selic states that a lack of information on how to use UML profiles has caused [Sel07],

“... very many UML profiles that are either technically invalid because they contravene standard UML in some way (and, thus, cannot be properly supported by standard UML tools ) or they are of poor quality.”

While Selic does present a method to create UML profiles, it is reliant on ensuring concepts agree semantically. The reason for this limitation is that according to Selic, “At present,

there is no standard associated with UML that would allow a formal specification of the semantics of a UML profile.”. This situation may improve in the future with the introduction of the fUML standard, which will allow UML semantics to be defined using a small formally specified subset of the UML. Until this occurs, however, many researchers consider the use of UML profiles as an extension mechanism to be limited [FS06, HT06, Sel07].

xtUML has executable semantics, but it does not have an extension mechanism. xtUML domains focus on the system aspect modeling dimensions, and provide little support for different subject area modeling dimensions. Thomas questions whether xtUML is sufficient for all MDE applications [Tho04], “Many in the executable-UML crowd apparently work in domains that only require state machines, so they might have a vested commercial interest in UML remaining less than a complete language”.

A more relevant example of an extension mechanism for defining DSLs based on system behavior is Petri nets. Petri nets have recently begun a standardisation process to become an international standard (ISO/IEC 15909). The standard consists of three parts [HKPDT06]: Part I defines terms and definitions for place/transition nets and high-level Petri nets; Part II defines a transfer format for high-level Petri nets based on XML, and; Part III will define Petri net extensions and an extension mechanism, but this is yet to be started.

The design of the petri net markup language (PNML) on which the standard is based is governed by three principles [BCvH<sup>+</sup>03]: Flexibility, Unambiguity, and Compatibility. Flexibility ensures any type of Petri net with specific extensions and features can be represented. Unambiguity ensures that the original Petri net and type can be determined from its PNML representation. Compatibility ensures that any compatible information can be exchanged between Petri nets of different types.

The standard currently defines a core metamodel, a place transition nets metamodel and a high level Petri nets metamodel [HKPDT06]. The metamodels are built on extensions similar to Barbero et al.’s extension mechanism. That is, the high level Petri nets metamodel extends the place transition metamodel, which extends the core metamodel. This means that only extensions of the abstract syntax can be captured, not semantic mappings of high-level Petri net concepts onto place transition nets. Hillah et al. propose a solution to this [HKPDT06],

“We are also planning to develop an advanced version of PNML Framework, which will generate specific APIs for local variations on Petri net types, not specified by the standard ... The use of this advanced version requires deep knowledge of metamodeling and code generation techniques using EMF’s powerful features.”

An alternative extension mechanism, which may be used in the future for Part III of the Petri net standard, is to use a conventions document. The conventions document describes extensions to the syntax of Petri net labels which are affixed to Petri net places, transitions and arcs. PTCapacity, is an example label, which describes the number of tokens than can be present at one time in a place [BCvH<sup>+</sup>03].

Based on these examples, we propose the following foundations for an extension mechanism for a GPL for capturing system behavior:

1. The GPL should consist of a small core notation to avoid problems such as UML bloat, thereby making the language easy to understand and to extend.
2. The extension mechanism should support making compatible extensions, if possible, by mapping extension concepts to the core notation. This ensures that the extension concepts also have a clear semantics, defined in terms of the semantics of existing concepts of the GPL.
3. If the complexity of the extension concepts makes this prohibitive, domain-specific components should be provided to define the behavior.

## 2.4 Conclusion

This chapter presented an overview of Model-Driven Engineering with a focus on current approaches to modeling software-intensive systems. The review of current approaches focused on three realisations of the Model-Driven Architecture: Metamodeling, UML as a PIM, and xtUML. These approaches were evaluated based on their ability to model system behavior.

The review of current MDE approaches found deficiencies with the current MDE approaches used to model system behavior. Metamodeling approaches focus on capturing abstract syntax, which is suitable for static languages which focus on syntax but does not suit the dynamic semantically-rich languages that capture system behavior. Using the UML as a PIM has numerous issues that need to be overcome, such as: imprecise and incomplete semantics, a lack of formal use, an overly complex and large language, and an easily misused and underspecified extension mechanism. Some of these issues are addressed by xtUML by using a subset of UML with executable semantics and an action language. A consequence of this is that xtUML models at a lower level of abstraction and crosses the informal-formal barrier using iteration and verification.

Because of the deficiencies of current approaches to MDE at capturing system behavior, we propose a new GPL be created. The design of this new GPL consists of three fundamental principles: an appropriate representation for capturing system behavior, a scaleable methodology and suitable extension mechanism.

We investigated visual languages because they are a common representation used to model system behavior, and these languages share many similar characteristics. We found that some accidental complexity may be being introduced because many of these languages use OO as their underlying representation. OO simplifies representations by encapsulating data and methods in an object but a consequence of this is an increase in the complexity of defining the interactions of objects. Because object interaction (or component integration) is central to describing system behavior, we propose the new GPL use a BB CBSE representation.

Another issue this new GPL should address is the need for a scaleable methodology. For a methodology to be scaleable it must be closely linked to a representation. This is contrary to approaches such as UML, the aim of which is to be able to be used with any methodology. We propose that closely linking a methodology to a representation helps to deal with the informal/formal barrier and encourages formal usage.

In reviewing current MDE approaches, we also found that most extension mechanisms focused on defining extensions based on existing syntax rather than existing semantics. Defining the semantics of new concepts for capturing domain-specific system behavior is

central to defining a DSL based on system behavior. We base our proposed extension mechanism on Petri nets, which are currently being standardised. By defining DSL concepts using the semantics of existing GPL concepts, we hope to create a compatible family of languages.

In the next section, we introduce Behavior Engineering which we use as the basis for a new GPL for MDE.



# 3

## Introduction to Behavior Engineering

Behavior Engineering (BE) is founded on the principles of Genetic Software Engineering [Dro01] which were first presented by Dromey in 2001. BE [Dro06c] has since evolved into an integrated approach to systems development that supports the engineering of large-scale dependable software intensive systems at both the systems and software engineering level. The key strength of BE is its ability to model large-scale systems that are common in industry. These systems are initially defined by a large set of interdependent requirements represented in natural language which contain imperfect knowledge. BE manages the complexity of representing these systems by building a system out of its requirements with the use of a scaleable and repeatable methodology which addresses the imperfect knowledge.

This scaleable and repeatable methodology has been developed as the result of extensive industry trials that have been conducted since 2002. These trials involved the requirements

analysis of real-world large-scale systems, most with over a thousand requirements. BE detected defects during these trials at a rate approximately two to three times higher than conventional requirements analysis techniques [Pow07]. One of the companies recently involved in these ongoing trials, Raytheon Australia, reported the following results [Bos08],

“I see it [BE] as a key risk mitigation strategy, of use in both solution development and as a means of advising the customer on problems with the acquisition documentation. Our team now has a strong understanding of the FPS [Functional Performance Specification], at the cost of a relatively small investment. Whilst similar results could have been achieved using conventional processes, the cost and time involved would have been far greater. I will certainly use the process again on future programs.”

The scaleable and repeatable methodology of BE is a result of the tight interlinking of the behavior modeling language (BML) and the behavior modeling process (BMP). The BML is the representation used by BE and is composed of three integrated views where information is represented in a behavior tree (BT), a composition tree (CT) and a structure tree (ST)<sup>1</sup>. The BMP is the process used to create BE models and is made of the four distinct stages of formalisation, a fitness-for-purpose test, specification and design. Figure 3.1 illustrates how each of the four stages of the BMP results in a deliverable that is represented by the BML.

A key to the integration of the BML and the BMP is that each of the stages of the BMP utilises the BML differently to ensure a scaleable and repeatable methodology. Different stages also may use different parts of the BML. Despite the BML being utilised differently, however, the representation that forms the BML remains the same at all stages. This enables the same language to be used for all stages of system development.

The rest of this chapter describes the BML and BMP in further detail to foster an understanding of how BE provides a scaleable and repeatable methodology for requirements analysis and specification. The next section provides an overview of the BML integrated views of Behavior Trees and Composition Trees. This is followed by an overview of the BMP, excluding the design stage which is the focus of the subsequent chapter. The use of

---

<sup>1</sup>Structure trees are not discussed in this dissertation because they are still in the early stages of being formalised and integrated into the BMP.

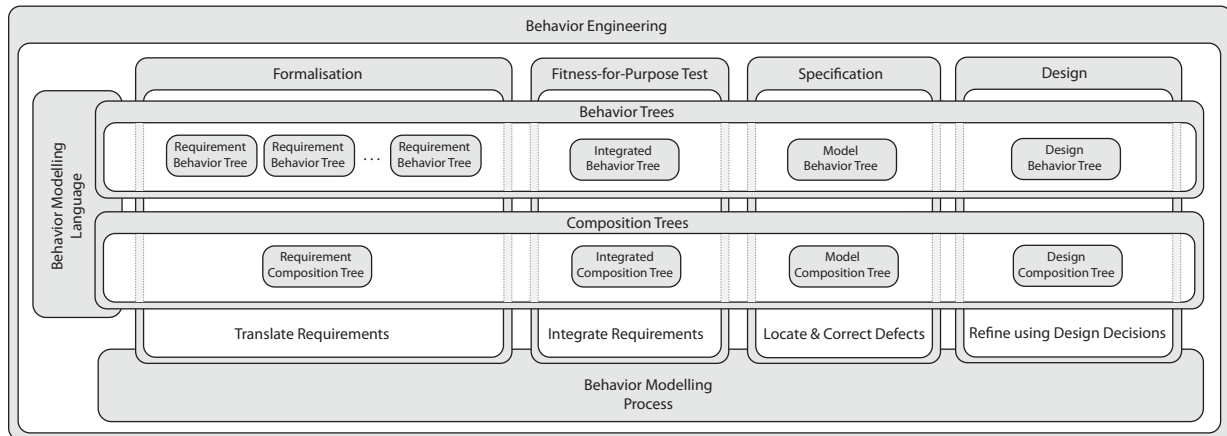


FIGURE 3.1: An Outline of Behavior Engineering

the BML and BMP is demonstrated with a case study which proceeds through the stages of formalisation, fitness-for-purpose testing and specification. Finally, related research involving BE is discussed.

### 3.1 Overview of the Behavior Modeling Language

This section provides an introduction to two of the integrated views of the BML, BTs and CTs. BTs were originally the sole representation used to model systems as part of the BE approach. BTs capture the dynamic aspect of a system by modeling system behavior as the integrated behavior of a group of interacting components. CTs were later added to the BML to capture the static aspect of a system by modeling the composition of the components that form the system. The addition of CTs enabled aliases and other vocabulary problems in the natural language requirements to be resolved.

The BML is introduced with separate overviews of BTs and CTs. In both sections, the notation of the integrated views is described. More focus is placed on the BT overview because it is also necessary to describe the semantics of BTs.

#### 3.1.1 Behavior Trees

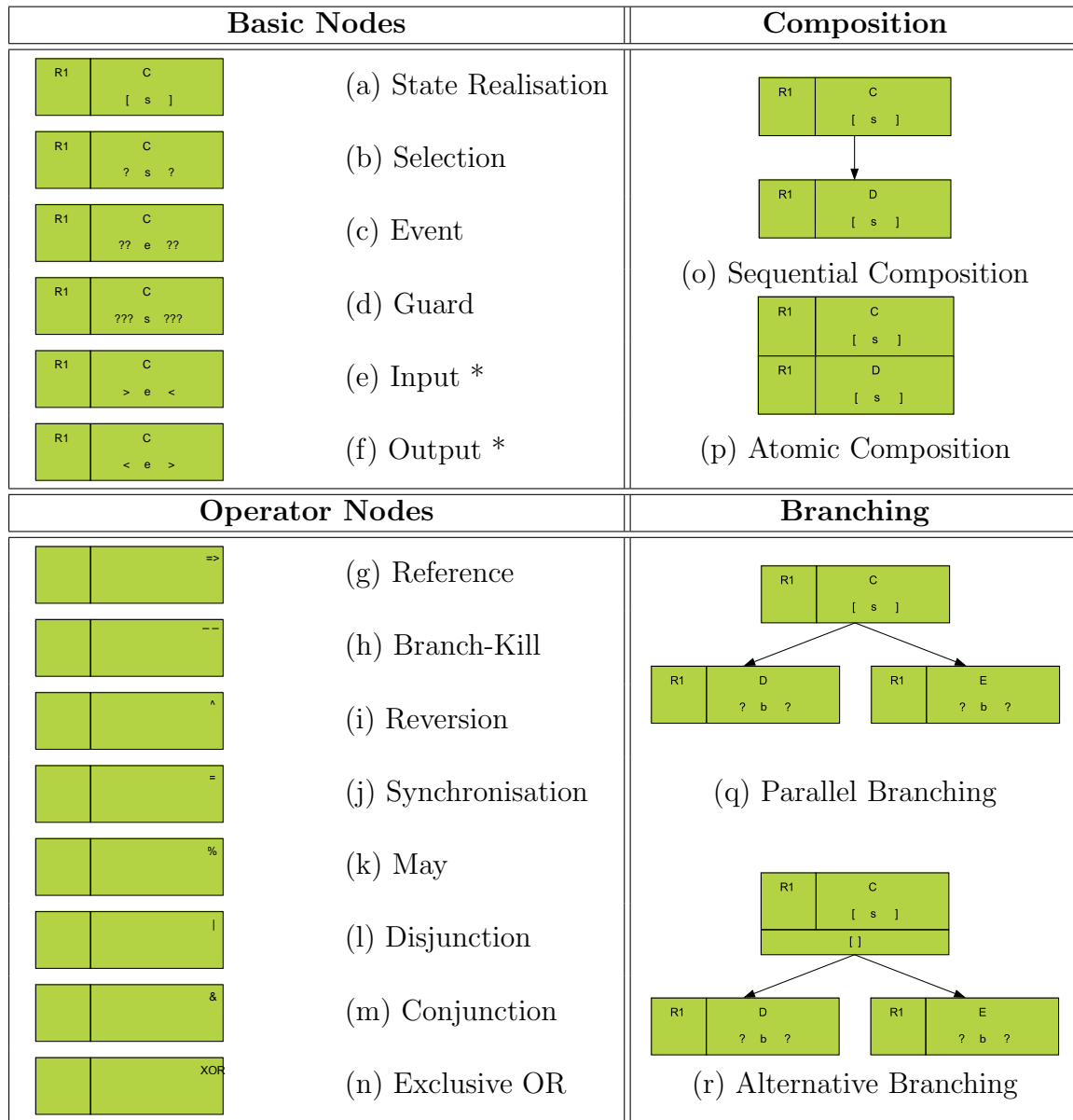
A BT is a formal, tree-like graphical form that represents the behavior of individual or networks of entities which realize and change states, create and break relations, make

decisions, respond to and cause events, and interact by exchanging information and passing control [Dro03]. BTs were created to capture and formalise the dynamic information in natural language descriptions. The focus of this dissertation is on software and systems engineering, so the entities that are captured using BTs are hardware or software components but the concept is widely applicable. In addition to being used for requirements analysis and specification in software and systems engineering, BTs have been used to capture and formalise natural language descriptions of information in legal contracts [MD02], standards, business processes, medical diseases and genetics ([Beh09], Behavior Trees).

In addition to being applicable to a wide variety of domains, in Section 2.3.1 we discussed some other unique characteristics of Behavior Trees. BTs avoid a network structure by using a tree-like graphical form which makes cycles implicit and use a different approach to modularisation that links each node to a component. The syntax of BTs creates a secondary notation [Pet95] which allows expert readers to easily view the flow of control and concurrency in the model. Together these characteristics allow BTs to scale by comprehensively describing large amounts of information where network structures would overwhelm a reader with a web of complex interconnections.

Another important characteristic of BTs is the use of formal transition types which are defined graphically. These transitions types are defined by a formal semantics [CH07] in a lower-level language called the behavior tree process algebra (BTPA). These formal semantics also have recently been revised to be based upon an extension to CSP which supports hierarchical states [CH09].

Several resources are available to aid in learning this new language. A summary of the BT notation that is introduced in this section is shown in Figure 3.2. Appendix B also provides a more comprehensive reference of the BT syntax that can be easily referred to whilst reading this dissertation. Finally, electronic resources accompanying this thesis include animations to demonstrate the semantics of the BT syntax. These animations demonstrate the flow of control for each of the behavior types, the composition and branching types, and node operators. For further details on these animations, refer to the Supplementary Materials section (p. xxvii).



(a) *State Realisation*: Component realises the described behavior; (b) *Selection*: Allow thread to continue if condition is true; (c) *Event*: Wait until event is received; (d) *Guard*: Wait until condition is true; (e) *Input*: Receive message\*; (f) *Output*: Generate message\*; (g) *Reference*: Behave as the destination tree; (h) *Branch-Kill*: Terminate all behavior associated with the destination tree; (i) *Reversion*: Behave as the destination tree. All sibling behavior is terminated; (j) *Synchronisation*: Wait for other participating nodes; (k) *May*: The node may or may not occur (l) *Disjunction*: Logical disjunction applied to group of nodes (m) *Conjunction*: Logical conjunction applied to group of nodes (n) *Exclusive OR*: Logical exclusive or applied to group of nodes (o) *Sequential Composition*: The behavior of concurrent nodes may be interleaved between these two nodes; (p) *Atomic Composition*: No interleaving can occur between these two nodes. (q) *Parallel Branching*: Pass control to both child nodes; (r) *Alternative Branching*: Pass control to only one of the child nodes. If multiple choices are possible make a non-deterministic choice;

\*Note: single characters (> <) / (< >) mean receive/send message internally from/to the system, double characters (>> <<) / (<< >>) mean receive/send message from/to the environment.

FIGURE 3.2: Summary of the Core Elements of the Behavior Tree Notation

### Notation of a Behavior Tree Node

Figure 3.3 displays the contents of a BT node. The general form of a BT node consists of a component realising a behavior. This is displayed in the main part of the node consisting of the name of the component and the behavior it exhibits qualified by a behavior type. The main part of the BT node may also have an optional operator and a label which are used as part of the BT syntax.

The behavior type of the BT node is indicated by delimiters on both sides of the behavior. The behavior types that are used in BTs are state realisations, selections, events, guards, inputs, and outputs. Inputs and outputs can be either internal or external to the system. Figure 3.2 (a-f) shows the graphical form of each of these behavior types.

A state realisation is represented by surrounding the behavior with square brackets ( [ ] ) and indicates that the component exhibits the described behavior. Usually, this is used to indicate the component realises a state. For example, a light that is switched on is represented as the component *LIGHT* realising the state *On*. The behavior of a state realisation can also be an expression which uses the assignment operator,  $:=$ , to assign the result to an attribute. For example, a multi-color light is set to display the color red, *LIGHT colour := red*.

A selection is represented by surrounding the behavior with single question marks ( ? ? ) and is similar to an if statement. If the behavior of the selection evaluates to true, flow of control continues along the branch, otherwise flow of control is terminated along the branch.

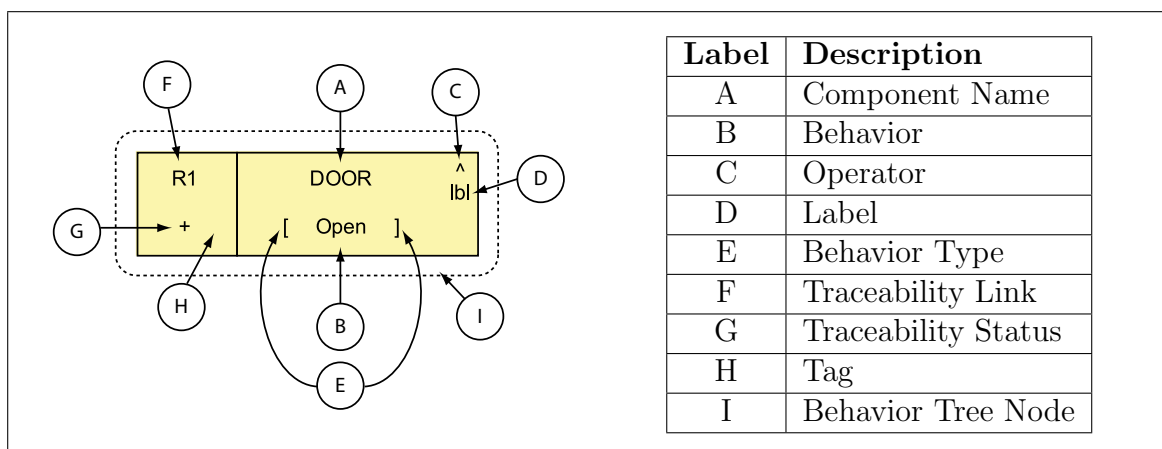


FIGURE 3.3: Behavior Tree Node Naming Conventions

Selections can be used to evaluate the state of a component or a more complex expression composed of attributes. Returning to our light example, this allows the modeling of behavior such as ‘if the light is on’ or similarly ‘if the light is red’.

An event is represented by surrounding the behavior with double question marks (?? ??) and is similar to a when statement. Events describe behavior that may occur in the future: when and if the behavior occurs flow of control continues along the branch. The behavior may occur in another thread in the system, or happen in the environment the system is located in. These imprecise semantics make an event useful for early stage requirements capture, when the precise semantics of the behavior may still be unknown. The precise type of the behavior of the event becomes evident at later stages of modeling and is converted to either a guard, an input or an output.

A guard is represented by surrounding the behavior with triple question marks (??? ???). A guard is a refinement of an event where the behavior is an expression to be evaluated. The expression is continually re-evaluated until it is true, at which point flow of control continues along the branch.

Inputs are represented by surrounding the behavior with angled brackets pointing inwards (> <). Inputs wait until the message described in the behavior is received. If the message is received when control has not been passed to the input event node, it is ignored. Outputs are represented by surrounding the behavior with angled brackets pointing outwards (< >). Outputs send the message described in the node behavior. Both input and outputs are distinguished as either internal by using single angled brackets, or external by using double angled brackets. Internal events are sent and received inside the BT, external events are communicated to the environment outside the BT.

Each BT node also has a tag attached to the left of the main part of the node (Figure 3.3(h)) which contains information to trace the node to the natural language requirements from which it was originally translated. The tag consists of a traceability link and a traceability status. The traceability link consists of an identifier which is used to link the BT node to any associated requirements. The traceability status indicates the status of this link using a set of values. Nodes are coloured based on their traceability status using a traffic-light theme to enable the traceability status of the complete BT to be easily viewed.

Nodes are coloured so that missing behavior is red, implied behavior is yellow and behavior in the original requirement is green. The complete set of traceability status values are:

- **Original Behavior:** The behavior is stated in the original requirements. The node is colored green and the traceability status is left blank.
- **Implied Behavior:** The behavior is not explicitly stated, but is implied by the requirement. The node is colored yellow with a '+' traceability status.
- **Missing Behavior:** The behavior is missing from the requirement. The node is colored red and a '-' traceability status is used.
- **Deleted Behavior:** The behavior is deleted from the system. The node is colored grey and a '- -' traceability status is used.
- **Design Refinement:** The behavior is a refinement of the behavior of the original requirement, indicating that the behavior is implied but the detail to describe it is missing. The node is colored white and a '+ -' traceability status is used.
- **Updated Behavior:** The behavior has been added in the post-development or maintenance phase. The node is colored blue and a '++' traceability status is used. Where several versions of changes have been made, the traceability status can be annotated with a version number e.g. ++v1.0.

Sometimes a requirement describes a behavior involving more than one component. In BE, this is referred to as relational behavior [WCD09]. The relational behavior qualifies the primary behavior described in a standard BT node and can be thought of as a question asked about the primary behavior. The set of allowable questions are: Who, What, Where, When, Why and How <sup>2</sup>. These can be further qualified with a preposition to handle ambiguity. For example, the relation ROBOT ??Place?? [*What()*] BLOCK [*Where()*] TABLE is potentially ambiguous as the BLOCK could be placed on the table or under the table. This ambiguity can be avoided by adding the preposition on or under to the BLOCK's relation to the TABLE.

---

<sup>2</sup>"I keep six honest serving-men (They taught me all I knew); Their names are What and Why and When And How and Where and Who." - from *The Elephant's Child* in *Just So Stories*, Rudyard Kipling, 1902.



## Behavior Tree Syntax

The edges which connect nodes in a BT are used to capture the flow of control throughout the tree. Various multi-threaded behaviors can also be captured with the addition of node operators that describe special operations required when the flow of control moves through several branches of the BT concurrently.

The behavior of a BT edge is defined by composition and branching. Composition defines the behavior of a single branch of BT nodes. Nodes composed sequentially indicate the order of operation, however, other behavior from concurrent threads may be interleaved in between the two nodes. If this is undesirable, then nodes may be atomically composed to form a single uninterrupted group of behavior. Multiple branches can be composed using either parallel or alternative branching. Parallel branching indicates that each branch is executed, creating several concurrent threads of behavior. Alternative branching indicates that only one branch may succeed; if several can be executed then one is chosen non-deterministically.

Several node operators exist to control the operation of concurrent threads of behavior. The node operators are defined in a source node which matches a destination node with the same component, behavior and behavior type. If multiple destinations exist with the same attributes then a label is used to disambiguate. To match a destination with a label the operator is prefixed with the name of the label, e.g. *label.operator*.

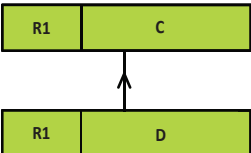

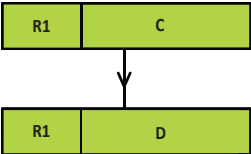



The operators used to control concurrent threads of behavior are reference ( $=>$ ), reversion ( $^$ ), branch kill ( $--$ ) and synchronisation ( $=$ ). The reference operator ( $=>$ ) makes the source node behave as the destination node. The destination node must appear in an alternative branch to the origin. The reversion operator ( $^$ ) also makes the source node behave as the destination node, but all sibling behavior is terminated. The destination node must also be an ancestor of the source node. The branch-kill operator ( $--$ ) terminates all behavior associated with the destination node. The destination node must be in an alternative branch to the origin. The synchronisation operator ( $=$ ) waits for the destination node (or nodes) to execute before allowing the flow of control to continue.

In addition to controlling concurrent threads of behavior, other operators exist which add syntactic sugar to the BT language. These operators include the may operator ( $\%$ ) and

the condition operators, disjunction ( $\parallel$ ), conjunction ( $\&$ ), and exclusive or (XOR). The may operator indicates that the node may execute normally, or may have no effect. In either case, flow of control continues regardless of the outcome. The condition operators are used with an atomic group of nodes, modifying them to have the respective logical operation of disjunction, conjunction, or exclusive or. The condition operators are most commonly used for atomic groups of selections or guards.

### 3.1.2 Composition Trees

Composition Trees complement BTs. They consist mostly of the same information but they are organised differently. CTs are designed to provide the complete system vocabulary in an easy to access format. This has two benefits. It supports teamwork and helps to expose defects not easily visible in a BT. Firstly, CTs support teamwork by coordinating parallel translation of requirements. Secondly, CTs help expose defects not easily visible in a BT such as aliases.

Composition	Basic Nodes
 <p>(a) Aggregation</p>	 <p>(c) Subtype</p>
 <p>(b) Specialisation</p>	 <p>(e) Zero or more</p>
	 <p>(e) One or more</p>
	 <p>(f) Finite number</p>

(a) *Aggregation*: Structural or functional aggregation relation; (b) *Specialisation*: Specialisation relation; (c) *Subtype*: A naming convention for a subtype component defined by a specialisation relation; (d) *Zero or More*: A group of zero or more components; (e) *One or more*: A group of one or more components; (f) *Finite Number*: A specified finite number of components;

FIGURE 3.4: Summary of the Core Elements of the Composition Tree Notation

A CT is segmented into a graphical part and a textual part as shown in Figure 3.5. The graphical part is a tree showing the hierarchy of components that the system is composed

of. Components with multiple instances can be defined using a naming convention where: an asterisk (e.g.  $C^*$ ) specifies a group of zero or more; a plus (e.g.  $C^+$ ) specifies a group of at least one; and a superscript numeral (e.g.  $C^3$ ) indicates a group of the specified finite number. Components are organised into a tree using aggregation relations and specialisation relations. Aggregation relations are defined with a upward pointing arrow and can be both structural (e.g.  $C_{DOOR}$  is physically a part of  $C_{CAR}$ ) and functional (e.g.  $C_{CAR}$  is functionally a part of  $C_{TRAFFIC}$ ). Specialisation relations are defined with a downward pointing arrow and are used to group similar components (e.g.  $C_{BLUE\_CAR}$  is a type of  $C_{CAR}$ ). A summary of these core elements of the composition tree notation is shown in Figure 3.4.

The textual part comprises a table consisting of the complete system vocabulary organised into components. The system vocabulary describes the requirements, states and events which the component is associated with. By placing all the information related to a single component in one place, defects such as aliases (which are not easily visible in a BT) are easily identified.

## 3.2 Overview of the Behavior Modeling Process

The Behavior Modeling Process (BMP) accompanies the BML to provide a scaleable and repeatable approach to system development. While the BML introduced in the previous section could be used independently, there are several advantages of following the BMP. Scaleability is ensured by maintaining a constant short-term memory load. The BMP also uses a bottom-up process that approaches repeatability – removing the need for intuitive miraculous leaps when proceeding from requirements to a specification.

The BMP addresses imperfect knowledge in three stages, each of which have clearly separated tasks and well-defined work products (see Figure 3.1). The first stage of formalisation results in several requirement behavior tree (RBT) and a requirement composition tree (RCT). The second stage involves a fitness-for-purpose test resulting in an integrated behavior tree (IBT) and an integrated composition tree (ICT). The third stage creates an executable specification consisting of a model behavior tree (MBT) and an model composition tree (MCT). The tasks performed during each of the first three stages

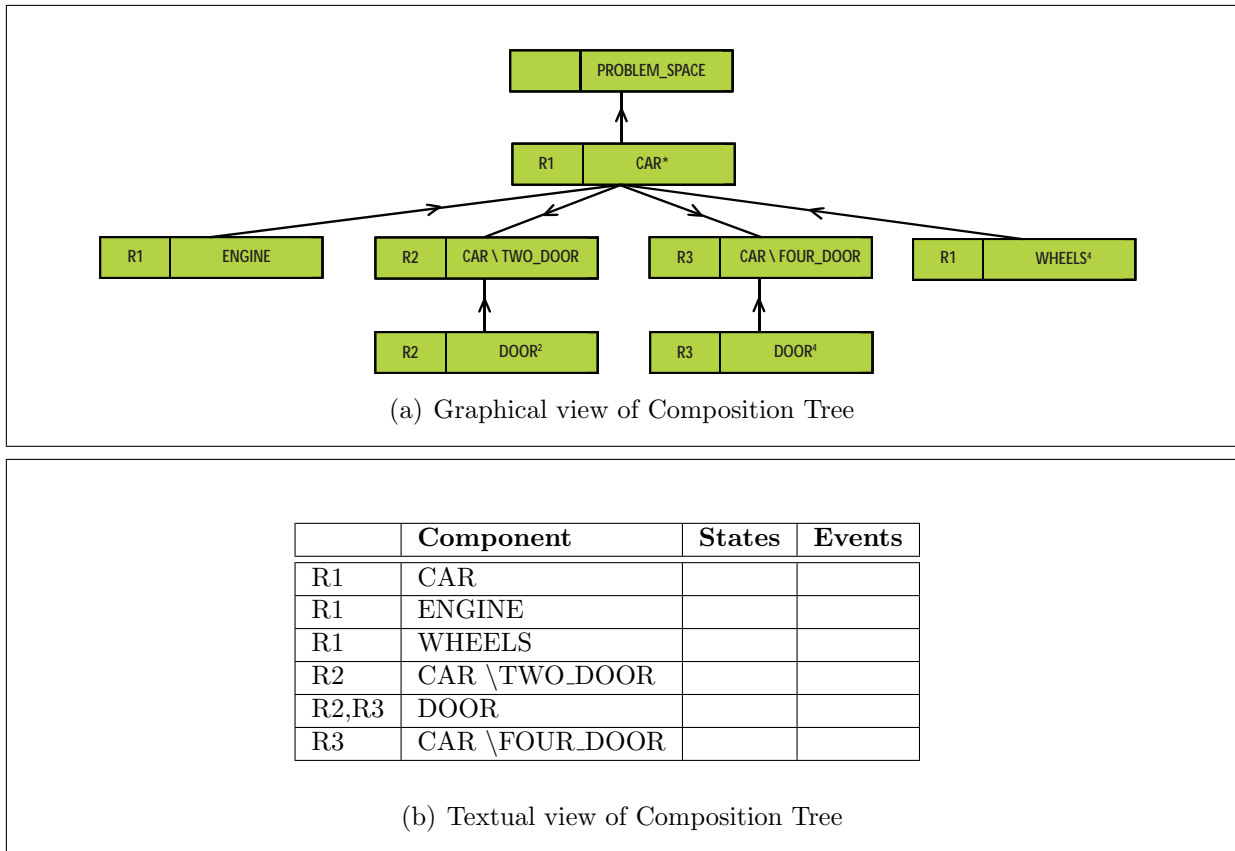


FIGURE 3.5: Example Composition Tree

of the BMP are summarised in Table 3.1. Each of these stages and their tasks will now be discussed.

### 3.2.1 Formalisation

During the formalisation stage each requirement is individually translated into an RBT, and a record of all the information currently captured is stored in a RCT. The aim of the translation stage is to capture the original intent of the system analyst whilst removing any ambiguity present in the natural language description. Original intent is captured by both avoiding the omission of information that is present in the requirement and avoiding the addition of information that is not present in the requirement. Using a bottom-up process to translate the requirements individually avoids unnecessary interpretation and enables rigorous and repeatable translations.

Several tasks are necessary to translate a natural language requirement into an RBT:

Stage	Task	
1. Formalisation	1.1	Identify components and behaviors
	1.2	Associate behaviors with components
	1.3	Determine the type of behavior being exhibited
	1.4	Determine cause and effect
	1.5	Identify implied or missing behavior
2. Fitness-for-purpose	2.1	Locate root node of one tree in another and integrate two trees
	2.2	Add implied or missing behavior if tree does not integrate
3. Specification	3.1	Check for reversions
	3.2	Refine causal behavior
	3.3	Resolve event types
	3.4	Decouple components
	3.5	Manage concurrency
	3.6	Check component initialisation

TABLE 3.1: Summary of the first three stages of the Behavior Modeling Process

(1) identifying components and behaviors; (2) associating behaviors with components; (3) determining the type of behavior being exhibited; (4) determining cause and effect; and (5) identifying implied or missing behavior. There is no set process as to how these tasks have to be applied to translate the requirement. Each task may be performed in sequence for the whole requirement, or all tasks may be performed for one portion of the requirement at a time until the requirement has been translated.

The first task which must be performed during the translation is to identify the components and behaviors present in the requirement. As a general rule, components are identifiable in the natural language as the nouns forming the sentence while behaviors are verbs. During this task aliases for components and behaviors that have previously been identified may be found. When aliases are found, both the preferred name and the alias should be noted in the RCT and the preferred name should be used in place of the alias for any subsequent translations. By convention the shorter name is preferred with the longer name becoming an alias.

The stage after identifying components and behaviors is to identify the component associated with the behavior. This can be as simple as associating the behavior with the component that is the closest noun. More complex situations, however, involve multiple

components sharing a single behavior to form a relation. Translating relations often requires ambiguity to be resolved. For example, the sentence ‘the man saw the woman on the hill with the telescope’ has several possible translations depending on how ambiguity is resolved<sup>3</sup> ([Sim88], p92).

Once a behavior has been associated with a component (or multiple components in the case of a relation), the next task of translation is to determine the type of behavior that is exhibited by the component. The types of behavior were described in the previous section but generally during translation only the state realisation, selection and event behavior types are used. The reason for this is that determining if guard and input/output event behavior types should be used usually requires information present in other translated requirements.

For example, using a guard behavior type requires the knowledge that the behavior is a state or an expression based on attributes belonging to the component. A guard behavior type also implicitly requires another thread to be present in which the behavior that is the subject of the guard is altered, otherwise the guard is redundant. The component’s complete list of attributes may not be available during requirements translation, and knowledge of other threaded behavior is unavailable until integration. Similarly, for an input/output to be defined as external or internal requires knowledge of whether the complementary input/output occurs elsewhere in the behavior tree. In some cases during translation, an event may not even be able to be categorised as input or output until the wider system context is known. For example, the node TELEPHONE ??Rings?? is an external input in the context of the person receiving a telephone call, an external output in the context of the telephone, or it could be an internal input or internal output if both systems are being modelled together. Thus, unless the requirements are explicit, guards, inputs and outputs are captured as an event behavior type during translation.

---

<sup>3</sup>Two sources of ambiguity exist in the sentence and these must be resolved to determine the structure of the relation. The first source of ambiguity is what components are actually on the hill. Possible translations include the woman is on the hill, the man is on the hill, the woman and the telescope are on the hill, the man and the telescope are on the hill, and the man, the woman and the telescope are on the hill. The second source of ambiguity is whether the telescope was used for the man to see the woman or whether the telescope was part of what the man saw. This translation is dependent on the translation chosen for the first source of ambiguity as the telescope must be in the same location as the man to be used to see the woman and must be in the same location as the woman if it is part of what the man saw. Based on this analysis, there are six possible translations, though some translations such as the man being on the hill and seeing the woman who is not on the hill is less likely due to the structure of the sentence.

Once some BT nodes have been captured, they can start to be ordered into an RBT. The ordering of the BT nodes captures the cause and effect present in the requirement. Cause and effect can normally be determined by the structure of the sentence. For example, ‘if the user opens the door the light turns on’, indicates a clear chain of cause and effect where the user opens the door, leading to the door being open, which is followed by the light being on. Sometimes the cause and effect can be ambiguous such as in, ‘if the user opens the door the light turns on and the powertube turns off’. From this requirement it is not clear if the light turning on must precede the powertube turning off, or if the two actions are related at all. In these cases, the cause and effect should be interpreted, the nodes given an implied traceability status, and the issue should be recorded (for the interpretation to be approved later by the client). During translation a RBT normally just consists of nodes linked by sequential composition and possibly parallel branching to indicate concurrent behavior in the requirement. Atomic composition and alternative branching are normally added during specification when a complete view of the integrated requirements is available.

The final stage of requirements translation is to identify implied or missing behavior. Most implied or missing behavior is identified in the requirements integration stage when integration occurs, though occasionally implied or missing behavior can be identified in the requirement itself. This normally takes the form of a missing cause or effect for a captured BT node. For example, if the user pushes the button, an implied behavior is that the button is pushed.

### 3.2.2 Testing Fitness for Purpose

During the fitness-for-purpose stage, RBTs translated in the requirements translation stage are integrated together to form an IBT. The name of this stage is derived from the emergent property of clarifying intention provided by integrating the requirements. The process of integrating the formalised requirements clarifies intention by identifying incompleteness, inconsistency and redundancy in the translated requirements. Integrating the requirements, in essence, performs a fitness-for-purpose test on the translated requirements.

Performing requirements integration relies on two axioms [Dro06c]:

**Precondition Axiom:** “Every constructive, implementable, individual functional requirement of a system, expressed as a behavior tree, has associated with it a precondition that needs to be satisfied in order for the behavior encapsulated in the functional requirement to be applicable.”

**Interaction Axiom:** “For each individual functional requirement of a system, expressed as a behavior tree, the precondition it needs to have satisfied in order to exhibit its encapsulated behavior, must be established by the behavior tree of at least one other functional requirement that belongs to the set of functional requirements of the system.”

Given these two axioms, we can make two inferences. If all the requirements of the system are complete, then they can be integrated to form an IBT. Conversely, if a requirement is unable to integrate, it is a clear indicator of missing behavior, implying that at least one of the requirements are incomplete.

The fitness-for-purpose stage consists of two tasks. The first task is to integrate each RBT until all the RBTs have been integrated to form an IBT. Integration typically consists of finding where the root node of one RBT occurs in another RBT (or the partially formed IBT) and integrating the two trees at that point.

To accomplish this, each RBT can be viewed as a piece of a jigsaw puzzle, which when solved results in a completed IBT. Based on this jigsaw analogy, we can also define some properties of this task. In a jigsaw puzzle, pieces are connected together one at a time. The order in which the pieces are connected is not critical, though it may be easier to solve the puzzle by beginning with certain pieces (such as corner or edges pieces). As more pieces are connected, an overall picture emerges which greatly simplifies the task of identifying missing pieces. A benefit of this process is that even when completed by several different people, the end product of the jigsaw puzzle will be the same as when completed by one individual. Furthermore, the task is repeatable since subsequent solutions of the same task will produce the same outcome, even though it is unlikely that the same sequence of connecting pieces is followed each time.

The only flaw in the jigsaw analogy, is that unlike the pieces in a jigsaw puzzle, it is



possible when attempting to integrate an RBT that a point of integration at which to connect to other RBTs cannot be found. If this occurs, it is a clear indicator of missing behavior in the original natural language requirements. A second task is necessary to address this missing behavior by determining the behavior that will enable the requirement to integrate. This is the only time during this stage where additional behavior can be added and changes can be made to the RBTs. As with the formalisation stage, the way in which this ambiguity is resolved may result in different interpretations thereby reducing repeatability. This weakness can be addressed by discussing possible interpretations with the client to determine which solution best preserves their original intent.

### 3.2.3 Specification

The IBT formed during the fitness-for-purpose stage provides a complete view of the information in the requirements and allows the original intention of the requirements to be clarified. The next stage creates an executable specification from the integrated view provided by the IBT by addressing issues of incompleteness, inconsistency and redundancy. The executable specification represented in the resulting MBT can then be used to perform further investigation of the system. This is commonly achieved by translating the MBT into other formal languages to perform model checking or simulation.

Zafar ([Zaf09], Appendix B, p289) identified a sequence of tasks (or analysis and design patterns) required to convert an IBT into a MBT: (1) check for reversion nodes; (2) check for requirements completeness and correctness; (3) refine causal behavior; (4) resolve event types; (5) decouple components; (6) manage concurrency; and (7) check component initialisation. The steps that must be taken to perform each of these tasks will now be described.

The first task in converting an IBT to a MBT is to check for missing reversion nodes, that is nodes with a reversion operator. Each leaf node in an IBT potentially has a missing reversion node that is required for the thread of behavior to continue. Before adding a missing reversion node, an appropriate destination node to link to the originating reversion node must be found. An appropriate destination node is an ancestor of the leaf node with a

system context, namely the state of all components in the system, that matches the context of the leaf node. Sibling branches should also be checked when adding reversion nodes to avoid creating race conditions<sup>4</sup>.

The second task is to check for requirements completeness and correctness. Successfully integrating all the RBTs to form an IBT does not preclude further missing behavior from being present in the IBT. The integrated view of the IBT, however, does simplify the task of locating further conflicting, redundant and missing behavior. These behaviors can be located and corrected in the following ways. Conflicting behavior describe two distinct paths through the tree resulting in the same behavior occurring. Once identified, conflicting behavior should be resolved by deleting or modifying nodes as necessary. The issues that were identified and how they were resolved should also be recorded. Redundant behavior describes two similar paths through the tree resulting in the same behavior occurring. Redundant behavior can often be simplified by using node operators such as the reference operator. Finally, missing behavior can be identified by checking constraints, by checking events which are applicable elsewhere, and by checking the behavior of individual components. Requirements completeness and correctness can then be partially verified by ensuring any constraints in the tree are satisfied. If constraints are satisfied, they can be given a deleted traceability status. This allows the deleted nodes to remain in the tree but they can also be hidden with tool support.

The third task is to refine causal behavior. Causal behavior occurs when the integration of RBTs results in parallel branches of interleaved behavior. The semantics of parallel branching allows the interleaving of behavior from branches in any order. Sometimes, one of these possible interleavings may be preferred so the causal behavior should be refined to capture this.

The fourth task is to resolve event types. The IBT provides a complete view of the system that allows events to be resolved into either a guard, input or output. If the behavior of the event is an expression that is updated in another thread of behavior, then the behavior type should be changed to a guard. If the behavior is for passing a message, however, and another

---

<sup>4</sup>Race conditions are possible because a reversion terminates all sibling behavior of the destination node. This makes it possible for a race condition to exist between concurrent behavior finishing and the reversion occurring.

event exists in the system matching the behavior, then both events should be changed to internal inputs and outputs. Conversely, if there is no matching event, then the event should be changed to an external input or output.

The fifth task is to decouple components. This involves identifying environmental components that are independent to the system and decoupling them into separate environmental threads. Environmental components typically can be interacted with at any time regardless of the state of the system. Once identified, environmental components are decoupled from the system and placed in parallel branches of the root node. These environmental threads then interact with the system using internal inputs and outputs.

The sixth task is to manage concurrency. Determining alternative branching and atomic composition is avoided until the integrated view of the IBT is available. Interleaved threads should be checked to ensure they can run concurrently. If they are mutually exclusive, this should be indicated by reassigning the branch as alternative. If only a section of the behavior does not allow interleaving, it can be made into an atomic block using atomic composition.

The seventh and final task is to check component initialisations. The MBT should be analysed to determine if any components need to be initialised prior to the system starting. The components can be initialised at the root of the tree in an atomic block prior to the current root of the tree.

The following section applies the first three stages of the BMP and their associated tasks to the modeling of a simple microwave oven.

### 3.3 Modeling the Simple Microwave Oven

The system used for this case study is based upon the One-minute Microwaver which was used to describe state-transition diagrams of the Shlaer-Mellor method [SM92]. This system has been described in various subsets of UML [WW03, Mel07b] (See also Appendix A). It has also been used previously to discuss in detail how to perform requirements analysis with BE [Dro06c]. The One-minute Microwaver is a simplified microwave oven that cooks in increments of one minute, as determined by the number of times a button is pushed. The system has the restriction that when the door is open, the oven is unable to cook due to the

potentially harmful effects. The requirements for the microwave oven are shown in Table 3.2.

We will now discuss how to model the One-minute Microwaver using BE for the BMP stages of formalisation, the fitness-for-purpose test and specification. The resulting specification is used as the basis for demonstrating the design stage of BE in Chapter 4 and the development of an embedded controller using BE in Chapter 5.

### 3.3.1 Formalisation

Figure 3.6 shows the result of translating the first requirement of the microwave oven into the resulting RBT and RCT. The rest of the section discusses the steps that are taken to produce this model.

We begin translation by identifying the components and behaviors that are involved in the requirement. In the following text, the requirement has been modified such that **COMPONENTS** are bolded and capitalised and *behaviors* are bolded and italicised:

R1. There is a single **CONTROL BUTTON** available for the use of the **OVEN**.  
If the **OVEN DOOR** is *closed* and **YOU** *push* the **BUTTON**, the **OVEN** will start *cooking* (that is, *energize* the **POWER-TUBE**) for *one minute*.

Requirement	Description
R1	There is a single control button available for the use of the oven. If the oven door is closed and you push the button, the oven will start cooking (that is, energize the power-tube) for one minute.
R2	If the button is pushed while the oven is cooking, it will cause the oven to cook for an extra minute.
R3	Pushing the button when the door is open has no effect.
R4	Whenever the oven is cooking or the door is open, the light in the oven will be on.
R5	Opening the door stops the cooking.
R6	Closing the door turns off the light. This is the normal idle state, prior to cooking when the user has placed food in the oven.
R7	If the oven times out, the light and the power-tube are turned off and then a beeper emits a warning beep to indicate that the cooking has finished.

TABLE 3.2: Requirements of the Microwave Oven

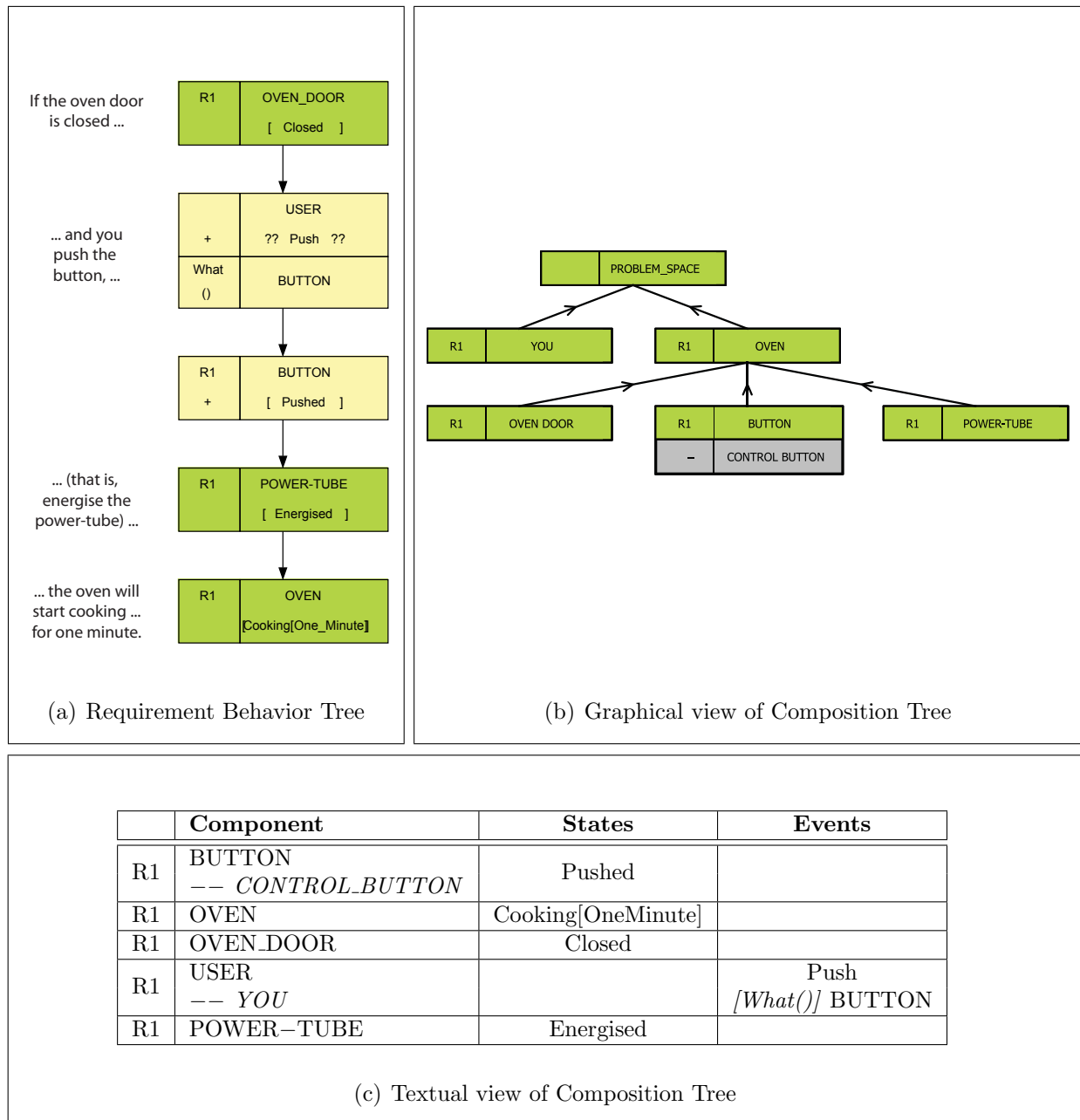


FIGURE 3.6: Initial Translation of Requirement #1 of the Microwave Oven

The first sentence identifies two components of the microwave oven system: the Oven and the Control Button. As there is no behavior associated with these components yet, we can only add them to the CT with a R1 traceability link. The second sentence contains a description of the required behavior.

The first fragment of behavior states “If the oven door is closed ...”. The ‘if’ in the sentence indicates the oven door component exhibiting the closed behavior is a precondition

for the remaining behavior described in the sentence. The closed behavior is captured as a state realisation, as indicated by the [ ] square brackets surrounding the behavior in the BT node. This node, and all other nodes for this requirement, have a traceability link of R1, indicating the requirement from which they originated.

The next fragment of behavior “... and you push the button ...”, describes a behavior, push, taking place that involves two components, You and Button. In BE, we refer to behavior that is shared between two or more components as relational behavior. Relations are captured by firstly identifying the primary behavior involved and the associated component, You push. It is a convention of the BML, however, to avoid the usage of personal pronouns. To conform to this convention, we change the name of the You component to User, recording the change with an implied traceability status. Because the push behavior doesn’t necessarily occur immediately but may occur at some point in the future we capture it with an event behavior type. The relational portion can now be added, indicating that the push behavior is related to the Button component with the qualifier of What.

Whilst adding the Button component to the CT, it should be evident that previously in the requirement a component called Control Button was translated. If both Control Button and Button refer to the same component, then one of these component names must be recorded as an alias. Aliases can complicate a specification, as behavior may appear to belong to several distinct components which are in fact the same component. An alias is indicated in the CT, but using the deleted traceability status results in the alias being greyed. Subsequent translations of the alias must then be substituted with the preferred name, which is generally the most concise term.

Returning to translation, as we know from the requirement that OVEN\_DOOR [Closed] occurs prior to USER ??Push?? [What()] BUTTON, we can link the two nodes with a sequential connector. This indicates that after the Oven Door is closed, at some point the user may push the button but other behavior may also occur in between if other threads exist. Generally, specialised multi-threaded behavior such as atomic blocks and alternative branching is added during specification.

The next fragment of behavior following “you push the button” is that “the oven will start cooking (that is, energise the power-tube) for one minute”. This has the implication

that USER ??Push?? *[What()]* BUTTON was successful, resulting in the button being pushed. The implied behavior, BUTTON [Pushed] should be added into the RBT with an implied traceability status. Also for each implied or missing behavior, the associated issue should be added to the issue list.

We may now continue with translating the final fragment of behavior. The behavior fragment consists of two BT nodes, OVEN[Cooking[OneMinute]] and POWER-TUBE[Energised]. The Cooking[OneMinute] behavior consists of the state Cooking and the sub-state OneMinute. A sub-state is used to further qualify the primary state. Note that OneMinute cannot be captured as a relation because OneMinute is not a component.

The behavior fragment is ambiguous as to whether OVEN[Cooking[OneMinute]] or POWER-TUBE[Energised] occurs first. Using intuition and domain knowledge, we can ascertain that the Power-tube must be energised for the Oven to realise the cooking state.

Each of the remaining requirements can be translated similarly, resulting in the RBTs in Figure 3.7. The RBTs are accompanied by an issues list, shown in Table 3.3, which describes the issues associated with implied and missing behaviors that were translated.

Requirement	Issue
R1	Is pushing of the button successful? i.e. results in the button being pushed.
R2	Does the user push the button?
R3	Does the user push the button?
	Has no effect on the button?
R5	The oven must be cooking prior to stopping the cooking.
	Does the user open the door?
	The power-tube must be turned off to stop the cooking.
	Stops the cooking of the oven?
R6	Prior to the oven cooking?
	The oven's normal idle state?
	Does the user close the door?
	The door must be open to be able to be closed.
R7	The oven must be cooking to be able to time out.
	Is the light or the power-tube turned off first? Is the ordering important or arbitrary?
	The oven has finished cooking?

TABLE 3.3: Issues found during translation of requirements of the One-minute Microwaver

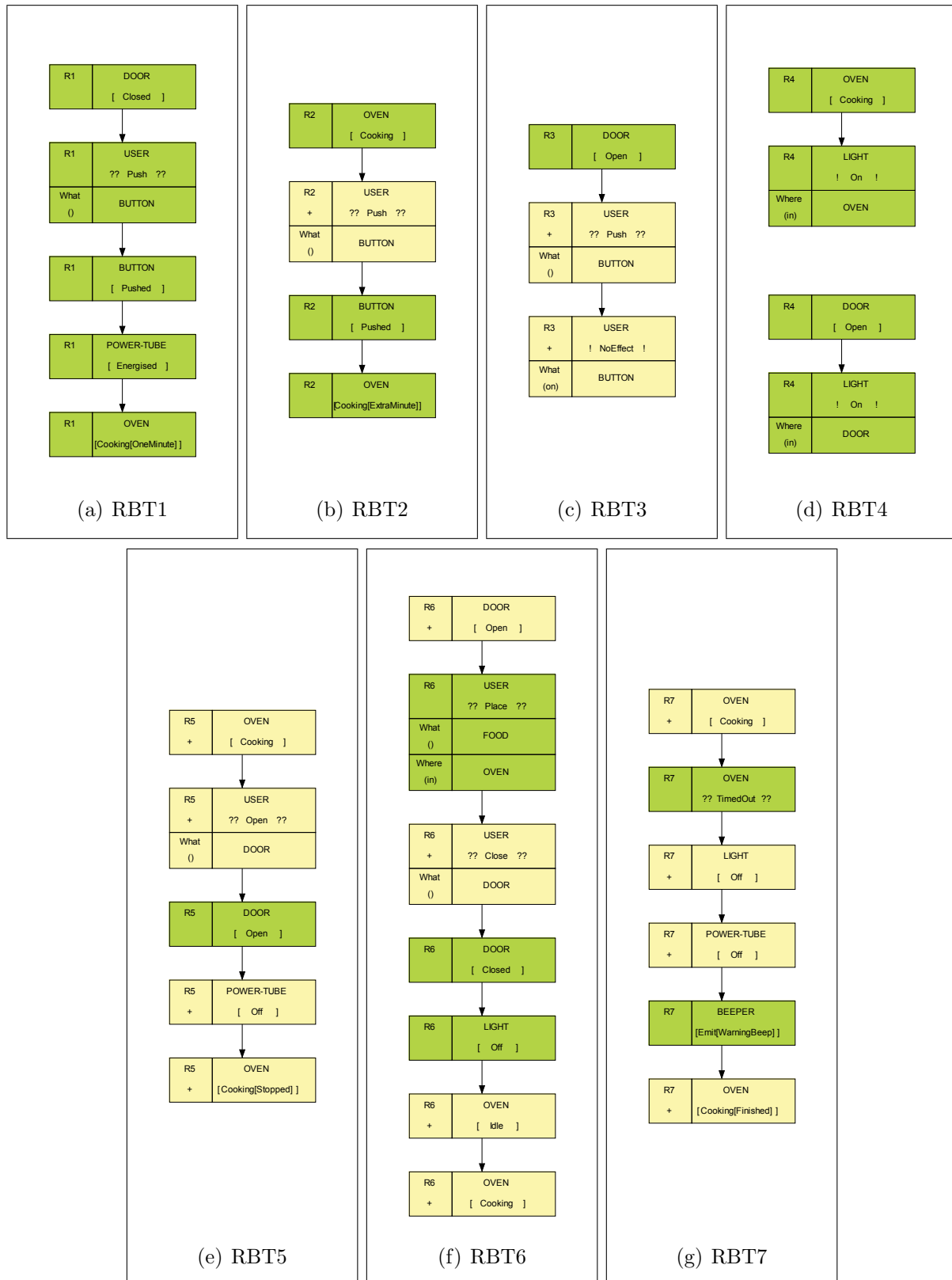


FIGURE 3.7: The Requirement Behavior Trees of the Microwave Oven



### 3.3.2 Testing Fitness for Purpose

Figure 3.8 shows the result of integrating RBT2 and RBT5 to form a partial IBT, and the result being integrated with RBT7. Figure 3.9 shows the IBT resulting from integrating all of the RBTs of the microwave oven. Note that some RBTs such as RBT3 and RBT4 are integrated several times to form the IBT. The associated ICT is shown in Figure 3.10.

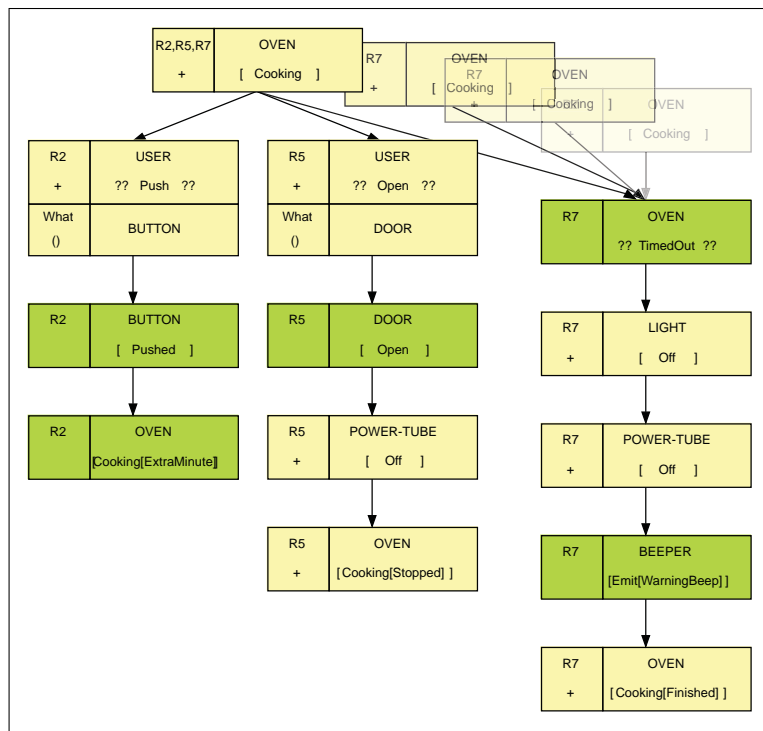


FIGURE 3.8: Integrating RBT2 and RBT5 and integrating the result with RBT7

The following integrations were made to form the IBT:

- RBT3, RBT4 and RBT6 share the integration node DOOR [Open]
- RBT1 and RBT6 share the integration node DOOR [Closed]
- RBT2, RBT4, RBT5 and RBT7 share the integration node OVEN [Cooking]
- RBT3, RBT4 and RBT5 share the integration node DOOR [Open] (descendant of OVEN [Cooking])

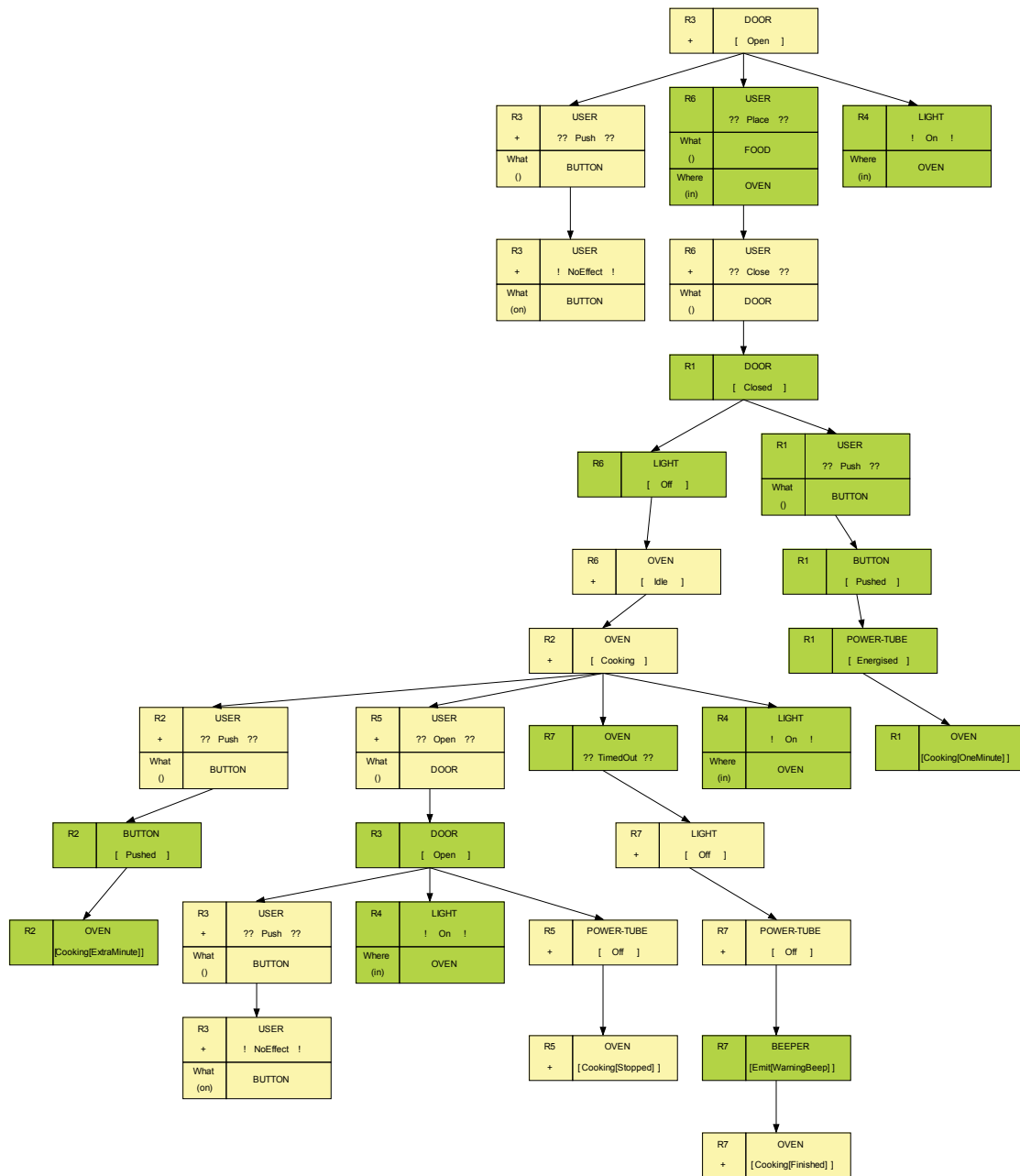
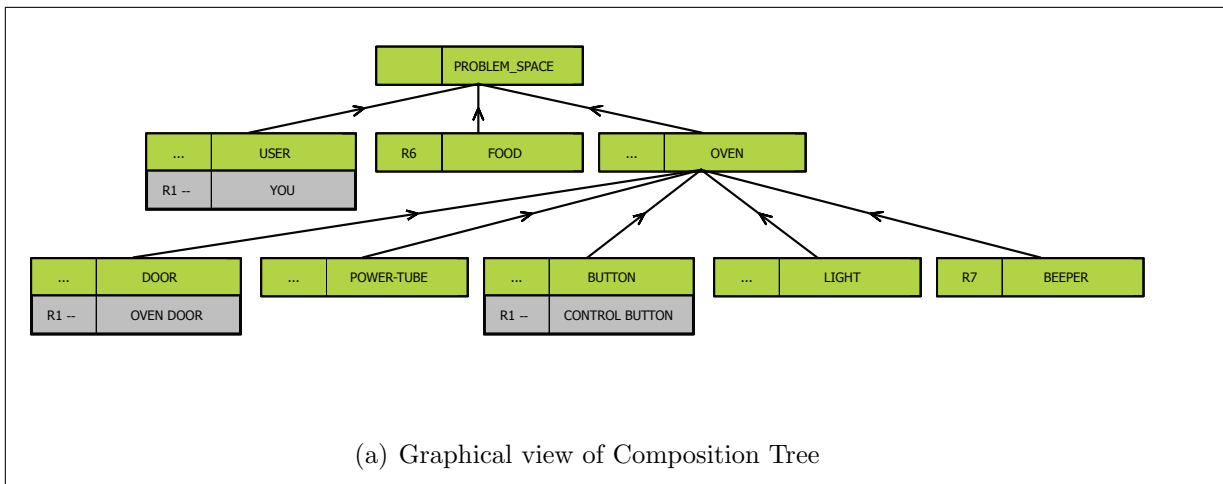


FIGURE 3.9: The Integrated Behavior Tree of the Microwave Oven



Component	States	Events
BUTTON -- CONTROL BUTTON	Pushed	
OVEN	Cooking[OneMinute]	Timed Out
	Idle	
	Cooking[ExtraMinute]	
	Cooking[Stopped]	
	Cooking[Finished]	
DOOR -- OVEN DOOR	Closed	
	Open	
USER -- YOU	No Effect	Push
	[What(on)] BUTTON	[What()] BUTTON
		Place
		[What()] FOOD
		[Where(in)] OVEN
		Close
POWER-TUBE	Energised	
	Off	
LIGHT	Off	
	On	
BEEPER	Emit	
FOOD	[What()] Warning Beep	

(b) Textual view of Composition Tree

FIGURE 3.10: The Integrated Composition Tree of the One-minute Microwaver

### 3.3.3 Specification

Figure 3.11 shows the partially completed MBT after the system component was chosen and looping behavior was resolved. The following actions were applied to the IBT:

- OVEN is a system component, any node in the MBT where it is the primary component uses a double-line border.
- Branch R3 + USER ??Push?? [*What()*] BUTTON reverts back to itself to allow the event to occur multiple times. Note this applies to both the branch of OVEN [Open] and the branch of OVEN [Cooking].
- Branch R1 USER ??Push?? [*What()*] BUTTON should begin after OVEN [Idle] because it is a convention that events occur after a system state realisation. The leaf node R1 OVEN [Cooking[OneMinute]] also implies that OVEN [Cooking] applies, but rather than using a reference, the whole branch can be inserted between OVEN [Idle] and OVEN [Cooking]. Note that this strengthens the behavior described in the IBT because parallel branching includes the possibility of interleaved behavior.
- The leaf node of Branch R2 USER ??Push?? [*What()*] BUTTON (child of OVEN [Cooking]), OVEN[Cooking[ExtraMinute]] implies that OVEN[Cooking] follows this behavior. This is included by including a reversion to OVEN[Cooking].
- The leaf node of Branch R7 OVEN ??TimedOut??, OVEN[Cooking[Finished]] implies that the Oven is no longer in the cooking state. The current state of Door, Button and Power-tube matches the Oven Idle system state, so a reversion to OVEN[Idle] is added.
- The leaf node of branch R5 USER ??Open?? [*What()*] DOOR, OVEN[Cooking[Stopped]] implies that the Oven is no longer in the cooking state. OVEN[Idle] is not suited to be a reversion destination because Door is currently in the state open and in OVEN[Idle] Door is in the state closed. This branch requires a new system state OVEN [Open] to be added after the root node R1 + DOOR [Open], which can then be used as a reversion destination for the branch.

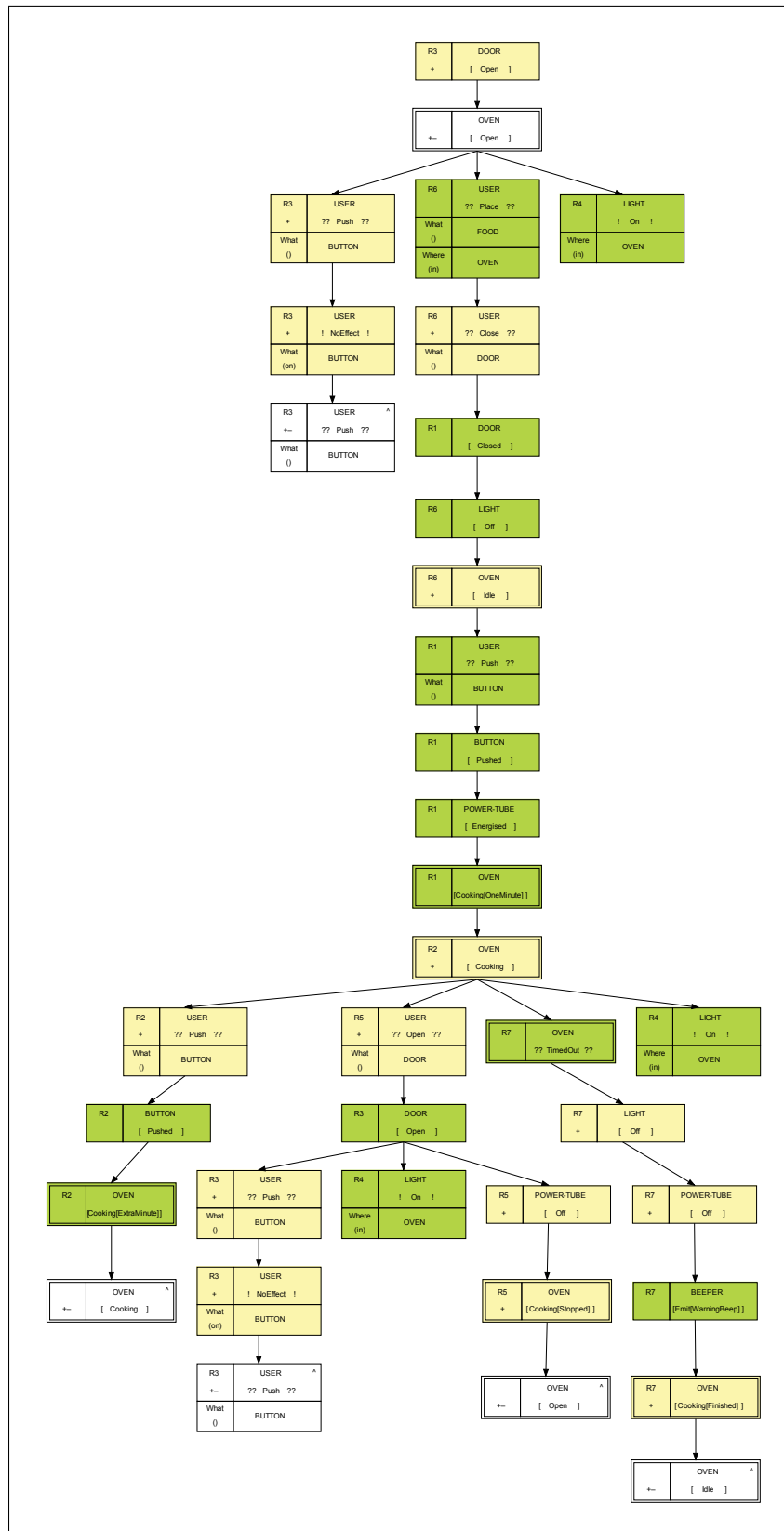
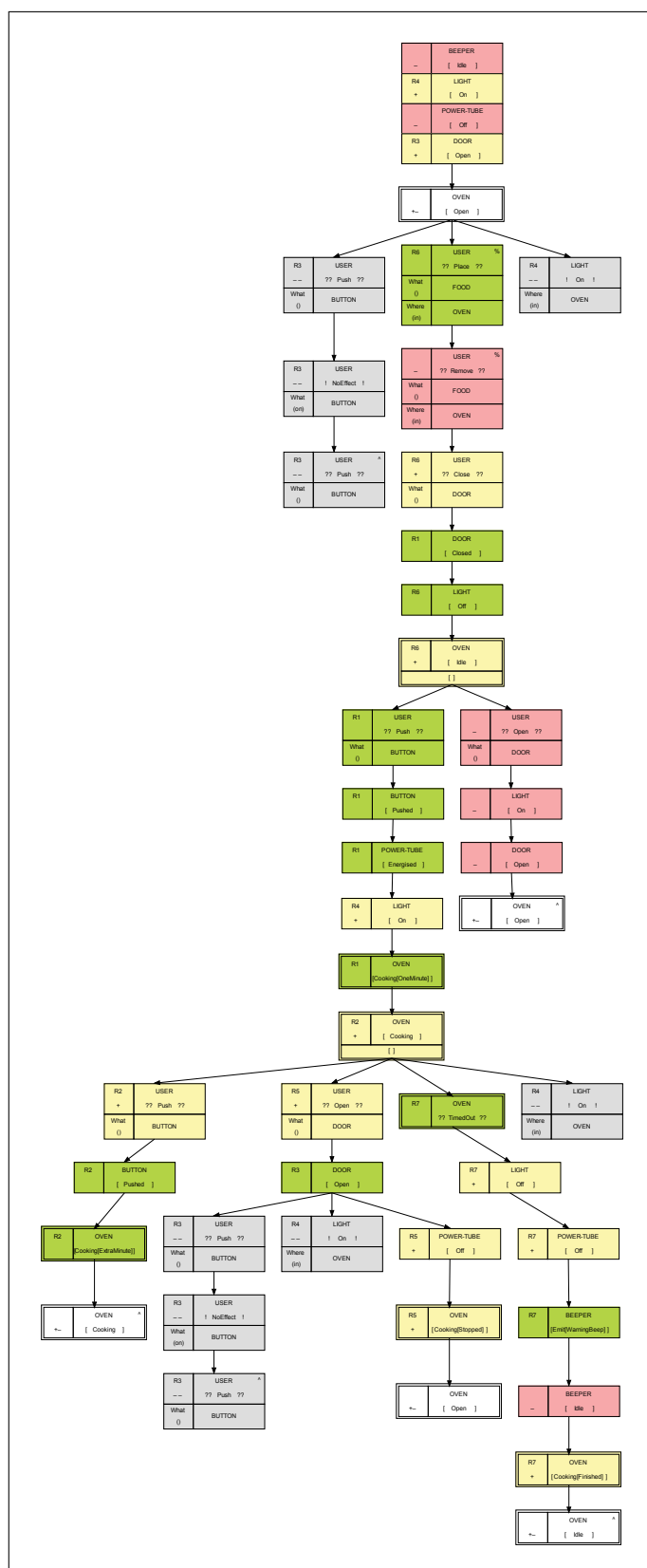


FIGURE 3.11: Partial Model Behavior Tree #1

Figure 3.12 shows a second stage of the partially completed MBT after missing behavior was identified by checking constraints, events and initial states. The following actions were applied:

- The LIGHT !On! assertion node (child of OVEN [Idle]) is satisfied by adding an initial state R4 + LIGHT [On]. The initial state is linked to requirement R4 with an implied traceability status because it is implied by the assertion node. The assertion node LIGHT !On! is given a deleted traceability status because it is satisfied.
- The USER !NoEffect! assertion node (close descendent of OVEN [Open]) is satisfied as long as no behavior is added after the USER ??Push?? [*What()*] BUTTON node. Therefore the branch can be given a deleted traceability status because it is satisfied.
- The LIGHT !On! assertion node (child of OVEN [Cooking]) requires R4 + LIGHT [On] to be added before OVEN[Cooking[OneMinute]]. The LIGHT !On! assertion node can be given a deleted traceability status.
- The USER !NoEffect! constraint (descendant of USER ??Open?? [*What()*] DOOR) is satisfied by the previous assertion once the OVEN [Open] reversion occurs. For this constraint to also be satisfied before the reversion occurs, none of the other events that are children of OVEN[Cooking] can occur. This can be ensured by using alternative branching, thus making it permissible for only one of the branches to succeed (This also avoids a potential race condition from occurring between the branches). The assertion node can now be given a deleted traceability status.
- The LIGHT !On! assertion node (descendant of USER ??Open?? [*What()*] DOOR) is already satisfied for OVEN [Cooking], so can be given a deleted traceability status.
- The USER ??Place?? [*What()*] FOOD [*Where(in)*] OVEN should not be required to occur for USER ??Close?? [*What()*] DOOR to occur. This can be indicated by using the may (%) operator. Also, if the User can place Food in the Oven, they should also be able to remove Food from the Oven. This can be specified by adding the node USER ??Remove?? [*What()*] FOOD [*Where(in)*] OVEN also with the may operator.



- After OVEN [Idle] it is possible that the User may open the door but this has not been specified. Because this behavior was not specified in the requirements, it should be added to the MBT with a missing traceability status. The node USER ??Open?? [What()] DOOR should be added resulting in the behavior DOOR [Open]. Requirement R4 requires that LIGHT [On] be added before DOOR [Open]. The branch should then be completed with a reversion to OVEN [Open].
- The Beeper component enters the Emit[WarningBeep] state, but never enters another state, indicating at least one possibly missing state. The node BEEPER [Idle] is added after BEEPER [Emit[WarningBeep]] with a missing traceability status.
- The components Power-tube and Beeper should also have initial states. Beeper should begin in the Idle state and Power-tube should begin in the Off state.

The MBT is completed by decoupling environmental components and by resolving events. The finalised MBT is shown in Figure 3.13. Because the User component is not part of the Oven system, it may interact with Door and Button at any time regardless of the current state of the microwave oven. To represent this in the MBT, the User component is decoupled from the main part of the BT and replaced by concurrent environmental threads for independent events using parallel branching. Events in these environmental threads are replaced by external input to indicate they are received from the environment and relay information using internal output to the main portion of the MBT. The following actions were taken to finalise the MBT:

- USER >>Push<< [What()] BUTTON causes PushButton internal event
- USER >>Close<< [What()] DOOR and USER >>Open<< [What()] DOOR are related, so they are placed in the same environmental thread. USER >>Close<< [What()] DOOR causes CloseDoor internal event, and USER >>Open<< [What()] DOOR causes OpenDoor internal event.
- USER >>Place<< [What()] FOOD [Where(in)] OVEN and USER >>Remove<< [What()] FOOD [Where(in)] OVEN are related, so they are placed in the same



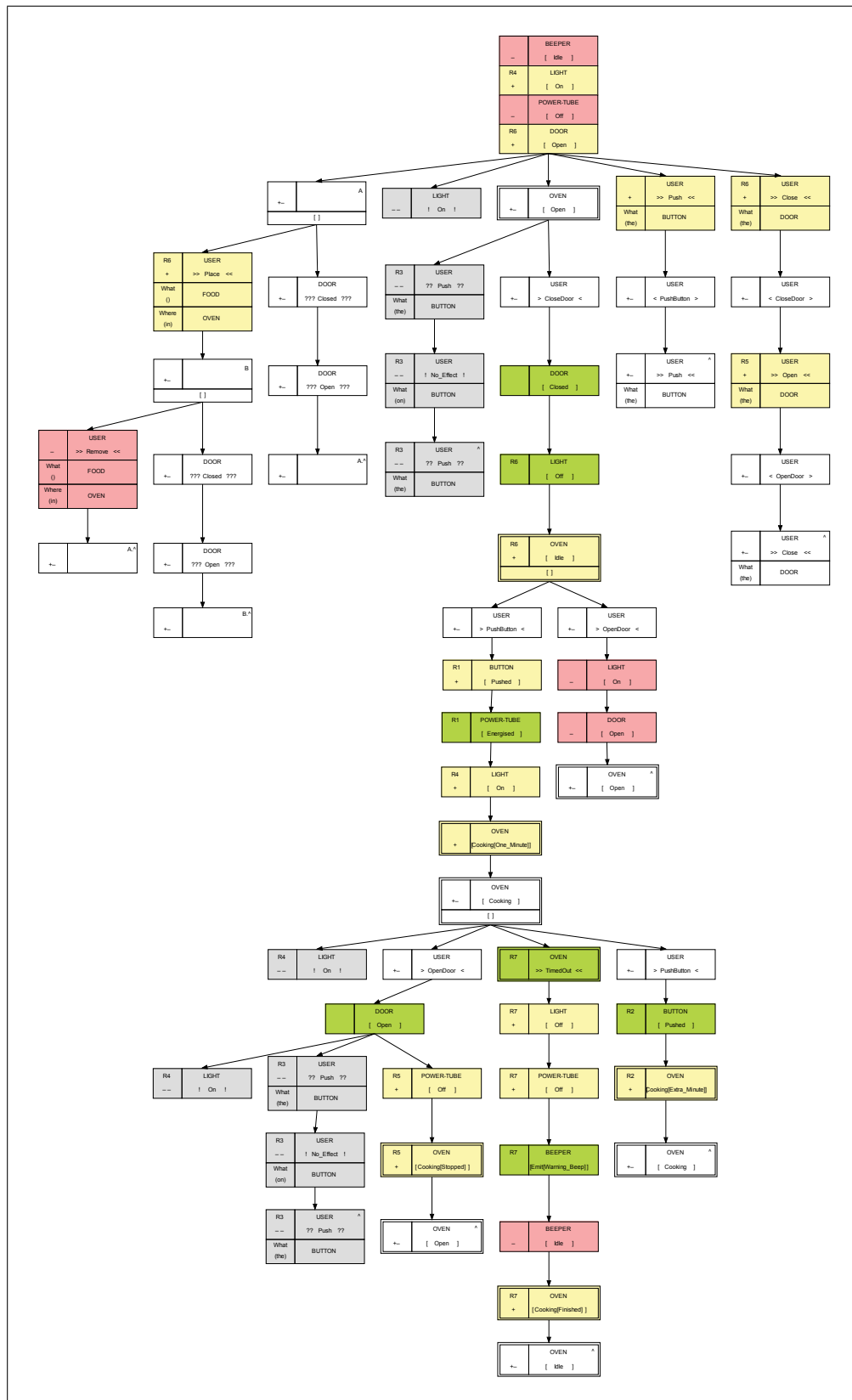


FIGURE 3.13: Model Behavior Tree of the One-minute Microwaver

environmental thread. The Place and Remove external input events are also related to the state of Door, so a guard is added to check the Door state and only allow the events to occur when DOOR [Open].

### 3.4 Related Research

Current research in BE has focused primarily on the use of the specification resulting from applying the BE approach to a set of natural language requirements. The BE approach has been utilised to deal with requirements change, perform model-checking, and perform safety and security analysis.

Wen and Dromey [WD04, WD06] used BE to investigate requirements change by creating a traceability model that determines the impact that changes in the functional requirements have on the system. Changed requirements are formalised and integrated into the existing BT specification and compared to the original specification. Comparison with the changed tree reveals how the change affects the system architecture, what components are affected and what behavioral changes are required to be made to the components. The specification can also be compared at different stages to manage and record the evolution of the system resulting from changes to the requirements.

The BE specification can be used to perform model-checking which allows automatic testing of whether safety and security properties are met. Model-checking is performed using the Symbolic Analysis Laboratory (SAL) tool suite whereby a software tool is used to convert the BE specification into the SAL language [GWY08]. Model-checking with BE has also been utilised to perform failure modes and effects analysis (FMEA) [GLYW05], which determines if hazardous states can occur during the failure of components in the system. Together with extensions to the BT syntax, Timed and Probabilistic FMEA can also be performed on BE specifications. Timed FMEA [CGW08] is performed by transforming timed BTs into a format suited to the timed model checker UPPAAL. Probabilistic FMEA [GCW07] is performed by transforming probabilistic BTs into a format suited for the probabilistic model checker, PRISM.

Zafar [ZD05a, ZD05b, ZWCD06] applied BE to create a new approach to ensuring safety

and security of systems by integrating safety and security requirements into a BT. The resulting extension, BT-RBAC, enables role-based access control to be specified, validated and automatically verified using the model-checking capabilities of BE. This approach offers improvements over other approaches which treat security in isolation from software and systems engineering causing problems with formal specification and verification.

## 3.5 Discussion

The main limitation of the current BE methodology presented in this chapter is that it can only be initiated after a set of natural language requirements have been written. This is not an issue for many systems, where there is at least a broad concept of what needs to be delivered that can be used as a starting point. The need for natural language requirements, however, does become an issue when a system is being described around a predefined set of components with an allowable set of relationships. This class of systems commonly occurs when the system requires a problem to be analysed and a solution to be engineered. For this class of systems, it is more natural for a methodology to capture the structure of the system prior to describing the behavior that operates using that structure. This need can be addressed in future work with the addition of structure trees to the BML to allow the graphical definition of domain-specific constraints.

A limitation of the case study presented in this chapter is that it does not show extensive use of the multi-threaded capabilities of BTs. Other than the environmental threads in the MBT, the behavior of the one-minute microwaver only requires a single thread of control. One of the main benefits of the BT language is that it provides a multi-threaded component based representation, so in Chapter 6 modeling of the automated train protection system demonstrates the capturing of a system with multi-threaded behavior.

## 3.6 Conclusion

A scaleable methodology is necessary to manage the complexity of developing the ever larger software-intensive systems of software and systems engineering). Mainstream approaches to

software and systems engineering, however, need an intuitive miraculous leap to proceed from natural-language requirements to a formal specification. In this chapter, we introduced the BE approach which provides a scaleable methodology to create a formal specification from natural-language requirements. The scaleable methodology of the BE approach is a result of the tight coupling of the BML and the BMP. When used together, the BML and the BMP ensure a scaleable methodology by minimising the local problem space at all stages of modeling. Firstly, each requirement is translated independently. Secondly, each translated requirement is integrated one at a time to form a complete integrated view of the system. Finally, changes are made one at a time to the integrated view to create an executable specification.

In addition to a scaleable methodology, the BE approach also has the following benefits:

1. The same language is used throughout the whole process. This ensures traceability to the original natural language is maintained at all stages of modeling.
2. The BMP is an efficient means of locating and correcting defects at the early stages of development. Identifiable defects include incomplete, conflicting and redundant behavior and aliases. These defects are often difficult to detect by other means as they can be widely distributed amongst hundreds of requirements.
3. The BT language of the BMP provides a multi-threaded component-based representation that separates integration from computation. This makes the BE specification an excellent candidate for creating a component-based design.
4. The wholistic view of the system provided by the specification resulting from the BE approach is quite useful. Current applications of the specification include analysing requirements change, performing model-checking and safety and security analysis.

# 4

## Design with Behavior Engineering

In this chapter, we introduce the new design stage of the BMP. This new design stage describes how to create a design that satisfies the BE specification by individually applying design decisions to the specification. Because these design decisions can be applied individually and recorded in an evolving DBT, the design stage maintains a uniform local problem space regardless of the size of the global problem space. As a result of defining a BE component model and an extension mechanism for BTs, BE designs also gain the productivity benefits of an MDE framework.

This chapter is organised into six sections. The first section discusses the range of potential applications of a BE specification and argues the need to extend the BMP to include a design stage. The second section describes a component model for deploying BE designs and it investigates the hardware component model that is used as a foundation for

the BE component model. In the third section, we describe an extension mechanism that allows BTs to capture domain-specific behavior. In the fourth section, we discuss the design decisions that form the new design stage of the BMP and in the fifth section, the design stage is demonstrated by creating a BE design of the simple microwave oven from the previous chapter. Finally, in the sixth section, we compare the BE component model with a previous approach to design with BE outlined by Dromey [Dro06b].

## 4.1 Why Design With Behavior Engineering

Since the conception of BE by Dromey [Dro01], a central goal has been to provide a rigorous path from requirements all the way through to a design. This is at odds, however, with current research involving BE, which has primarily focused on requirements analysis and various uses of the BE specification [WD04, WD06, GLYW05, GWY08, GCW07, ZD05a, ZD05b, ZWCD06]. As BE developed into a more formal, rigorous process that can systematically proceed from requirements to a specification, the need to provide a rigorous path from requirements to design with the one approach was questioned. This shift has resulted in BE currently being used most often to develop a throw-away specification, leaving design to be performed using more mainstream approaches. Adding a design stage to BE is a central thrust of this dissertation, so it is important to address the question: why design with BE?

To consider the need for a BE design stage, let us first examine the various options for using a BE specification to support design. Figure 4.1 presents four possible options: (1) throw away the specification; (2) translate to another language; (3) use the specification to generate test cases; and (4) extend BE to include a design stage. In the figure we have also made rankings for each of these options based on the varying levels of difficulty to be delivered as well as the varying levels of potential benefits they would be able to provide.

The first option as to how to use a BE specification to support design consists of developing a BE specification for the sole purpose of finding and correcting defects in the original requirements. The BE specification is then thrown away and the requirements which have been modified to address the defects that were identified are used for a conventional

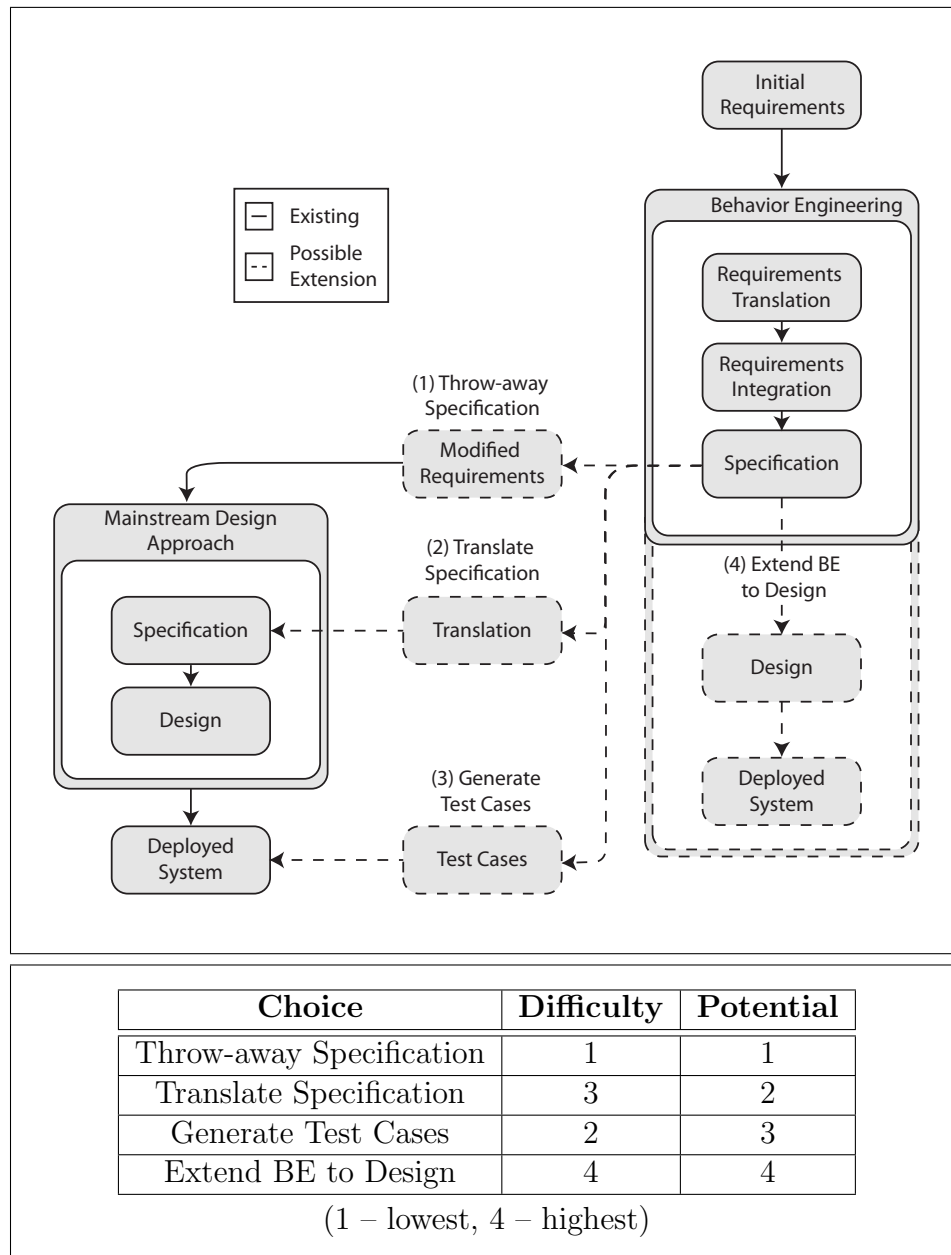


TABLE 4.1: Comparison of Potential Applications of the BE specification

analysis and design approach. This is how a BE specification that has been created to perform requirements analysis is currently used, and while this represents the easiest option to deliver, it has a number of drawbacks. The main drawback is the unnecessary rework created by throwing away the BE specification, because the modified requirements must once again be used to redevelop a specification in a different language with a different approach that supports design. Another drawback of this option is that due to the lack

of a rigorous and scaleable methodology the redeveloped specification is likely to contain introduced defects even with the use of the modified requirements.

The second option enables design from a BE specification by translating the specification into another language associated with a suitable design approach. This choice would then allow any of the many existing design approaches to be used to create a design from the translated specification. Ideally, translating a BE specification into a more popular language for performing design like UML, would enable the choice of many current design approaches and would provide access to a greater choice of tool support. In reality, however, a mismatch between the BML and the languages used by most current design approaches is likely to result in a loss of information during translation and, in the worst case scenario, it may lead to the introduction of defects due to misrepresentations of BT semantics. Translating a BE specification into an OO language would also preclude other potential benefits of BE, such as its component model and its scaleable and rigorous methodology.

The third option for enabling design from a BE specification is a variation of a throw-away BE specification. In this option the modified requirements resulting from creating a BE specification are still used for analysis and design with another approach; however, the BE specification is not thrown away. The BE specification is instead used to generate test cases which in turn are used to validate the system created out of the modified requirements. This approach involves added iteration, because a construct by correction approach is required to locate and resolve defects rather than minimizing or removing these defects with a rigorous methodology. The test cases which are generated could be used to perform integration testing, but unit testing would not be possible because the internal component functionality is not specified in BE.

The final option, advocated by this dissertation, is to extend the BMP of BE to provide a design stage. This option is the most difficult to deliver but also provides the most potential benefit. Just as BE overcomes the requirements-specification gap, a BE design stage can overcome the specification-design gap by extending the scaleable methodology of the BMP. A BE design stage can leverage the integrated view provided by BE specification to evaluate design decisions one at a time, maintaining a minimum local problem space regardless of the size of the global problem space. Together with a component model and an extension



mechanism, a BE design stage can also benefit from the productivity benefits of an MDE framework.

## 4.2 Deploying a BE Model

Adding a design stage to the BMP requires the BE component model to be specified. According to Lau and Wang a component model [LW07], “defines what components are, how they can be constructed, how they can be composed or assembled, and how they can be deployed, as well as, ideally, how we can reason about all these operations on components so that quality certification may be tractable.”. Other research in BE has not required a complete component model because it has been acceptable for components to be represented as variables of an enumerated datatype consisting of the possible states of the component ([GLYW05], p136). A complete BE component model, however, requires the BML to be able to integrate executable components which provide a means of accessing encapsulated computation and passing data.

The BE component we propose is based upon hardware components, in an attempt to replicate the success of hardware components in software. Firstly we will discuss why this approach was taken, then we will describe the BE component model.

### 4.2.1 Replicating Hardware Success with Software Components

Attempting to replicate the success of hardware components in software is not a new concept. In the 1968 NATO Software Engineering conference, McIlroy introduced the concept of software components [Mci68]. He discussed the need to produce software using mass production techniques comparable to those used to develop hardware as well as similar techniques in other engineering disciplines. Brad Cox continued the analogy of software and hardware components by advocating the development of software integrated circuits [Cox86],

“The appeal of all this is the possibility that the software industry might obtain some of the benefits that the silicon chip brought to the hardware industry; the ability of a supplier to deliver a tightly encapsulated unit of functionality

that is specialized for its intended function, yet independent of any particular application.”

Similar views have also been proposed more recently by Bennett [Ben97], “Reuse needs physical components with visible interfaces: chips and pin-outs”. At the core of these attempts is the desire to replicate the success of the hardware component model in software.

The success of the hardware component model can be attributed to attaining two central goals: reuse and scalability. The most prominent proof of the hardware component model achieving these goals is Moore’s Law. In 1965, Moore stated that [Moo65], “The complexity for minimum component costs has increased at a rate of roughly a factor of two per year.”. This essentially predicts a doubling in the number of transistors per integrated circuit (IC) each year <sup>1</sup>. Figure 4.1 shows the exponential growth predicted by Moore fulfilled by Intel ICs [Int09a].

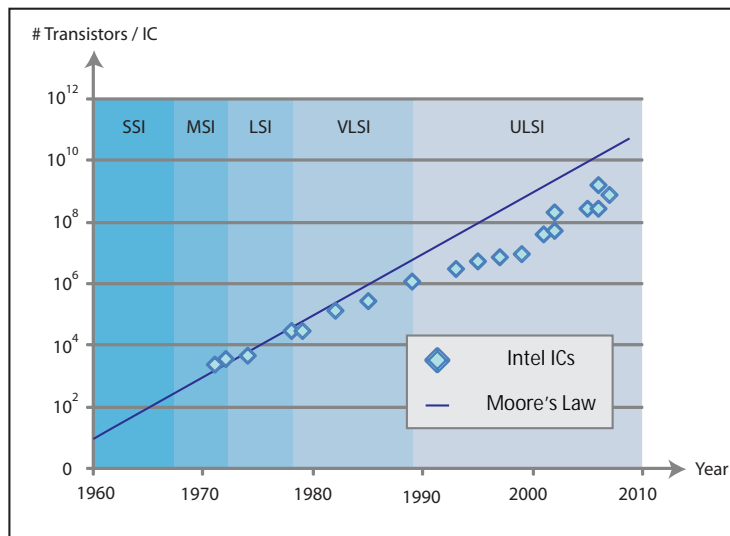


FIGURE 4.1: Exponential Growth of Transistors per Integrated Circuit

Appendix C contains a case study that investigates the characteristics of hardware components that contribute to achieving this reuse and scalability. The case study builds the circuitry of a digital clock using a group of digital logic components known as Transistor-Transistor Logic (TTL). Based on this case study, several characteristics which contribute

<sup>1</sup>Moore later revised this in 1975 to state that the number of transistors on a chip will double approximately every two years. The law is also often misquoted as having a rate that doubles every 18 months [Wik09].

	Characteristic	Hardware Component Model	BE Component Model
(1)	Fully encapsulate computation inside a component	✓	✓
(2)	Component can have multiple configurations	✓	✓
(3)	Separate Computation from Integration	✓	✓
(4)	Active configuration determined by integration	✓	✓
(5)	Redesign encapsulated computation to make a new component	✓	✓
(6)	A component can interface between two incompatible components	✓	✓
(7)	Functionally group components and reuse	✓	✗
(8)	Integrate functional group with additional components	✓	✗

TABLE 4.2: Hardware Components versus BE Components

to achieving reuse and scalability were observed.

Table 4.2 summarises the observed characteristics from the case study. We will now discuss how the BE component model adheres to each of these characteristics. The BT is analogous to the wiring that integrates hardware components, but is instead used to integrate software components. As with the hardware component model, these software components encapsulate computation (1). By separating the encapsulation into a number of computational units, a component can be placed in a number of different configurations (2). The BT captures the flow of control which determines the configuration of each component (4), ensuring a clear separation of integration from computation (3). BE components can easily be redesigned by modifying or adding to the computation encapsulated inside a component to create a new component (5). This is actually easier with a BE component than a hardware component, because software limitations hindering reuse are not as restrictive as the physical limitations of hardware. BE components can also be used to interface between two incompatible components (6). This characteristic, however, is less likely to be used in BE because data passing is not as commonly used as with hardware components. The BE component model currently does not support functionally grouping components for reuse (7-8). In BE, this is known as the systems-of-systems construct, where a system defined in BE is reused as a component in a higher-level system. This is beyond the scope of this dissertation, but is intended to be addressed in future work.

### 4.2.2 The BE Component Model

Figure 4.2 shows the structure of a deployed BE model and the interactions that occur across the two boundaries that are specified by the DBT. A deployed BE model is composed of an environment in which the model is deployed, a system that executes the model, and components that are referenced by the model. The system interacts with the environment across the system-environment boundary and interacts with the components across the system-component boundary. The environment is prohibited from directly interacting with the components and may only interact with components indirectly through the system. The BE component model specifies the structure of the environment, the system, the components and the communication that occurs across the two boundaries of the deployed BE model.

The environment is indirectly specified by a deployed BE model. The DBT defines the external input that is received by the system and the external output that is sent by the system across the system-environment boundary. The environment must have a means

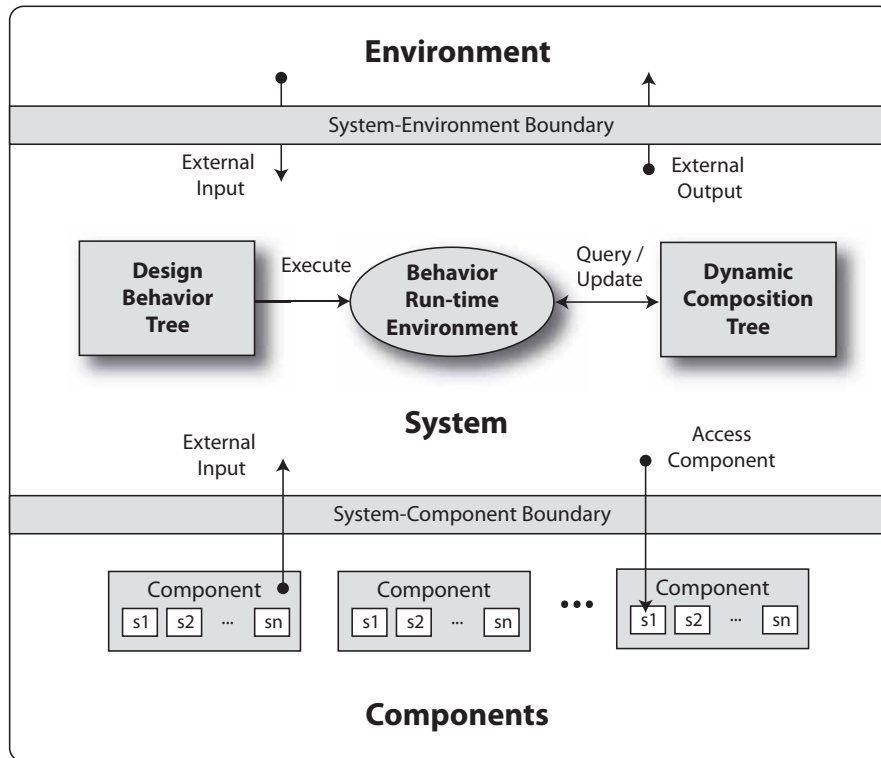


FIGURE 4.2: The structure of a deployed BE system

to send and receive information across the System-Environment boundary. Keeping the BE model platform-independent, however, is the task of middleware that is chosen at deployment to suit the environment the BE model is deployed in.

The system created by deploying a BE model is composed of a design behavior tree (DBT), a dynamic composition tree (DynCT) and the behavior run-time environment (BRE). The design stage of the BMP results in a DBT and a deployment composition tree (DpyCT) which is a precursor for the DynCT. The DpyCT contains the starting context of the system and the starting state of the components. The BRE begins execution using the DBT and the DpyCT. When the BE model executes, the DpyCT becomes the DynCT and stores the current context of the system and the state of each component. Internal inputs and outputs are managed by the BRE and do not pass across the System-Environment or System-Component boundaries. The BRE contains an expression parser which can query the information stored in the DynCT allowing simple expressions to be evaluated. This reduces the communication between the system and the components.

BE components are primarily composed of encapsulated computation which is accessible by the system. Each component is also provided read-only access to attributes it is associated with in the DBT. This allows BE components to describe both passive and active behavior. A BE component exhibits passive behavior by responding to a request by the system to access encapsulated behavior. The BE component executes the encapsulated behavior and returns control to the system. A BE component can also exhibit active behavior by executing behavior independent of the system, however, the only means for active behavior to interact with the system is through an external input to the system. This can also be used to pass data from a BE component that is stored in an attribute in the DBT.

The environment, system and components interact across two separate system boundaries. The system and the environment communicate across the System-Environment boundary and the system and components interact across the System-Component boundary. There are two allowable types of interactions that occur across the System-Environment boundary and two allowable types of interactions that occur across the System-Component boundary. External input can be received by the system and external output can be sent by the system across the system-environment boundary. Similarly, external input can be

received by the system and a system can access the encapsulated computation of a component across the system-component boundary. All four of these interactions are defined in the DBT, and are described by a message passing expression syntax.

The syntax of external input and output messages across the system-environment boundary consists of a message identifier and an optional expression surrounded in brackets:  $message_{identifier}(expression)$ . The  $message_{identifier}$  is used to match a message from the environment to an external input specified by the DBT. The parenthesized  $expression$  is a partial expression used to transmit data values, with one part described in the external input and one part described in the external output. The external output, sent either by the system using an external output node or the environment using middleware, sends a message with a value inside the parenthesis. The external input, received by the system using an external input node, receives the message if it has a matching message identifier and stores the value inside the attribute named inside the parenthesis. For example,  $<< message_{ID1}(2) >>$  and  $>> message_{ID1}(a) <<$  have matching message identifiers ( $message_{ID1}$ ) so the two partial expressions are combined to result in the value 2 being stored in attribute  $a$  ( $a := 2$ ). Attributes are encapsulated inside a component, so the component with attribute  $a$  must also receive the external input.

Across the system-component boundary, the encapsulated computation of a component can be accessed by a BT node in the DBT with either a state realisation, selection or a guard behavior type. If the expression cannot be resolved by the expression parser in the BRE, the expression is sent to the component to be executed. The simplest expression is a state realisation which indicates that the encapsulated behavior denoted by the state name should be executed. For example, the BT node  $C_1[s_1]$  specifies that the system should execute the encapsulated behavior  $s_1$  of component  $C_1$ . Similarly, a selection  $C_1?expression?$  and a guard  $C_1???expression???$  specifies that the component should evaluate the  $expression$  and return the boolean result to the system. If encapsulated behavior requires attributes when being evaluated, this can be specified by indicating the attribute names inside parenthesis,  $C_1[s_1(a_1, a_2, \dots, a_n)]$ . The component can then gain read-only access to the required attributes from the system. Data can be also passed from the component to the system using the message passing expression syntax. The external output

sent from the component is prefixed with the component name, to distinguish external input from a component and external input from the environment. For example, component  $C_{name}$  outputs  $\langle\langle C_{name}.message_{ID1}(2) \rangle\rangle$ , which is received by a component in the system using  $\rangle\rangle C_{name}.message_{ID1}(a)$  and stored in attribute  $a$  belonging to the component linked to the external input node.

### 4.3 An Extension Mechanism for BE

Previously in Chapter 2, we described the need for an extension mechanism for BE to be widely accessible and to gain the productivity benefits of MDE. To demonstrate these characteristics, consider a BE component that is required to compute the factorial of an attribute. Figure 4.3 (a) shows how the factorial can be computed by the state realisation  $Factorial(a)$  which accesses the encapsulated computation of component C. The result is returned as an external input and stored in attribute  $b$ . This approach is reasonable if the factorial is only required to be computed by one component. If there are several components that require to compute the factorial of an attribute, it would be desirable to introduce a factorial operator (e.g.  $!$ ) as shown in Figure 4.3(b). Without an extension mechanism, the expression parser of the BRE would need to be extended to parse and compute the factorial computation. This is not ideal because it would result in either one single BRE which supports several extensions that may not be necessary, or several versions of the BRE that support various extensions.

The BE extension mechanism avoids this problem by defining a mapping for a new expression in terms of the existing expression syntax. This gives the extension clear semantics which are defined by the existing semantics of the language. It also avoids the need to add code defining the computation into multiple components or to make changes to the expression parser of the BRE. Figure 4.3(c) defines the computation of the factorial operator using the existing BE expression syntax. Prior to the execution of the DBT, this mapping can be used to transform an expression with the factorial operator into the equivalent computation defined by a mapping to a BT fragment.

Unfortunately, complex extensions can make mapping the computation defining the

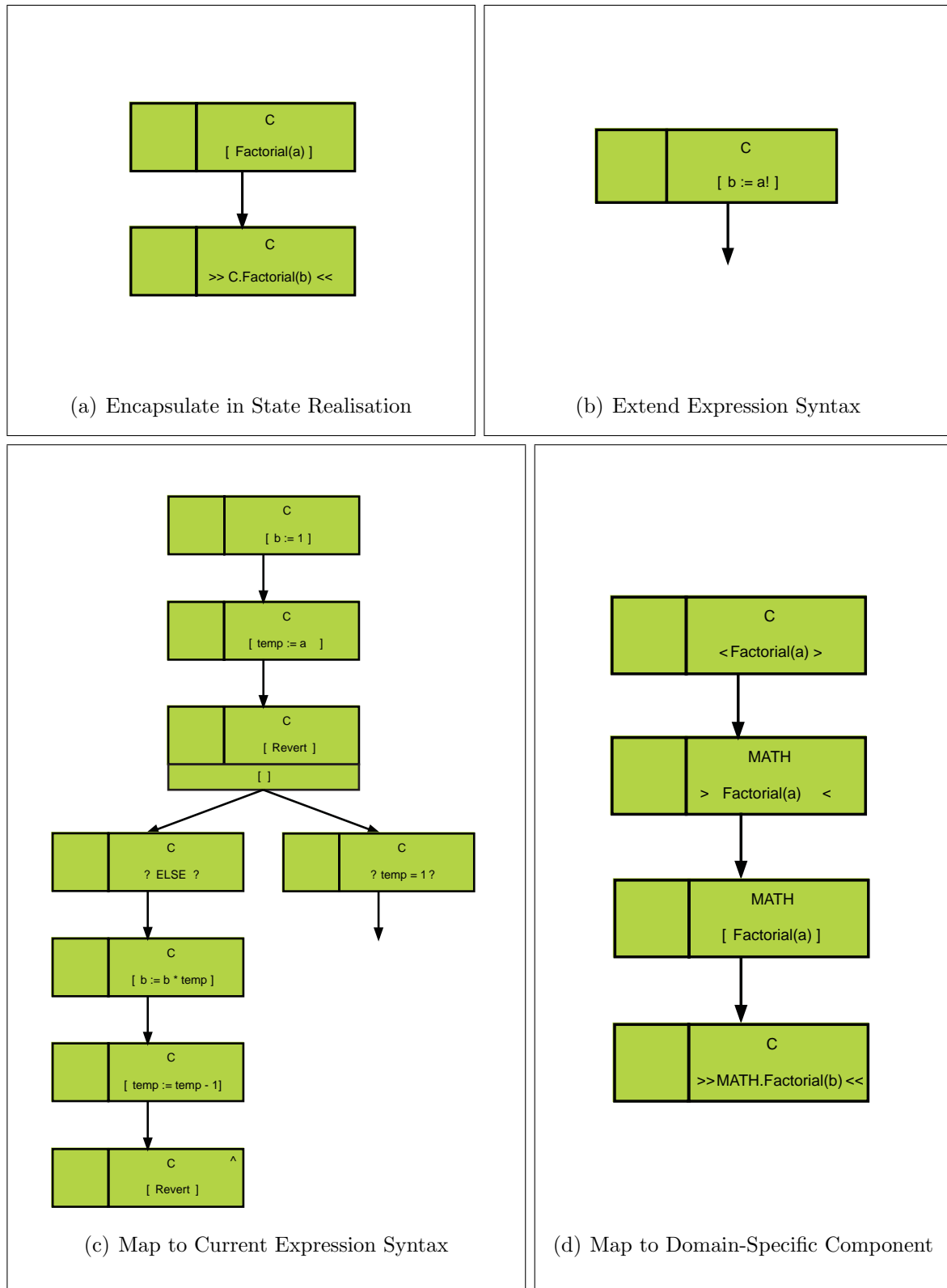


FIGURE 4.3: Extending BT Expression Syntax to include a Factorial Operation



extension into BTs expression syntax prohibitive. To address this, a second BE extension mechanism is used to define domain-specific components which encapsulate a related set of expressions. Figure 4.3 (d) shows how the factorial can be computed using a Math domain-specific component. When the factorial operator is used in an expression in the BT, the DBT is transformed prior to execution to use the Math domain-specific component to compute the factorial. This approach no longer maps the semantics of the extension in terms of the existing BT expression syntax, but instead allows the semantics to be defined by other mainstream programming languages.

## 4.4 The Design Stage of the BMP

The design stage of the BMP involves making design decisions that describe how the system will satisfy the specification defined by the MBT. The task of these design decisions is to resolve the location of the system-environment and system-component boundaries in the specification. By resolving these two boundaries, the DBT resulting from the design stage naturally emerges from the MBT. The DBT can then be designed to be deployed in multiple starting states using a Deployment CT (DpyCT) to describe the initial context of the system.

### 4.4.1 Resolving the System/Environment Boundary

Identifying the location of the system-environment boundary is aided by an important characteristic of the MBT. It is normal for the MBT to model a portion of the environment in which the system will exist, so that interactions between the system and the environment may be specified. The location of the system-environment boundary in the MBT can be resolved by making design decisions that determine which components are located in the environment of the DBT.

A component that forms part of the environment can be identified by two characteristics. Firstly, the system is unable to control the behavior of the component and secondly, the system is unable to determine the state of the component directly. Identifying these characteristics usually requires knowledge of the domain being modeled.

Once it has been established that a component is in the environment, design decisions must be made as to whether the system should be able to control the environmental component or whether it should be able to receive events from the environmental component. Adding this functionality to an environmental component requires an additional component to provide this functionality to the system. Choosing whether an environmental component requires an additional component to interact with the system is a design decision, so whether this is required or not depends on the desired characteristics of the final design. In some cases, however, the MBT can provide indicators of whether an additional component is necessary. If the environmental component contains behavior that other components in the system are dependent upon, then the system most likely requires a sensor to detect the state of the environmental component. If the environmental component is independent of all other behavior in the system, the environmental component can probably be removed from the system.

Environmental components are dealt with differently in the DBT depending on whether the component is composed of: (1) only external input (2) other behavior but not external input (3) both other behavior and external input. In the first case, if the external inputs are required by the system and can be provided by a sensor external to the system, the external inputs are reassigned to the system component. Otherwise, if the external inputs are not required by the system, the nodes of the environmental component are given a deleted traceability status. In the second case, if the system needs to know the state of the environmental component, the system requires the addition of a sensor external to the system. The sensor must then provide external input to the system indicating the current state of the environmental component. This external input can then either be used to maintain the state of the component in the DBT, or it can be used directly to control the related behavior in the DBT. Otherwise, if the states of the environmental component are not required to be known by the system, the nodes of the environmental component are given a deleted traceability status. In the third case, the environmental component may be a sensor with external input that requires software processing. If this is the case, the external input should remain associated with the environmental component in the DBT. On the other hand, if neither the external input or other behavior is required by the system,

then the environmental component can be treated as per the previous two cases.

#### 4.4.2 Resolving the System/Component Boundary

Resolving the location of the system-component boundary involves two tasks. Firstly, the location of system-component boundary in the DBT determines the balance between adding complexity to the BT or to the component implementations. Secondly, while resolving the system-component boundary, the principles of *responsibility-driven design* should be applied to determine if the current behaviors are most suited to the components to which they belong.

The first task in resolving the system-component boundary involves deciding the appropriate granularity of the components. Though the DBT can represent most algorithmic computations that are encapsulated in a BE component, third-generation textual programming languages are more suited to represent this lower-level behavior. Care should be taken, however, to avoid adding integrated behavior into the encapsulated behavior of components which should solely be captured in the DBT.

The second task in resolving the system-component boundary involves determining whether current behaviors are suited to the components they are associated with. This ensures that the components specified by the DBT are more likely to match existing component implementations, or if no matching component implementations exist, they are more likely to be reused in another system.

Another important part of resolving the system-component boundary is to remove relational behavior. Because relational behavior is not yet supported by the BRE, it must be removed from the DBT. How relational behavior is removed depends on the behavior type of the primary behavior. If the primary behavior is an input or an output, then the message should be renamed to capture the relation. If the primary behavior is a state, selection or a guard, then attributes or message passing should be used to capture the relation.

### 4.4.3 The Deployment Composition Tree

We can now demonstrate the design stage by creating a BE design of the simple microwave oven from the previous chapter. When designing the system, the system can often be deployed in several different starting states. The DpyCT is a product of the design stage of the BMP which describes the starting context of the system and the initial states of the components when the system is deployed. The design stage can generate several DpyCTs, each capturing a possible deployment of the system.

The design should have at least one DpyCT, which is composed of the information represented as the initial states in an atomic block at the root of the MBT. Because these initial states are represented in the DpyCT in the design stage, they can be removed from the DBT by assigning them a deleted traceability status. The DBT should also be reviewed to determine if other DpyCTs are required.

## 4.5 Designing the Simple Microwave Oven

The design of the simple microwave oven begins by determining the location of the system-environment boundary. It is clear that the User, the Door and the Button are not components in the Oven system by the following reasoning. The system does not control these components i.e. the system cannot open the Door or push the Button. The system also does not receive inputs directly from these components i.e. the Door does not tell the system it has been opened. Therefore a design decision can be made to remove the User, the Door and the Button components when creating the DBT from the MBT.

This particular design includes the design decision that the opening and the closing of the Door and the pushing of the Button are handled by physical components which relay the information to the microwave oven system. The information from the physical components are received by the DBT as external input. In Chapter 5, a door sensor and a physical button are connected to create interrupts in an embedded system. These interrupts are then used to send the appropriate external input to the BRE.

Figure 4.4 shows the partially completed DBT after the system-environment boundary

is resolved. The following actions were applied to the MBT:

- The User component only consists of events, so these events are reassigned to the system component with a refinement traceability status. The environmental threads of the User component are given a deleted traceability status.
- The Door component has BT nodes with guard and state realisation behavior types. The guards are only needed for the User component, so they can be given a deleted traceability status. The Door component now only consists of states, so all remaining nodes of the Door component are given a deleted traceability status.
- The Button component only consists of states, so all BT nodes with the Button component are given a deleted traceability status.

The DBT is completed by resolving the system-component boundary and the initial states of the microwave oven. The finalised DBT is shown in Figure 4.5. The following actions were taken to finalise the DBT:

- The Oven states OneMinute, ExtraMinute and Stopped are assigned to the Timer component.
- The Oven [Finished] node is given a deleted traceability status and the reversion to Oven [Idle] is used to indicate cooking is finished.
- The TimedOut message is still received by the Oven system component, but is prefixed with ‘Timer.’ to indicate the message originates from the Timer component.
- The initial states at the root of the tree are given a deleted traceability status and are replaced with a DpyCT.
- A DpyCT is created with an OVEN[Open] starting node and the LIGHT[On] and POWER-TUBE[Off] initial component states.
- A DpyCT is created with an OVEN[Idle] starting node and the LIGHT[Off] and POWER-TUBE[Off] initial component states.

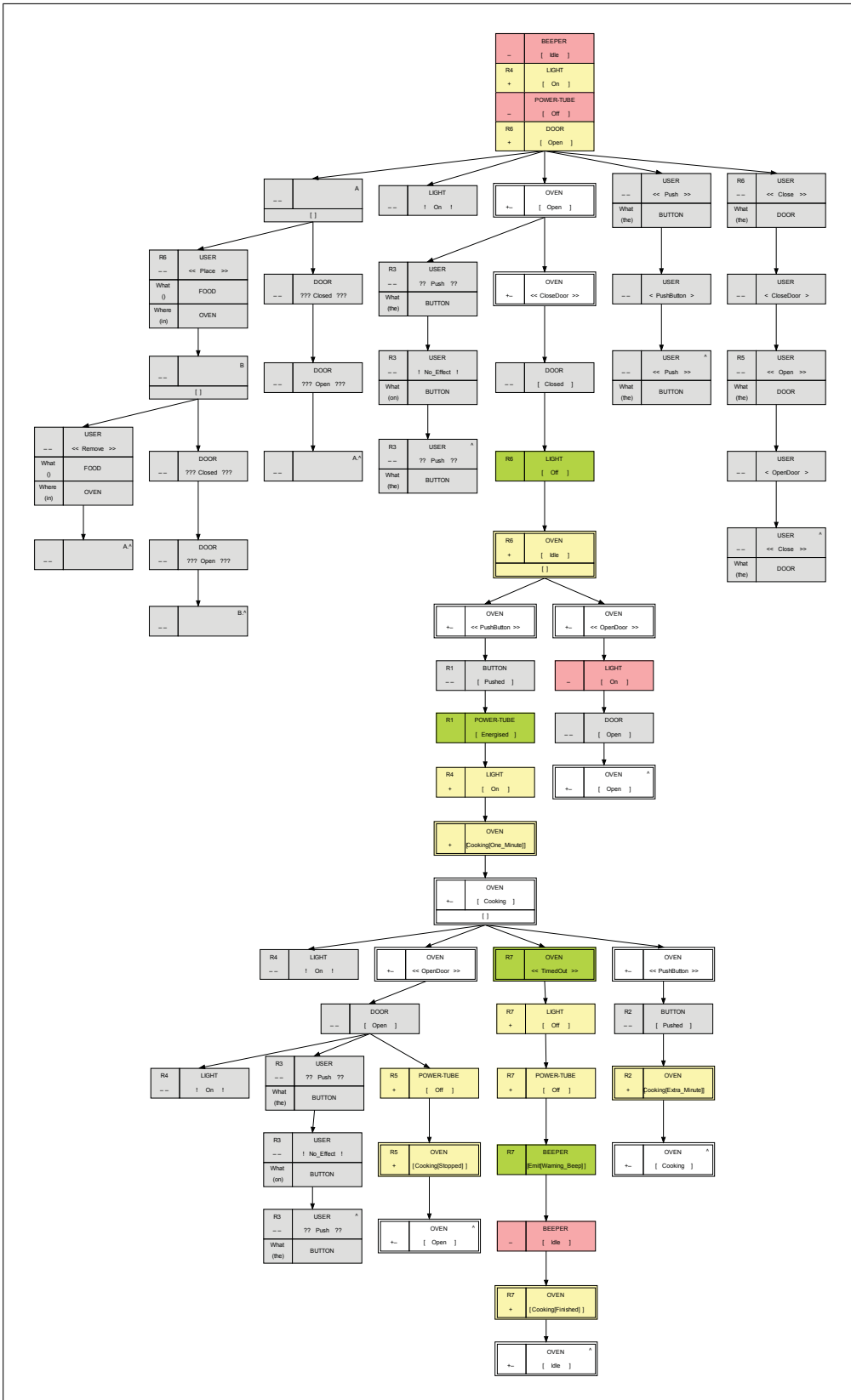


FIGURE 4.4: Partial Design Behavior Tree #1

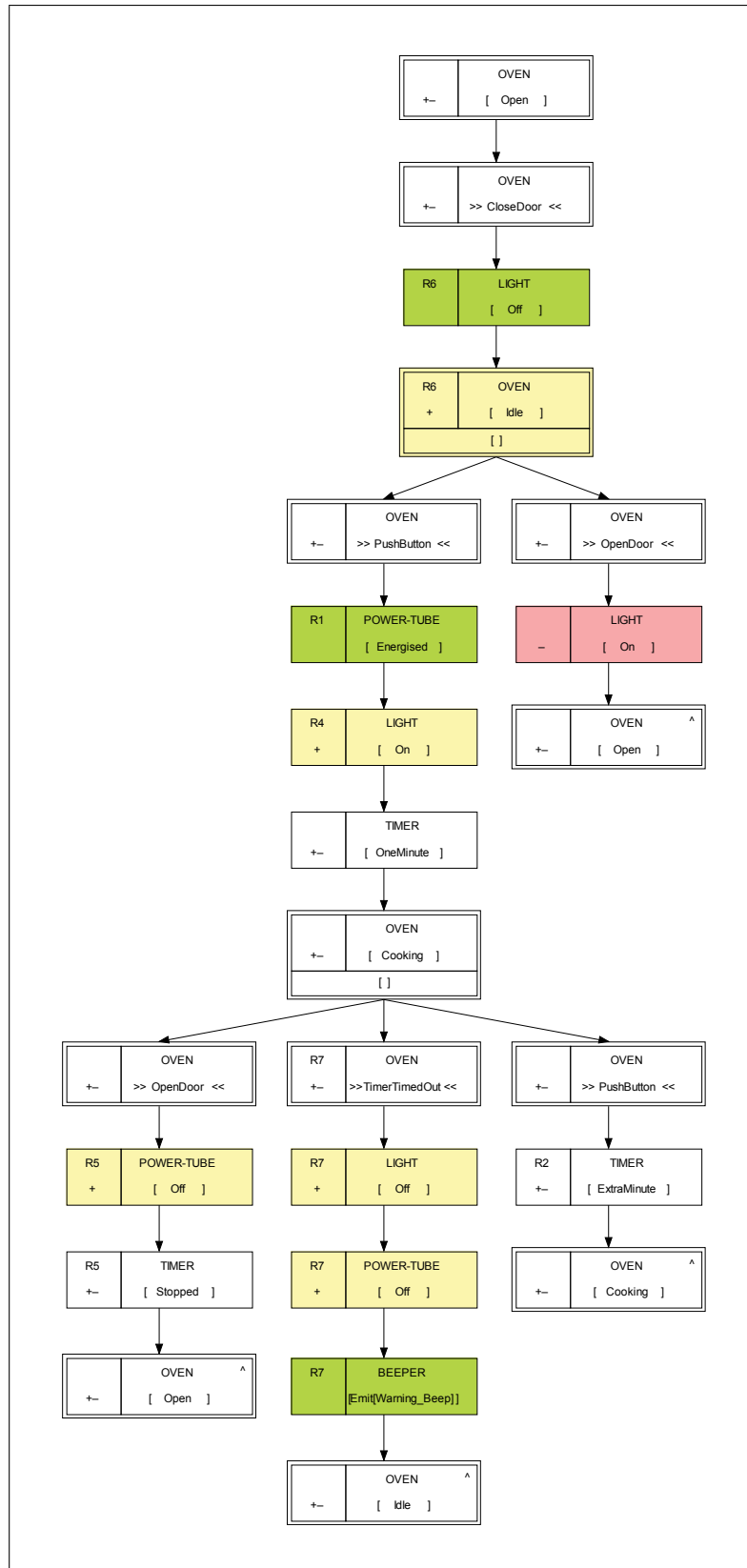


FIGURE 4.5: Design Behavior Tree (with deleted nodes hidden)

## 4.6 A Previous Approach to Design with BE

An approach to design with BE has previously been presented by Dromey [Dro06c]. This approach transforms a DBT into a component architecture using extractable views. The approach used three views extracted from the DBT to create a component architecture: the component interaction network (CIN), the component behavior tree (CBT) and the component interface diagram (CID). The CIN consists of each component represented only once, and interactions between two components are represented by a single line. The CBT is a projection of each of the BT nodes involving a single component whilst preserving the structure of the BT. The CID provides the preconditions and postconditions of a transition of the state of a component.

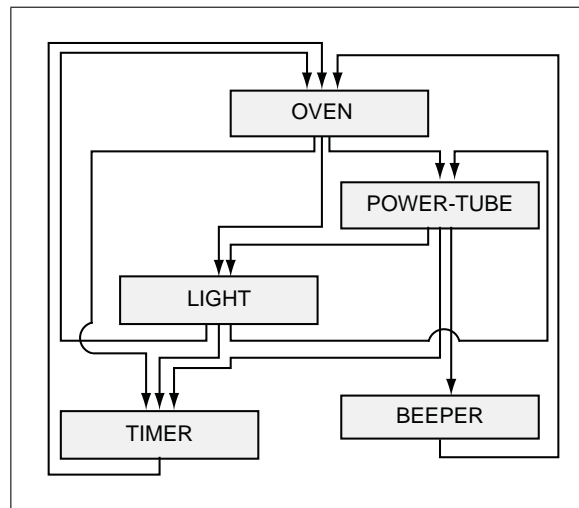


FIGURE 4.6: An Example Component Interaction Network (CIN)

This approach to design begins by projecting the CIN from the causal interactions that occur between components in the DBT. An example CIN of the microwave oven is shown in Figure 4.6. While the CIN documents the interactions between each of the components in the system, it does not describe the individual behavior that needs to be implemented and encapsulated inside each component. This is provided by a CBT, which is a projection of each of the BT nodes involving a single component whilst preserving the structure of the BT. Figure 4.7 shows an example CBT of the Light component. The CID provides the preconditions and postconditions of a transition of the state of a component by combining the



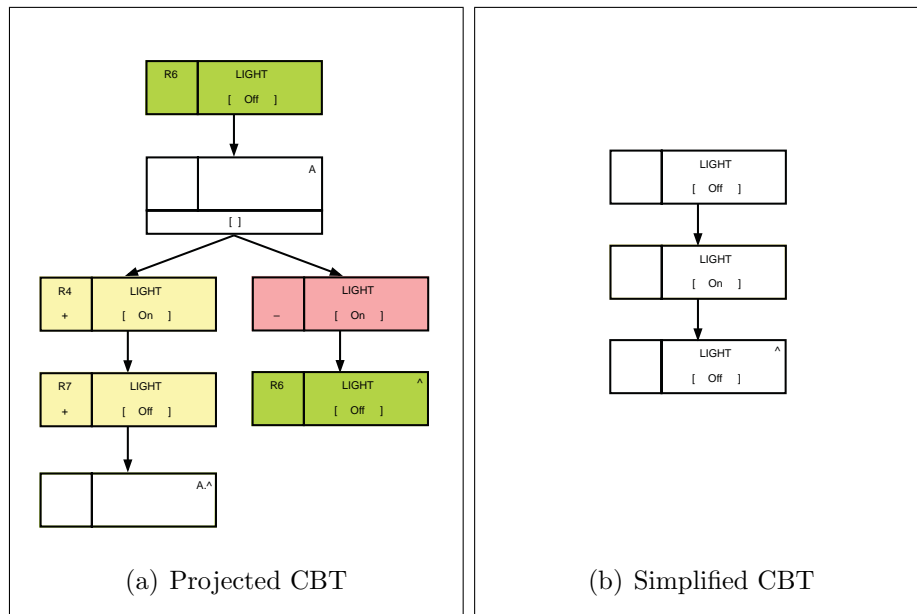


FIGURE 4.7: The Component Behavior Tree (CBT) of the Light Component

relevant information of each component in the CIN with the CBT to describe the interface for each component.

This approach to design by using views extracted from the DBT has two shortcomings. Firstly, this use of the DBT muddles the creation of a specification and the addition of any real design decisions beyond the specification. Without a design stage, the DBT remains in the problem space since it describes what is required of the system but not how to satisfy those requirements. The resulting component architecture therefore still contains environmental components that were captured to specify interactions between the system and the environment. The component architecture also does not have a clear component model that describes how information generated by components could be communicated to other components in the system. These issues are much easier to resolve with a design stage that adds design decisions to create a DBT, rather than with the extracted views of this approach.

Secondly, by extracting a component-based design from the DBT it is implicit that the resulting architecture is decentralised. The component architecture is implemented by connecting the components in a publish-subscribe architecture using the component interfaces described by the CID. If a component in the system transitions state, it publishes



this transition which then triggers transitions of other components. This decentralised approach was defined prior to the formalisation of multi-threaded behavior in BTs which significantly complicates this approach. For example, consider the component interface of the light component in Figure 4.8. The transition from Light [Off] to Light [On] occurs if OpenDoor external input is received by the Oven. It also has to be ensured, however, that the Oven is previously in the Idle state and that the PushButton external input is not received prior to OpenDoor. Once the Light has transitioned to the On state, it must trigger the Oven [Open] reversion, which could effect multiple threads, thereby further increasing the complexity of the component interfaces.

## 4.7 Discussion

Adding a design stage to the BMP allows BE models to be deployable component-based designs. The separation of integration and computation used by BE, however, can be difficult to grasp for developers with an object-oriented background. This is because object-oriented developers are accustomed to designing objects which can call the methods of other objects. In BE, this interaction must be managed in a BT, rather than in the encapsulated computation of the component, to avoid breaking the separation of integration from computation. Future work could involve developing a framework to aid academics and students to design and deploy BE models. The framework should include a simple to use editor for creating BE models and visualising the execution of BTs by the BRE. It should also manage a component repository that allows users to supply BE components.

The design stage presented in this chapter also has some limitations. The first limitation is the design stage of the BMP operates on a monolithic BT that is the result of integrating the requirements of the system. The addition of a means of separating a monolithic BT into several smaller integrated system of systems during the design stage could be beneficial to the approach.

The second limitation of the design stage is that multiple component instances are not supported. Adding support for multiple component instances would require extending the BRE to support dynamically creating and destroying BE components.

A third limitation of the design stage is that relational behavior is not supported. BTs have recently been extended to support an executable version of relational behavior [WCD09]. Adding support for executing relational behavior would require extending the DynCT and the expression parser of the BRE.

Finally, the BE extension mechanism described in this chapter requires more examples demonstrating how extensions are defined and used. This chapter presents a brief example of different means of extending the BT expression syntax. Chapter 7 also has a case study which demonstrates extending the BT expression syntax but this does not demonstrate defining an extension by using a domain-specific component.

The chosen case study of the one minute microwaver is useful to introduce the new design stage of the BMP because of its low complexity. The case study, however, does not demonstrate the design stage being applied to a large system with complex integrated behavior and multi-threaded control. While Chapter 6 does demonstrate the design of a more complex system, in future work it could be useful to design a large system with more complex integrated behavior that truly demonstrates the scalability of the methodology.

## 4.8 Conclusion

Adding a design stage to the BMP gives the BE approach an end-to-end scaleable methodology which can proceed all the way from the original natural language requirements to a deployable component-based design. The new design stage presented in this chapter deviates from a previous approach to design in several key areas:

1. The stages of specification and design have been clearly separated and the design decisions required to proceed from a specification to a design are outlined. A DBT naturally emerges from the MBT by resolving the location of the system-environment and system-component boundaries.
2. A component model for BE has been defined that describes how executable components can be integrated by a deployed BE model. The component model introduces the BRE, which integrates BE components by executing the DBT. The BRE also contains an

expression parser which minimises the communication between the system and its components by automatically evaluating simple expressions.

3. An extension mechanism is described that enables DSLs to be created that use BE as a host GPL. The extension mechanism provides two mechanisms to extend the existing BT expression syntax. Firstly, a new symbol can be defined by providing a mapping of the semantic meaning of the symbol using the existing BT expression syntax. Secondly, a new symbol can be defined by computation encapsulated in a domain-specific component.



## Part II

# Forward Modeling





# 5

## Behavior Engineering of Embedded Systems

The need for a scaleable methodology for design is especially evident in embedded systems. The complexity introduced by the increasing scale of embedded systems is compounded by the low level of abstraction of most programming languages used to create software for embedded systems and the close links between the software and the hardware. Currently, rather than using a scaleable methodology to build a system from its requirements the design of embedded systems is more often an informal process involving intuition and iteration. Intuition is used to make design decisions which are based on an intuitive miraculous leap from the requirements to a design. The basis of these decisions exists in the mind of the developer, and little or no documentation is provided to describe the reasoning process. The resulting design must be executed to verify if it meets the needs of each of the requirements. A cycle of iteration is used to incrementally improve the design until the

system meets all of its requirements.

<b>2003 Survey</b>	<b>%</b>
Canceled Projects	13.1%
Delayed Projects	54%
Meets < 70% Requirements	40.5%
Meets < 50% Requirements	32.8%
<b>2008 Survey</b>	<b>%</b>
#1 Concern: Meeting Schedules	51%
#2 Concern: Debugging Process	38%
#3 Concern: Increased Lines and Complexity of Code	26%
Not using a modeling technique	59%
Using UML	16%
Using 'Hardware C'	17%
Using Simulink / other modeling language	10%

TABLE 5.1: 2003 and 2008 Embedded Systems Development Survey Results

This iterative embedded design development process may seem unrealistic, but it is supported by the survey results shown in Table 5.1. A 2003 survey of embedded systems development [Kra03] involving 45,000 projects found that 40.5% met less than 70% of their requirements<sup>1</sup>. Of these, the system functionality of 32.8% of projects was assessed as failing to meet 50% of their requirements. The failure to meet requirements is a clear indicator that an informal process is being used to proceed from the requirements to a design. The survey also found that if a system failed to meet requirements, respondents indicated it led to schedule delays, minor software rewrites and removal of features. Schedule delays are also indicative of an iterative process in which designs are revisited upon failing to meet requirements. These survey results also highlight that it is common for current development processes to result in schedule delays, as 54% of projects surveyed were found to have been completed behind schedule with an average delay of 3.9 months.

These results highlight the need for a scaleable methodology that provides a clear path from requirements to embedded systems code. This requires a shift from design decisions being made in the developer's mind, to design decisions being developed in conjunction with a modeling language. As a result design decisions become documentable, enabling them to be analyzed and compared against requirements without iteratively deploying and testing

<sup>1</sup>The survey used the term pre-design expectations rather than requirements.

the whole system.

The need for modeling techniques in embedded systems is evidenced by a 2008 embedded systems survey [Nas08], in which respondents listed their highest ranked concerns during the development process. After the highest ranked concern of meeting schedules (51%), the next two concerns were the debugging process (38%) and increased lines and complexity of code (38%). A modeling language that documents design decisions should address these concerns. However, 59% of respondents indicated they were not using a modeling technique in their current project, and usage of a modeling technique had not increased from previous years [Uni06]. If an existing modeling technique exists that addresses the concerns of the developers, and there is sufficient awareness of the benefits of the modeling technique in the community, then it follows that the technique should gain an increasing rate of adoption. This increase in adoption rates of existing modeling techniques, however, does not appear to have occurred in the embedded systems domain.

Three common graphical modeling techniques used in embedded systems are: the Unified Modeling Language (UML), Finite State Machines, and Block Diagrams. In 2008, 16% of embedded systems developers used UML for embedded systems development. This number was previously expected to increase to 25% by 2007 [Mel06], but has now remained stagnant for several years. A number of tools have been developed to support embedded systems development using UML. Bridgepoint provides a tool to implement xtUML models called Bridgepoint Builder [Men09], which consists of a mixture of Class Diagrams, State Chart Diagrams and computation described in an action language. IBM also provides a tool to implement UML models called IBM Rational Statemate [IBM09], originally developed by the now defunct company I-Logix. Another UML based modeling technique is SysML, which adds improvements to UML specifically for systems engineering. Tools such as Enterprise Architect, which is produced by Sparx Systems, can be used to develop embedded systems from SysML diagrams [RM10].

State machine approaches distinguish themselves from more UML-centric approaches by using state machines as the primary notation to capture the design. Proponents of state machines are critical of the role of new UML diagrams such as statecharts [WW04], “Statecharts is a new, interesting model of control systems which is used in UML for

behavior specification and is not too helpful for creating the software.”. Tools supporting state machine development for embedded systems include UML StateWizard [Int09b], IAR visualState [IAR09] and the open-source State Machine Compiler (SMC) [Sou09].

Block Diagrams represent software as a collection of interconnected functional blocks. The application of block-diagram based tools in embedded systems is normally for systems with a focus in control systems or digital signal processing domains. Mathwork’s Simulink [Mat09] provides a graphical block-based tool with a library of predefined blocks. The graphical nature of block-based languages has also been suggested as a modelling technique which removes the developer from the complexities of low-level implementation on embedded systems, particularly with complex issues such as concurrency [Nat09]. The Safety Critical Application Development Environment (SCADE) [Est09] uses a block-diagram approach with the goal of developing safety-critical systems. Block-based languages have also been recognized as a suitable framework for software components in embedded systems [AS04].

None of these approaches, however, address complexity by building a design from out of the requirements using a scalable methodology. Applying BE to design embedded systems provides an end-to-end scalable approach for developing embedded systems from out of their requirements. This provides several benefits. Firstly, embedded systems built using BE are more likely to meet their requirements because the system is built from out of the requirements using a correct-by-construction approach. Secondly, reductions in the iteration required to ensure that requirements are met by the design should make it easier to build a system on schedule. Thirdly, a BE design uses the principle of separation of concerns to describe integration independent from computation. The integrated behavior of the system is represented in the BML, which simplifies the task of debugging the overall behavior of the system. Finally, the BE design also decomposes the computation performed by the embedded system into a component-based architecture with loosely coupled components. Implementing these components involves programming small pieces of encapsulated computation defined by the BE design. Because these small pieces are separated from the integrated behavior of the system, the size and complexity of manually created code is greatly reduced.

Figure 5.1 continues on from the previous chapter by describing the workflow for deploying a BE design as an embedded system. The process begins by entering the DBT

and DCT / DpyCT into the BE EMF Editor. The XMI model is then transformed using model-to-model (M2M) and model-to-text (M2T) transformations which results in C source code. The C code is executed using the embedded behavior run-time environment (eBRE), which consists of two core components: The Process Control Model and the BE Component Model. Each of these stages will now be discussed in detail.

## 5.1 The BE EMF Editor

The eBRE requires a means of capturing and storing BE models. The Eclipse Modeling Project provides the Eclipse Modeling Framework to support the creation of metamodels describing the format of a model. These metamodels can then be used to generate a customisable editor.

A metamodel for BTs has been described previously in the literature [GPHSD05]. This metamodel is now outdated as a result of changes that have since been made to BTs and the BML. The BT language is now defined by a formal semantics, requiring elements such as alternative branching and node operators which are not included in this metamodel. Composition Trees have also since been included in the BML and have become an integral part of BE.

Figure 5.2 shows the metamodel used to describe BE models for the eBRE. The metamodel consists of a root element (*BEModel*) which allows the user to name the model

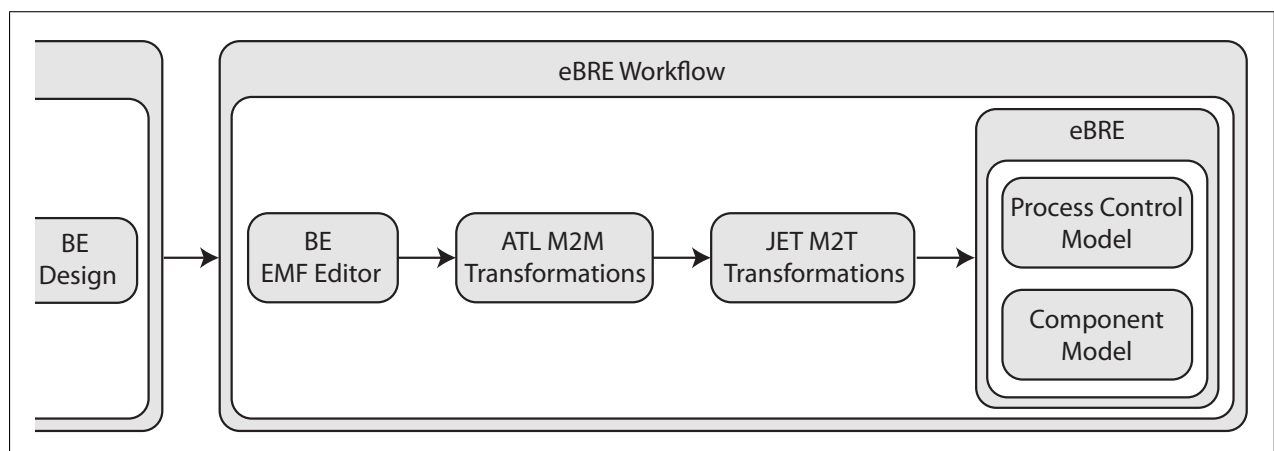


FIGURE 5.1: The workflow of the embedded Behavior Runtime Environment

(*name*). The BEModel contains a deployment composition tree (*dpyCT*) and a design behavior tree (*dbt*).

The BT structure (*BehaviorTree*) is defined in the metamodel using a recursive structure of nodes (*Node*) and edges (*Edge*). The BT begins with a single root node (*rootNode*) that can have multiple edges (*edge*). Each edge contains a child node (*childNode*), which again can have multiple edges and so on.

The information that is captured in a BT node is determined by its type. In the BE metamodel, BT nodes can be classified as either an empty node (*EmptyNode*), a standard node (*StandardNode*), a node with a label (*LabelNode*), a node with an operator (*OperatorNode*), or a node with an operator and a label (*OperatorWithLabelNode*). An empty node contains only a label, which is used to match against the label of a node with an operator and a label. A standard node contains a link to the node's component (*Component*) described in the CT (*CompositionTree*), as well as the behavior (*behavior*), behavior type (*behaviorType*), and traceability information of the node (*traceabilityLink* and *TraceabilityStatus*). The behavior type of a node can be state realisation (*StateRealisation*), selection (*Selection*), guard (*Guard*), internal output (*InternalOutput*), internal input (*InternalInput*), external output (*ExternalOutput*) or external input (*ExternalInput*). A node with a label is a specialisation of a standard node with a label added (*label*). A node with an operator is a specialisation of a standard node with the addition of an operator (*operator*). The operator may be either a reference (*Reference*), a reversion (*Reversion*), branch-kill (*BranchKill*), synchronisation (*Synchronise*), Conjunction (*Conjunction*), disjunction (*Disjunction*), or Exclusive OR (*ExclusiveOR*). These two specialisations can also be combined to have a node with both an operator and a label.

BT edges contain information on the composition (*composition*) and branching (*branch*) of the edge. The composition can be either atomic (*Atomic*) or sequential (*Sequential*). The branching can be either parallel (*Parallel*) or alternative (*Alternative*). The BT also contains special edges (*SpecialEdge*) that are added during M2M transformations. Special edges have a type (*type*) and a destination (*destination*) where the index of the matching node is stored. Special edge types which include reference, reversion, branchkill and synchronisation are processed by transforming the structure of the Behavior Tree.

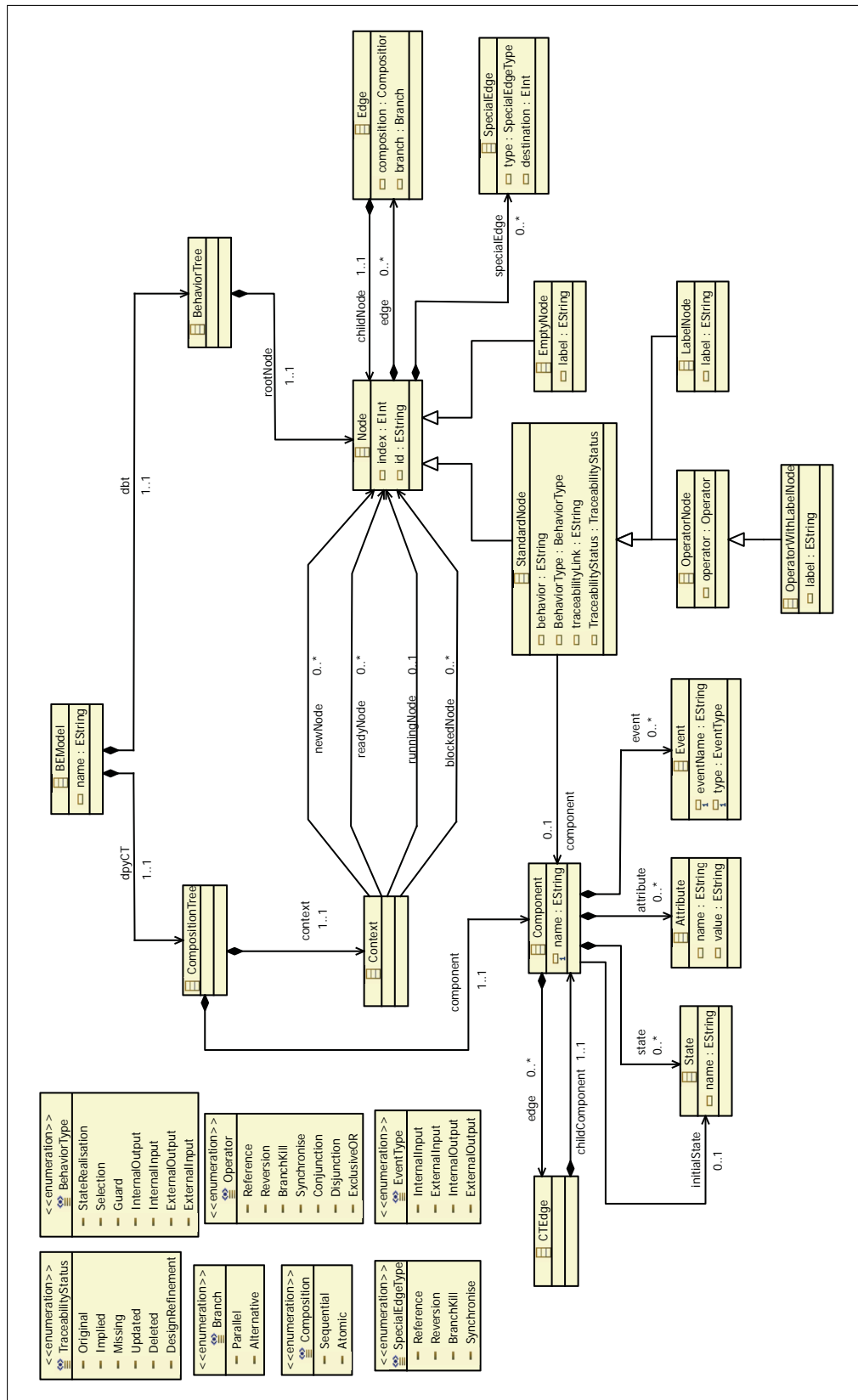


FIGURE 5.2: The Behavior Engineering Metamodel

The CT in the metamodel contains a mixture of the information from the DpyCT and the DCT. It is composed of a context (*Context*) and a hierarchy of components (*Component*) connected by a CT edge (*CTEdge*). The context describes the starting state of nodes in the Behavior Tree when the BE model is deployed. Nodes can start in the new state (*NewNode*), ready state (*ReadyNode*), running state (*RunningNode*) or blocked state (*BlockedNode*). The state of nodes in the context is described in more detail in the process control model (See Section 5.4).

Each component can have a starting state (*initialState*) that initialises the component at system deployment. Each component can also have multiple states (*State*), events (*Event*) and attributes (*Attribute*). However for the eBRE, attributes are not supported in order to minimize the eBRE's footprint.

The metamodel is used to generate the BE EMF Editor. The editor is customised to display the text for BT nodes and edges and to use icons that quickly node traceability status and edge branching and composition. Figure 5.3 shows the Behavior Tree and the Composition Tree of the Microwave Oven in the BE EMF Editor.

## 5.2 Model-to-Model Transformations

Model-to-Model (M2M) transformations are used to minimise the notation that is defined in the executable BE model to the core elements of the BT notation. This has two benefits. Firstly, it allows modellers to use 'syntactic sugar' which is additional notation that simplifies the description of some behavior that could already be defined in a more complex fashion with existing notation. Secondly, it simplifies the execution of BTs by the eBRE.

Currently the following M2M transformations are used by the eBRE: Index and ID, Conjunction, Disjunction, Exclusive OR, Reversion, Reference, Branch-Kill and Synchronisation. These M2M transformations are defined using an extension of an approach for defining transformations of Behavior Trees presented by Grunske et al. [GWY08]. Grunske et al.'s approach defined the abstract semantics of Behavior Trees by creating a graph grammar that describes well-formed BTs. The created graph grammar used advanced graph transformation techniques such as conditional, structure-generic and type-generic graph transformation





rules. The transformation rules that formed the graph grammar were used as a formal specification to develop a model-to-text translator which converts Behavior Trees to SAL code. This allowed model-checking of the transformed BTs to be performed using the SAL toolkit [dMOR<sup>+</sup>04].

The extended version of the BT transformation notation, the BT M2M transformation language, is designed to allow new transformation rules to be easily added if it is necessary to add new notation. The BT M2M transformation language differs from Grunske et al.'s approach in three key areas: (1) It defines partially refining transformations by assuming well-formed input rather than providing a complete set of transformation rules; (2) The order in which transformation rules are applied is prioritised to resolve conflicts, rather than as previously by adding constraints to the preconditions of the transformation rules; and (3) The description of transformation rules is enhanced to be able to describe edges independently and to describe groups of nodes. The rest of this section will introduce the BT M2M transformation language and describe how transformations defined in the BT M2M transformation language are implemented using the ATLAS transformation language (ATL).

### 5.2.1 The BT M2M Transformation Language

Transformations are defined by the BT M2M transformation language using a pattern matching notation which consists of both a graphical and textual matching notation. The structure of the BT pattern is matched graphically, whilst the details of specific nodes are matched textually.

The textual matching notation uses terms to define specific details of a BT node. These terms can match several properties of a node including a unique node identifier ( $N_x$ ), the component name (C), the behavior (b), the behavior type (bt), an operator (op) and a label (l). The terms and their relation to the BT node is shown in Figure 5.4.

The node identifier is a unique term given to each node which links the details defined in the textual matching notation with the structure of the pattern defined in the graphical matching information. The node identifier consists of an upper-case letter and a subscript number or letter. This enables both unique nodes ( $N_0$ ) and arbitrary nodes ( $N_{m-1}$ ) to be

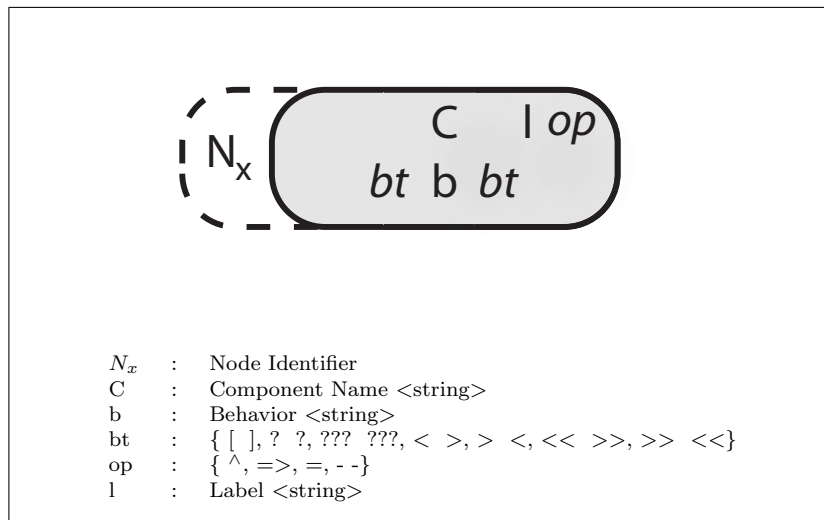


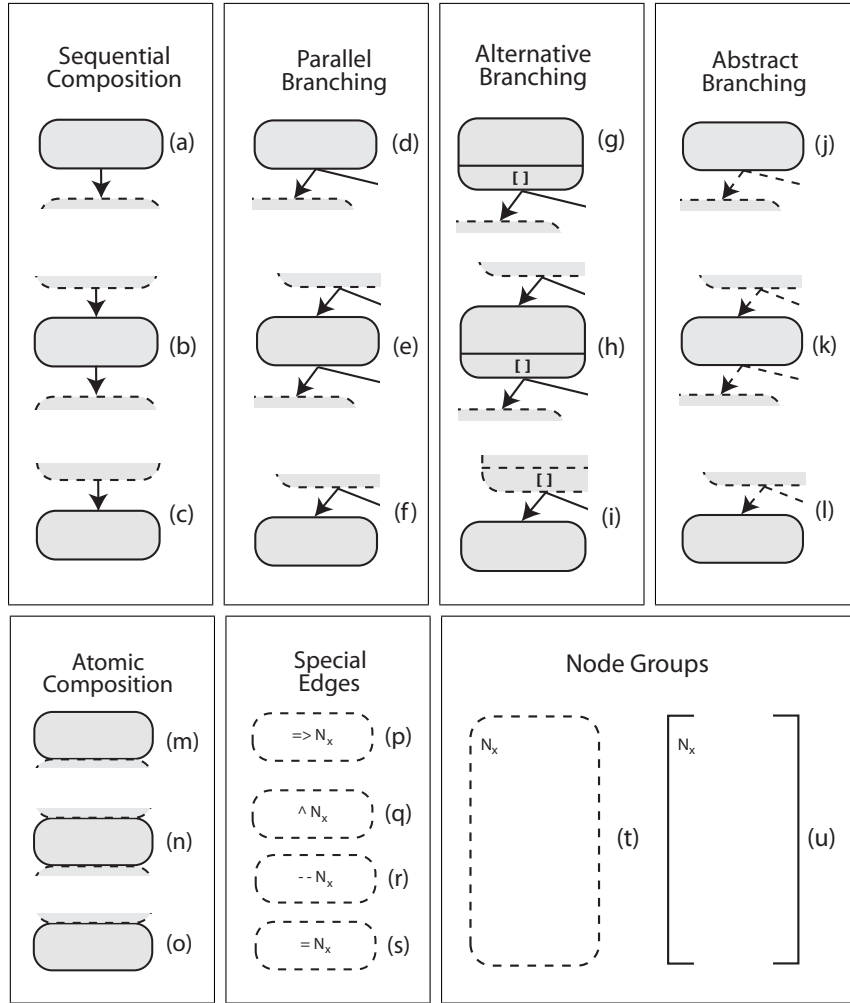
FIGURE 5.4: Textual Terms for Matching BT Nodes

described. These can be combined to form groups of nodes using a colon,  $N_0 : N_{m-1}$ . The number of nodes can be defined using the dot operator, e.g.  $N.m = 7$ . The upper-case letter of node identifiers is selected using a naming convention that a new upper-case letter is used for each group of related nodes.

Other node properties are matched using the node identifier and the textual term in combination with a dot operator. For example, the follow notation matches the component name of BT node  $N_0$ ,  $N_0.C = 'Light'$ . The terms of a group of nodes can also be matched using brackets to enclose the group,  $\{N_0 : N_{m-1}\}.bt = '['$

Figure 5.5 describes the notation used to graphically match the structure of a BT pattern. The graphical notation can be separated into four groups: composition, branching, special edges and node groups. Patterns are formed by describing the composition and branching of the edge that is used to link together the BT nodes. BT edges can be used to match the parent, child or node contained in a group of similar nodes depending on how the edges are connected to the node. BT edges can be matched using either sequential composition, atomic composition, parallel branching, alternative branching or abstract branching. Abstract branching is a special construct used by the BT M2M transformation language to define an edge that can match both parallel and alternative branching.

Special edges are added during transformations to simplify the execution of BTs. Special edges simplify execution by pre-processing operator nodes to locate the target node and by



(a) Parent Node of Sequential Composition; (b) Middle Node of Sequential Composition; (c) Child Node of Sequential Composition; (d) Parent Node of Parallel Branch; (e) Middle Node of Parallel Branch; (f) Child Node of Parallel Branch; (g) Parent Node of Alternative Branch; (h) Middle Node of Alternative Branch; (i) Child Node of Alternative Branch; (j) Parent Node of Abstract Branch; (k) Middle Node of Abstract Block; (l) Child Node of Abstract Branch; (m) First Node of Atomic Block; (n) A Middle Node of Atomic Block; (o) Last Node of Atomic Block; (p) Reference Edge; (q) Reversion Edge; (r) Branch Kill Edge; (s) Synchronisation Edge; (t) Required Group of Nodes; and (u) Optional Group of Nodes;

FIGURE 5.5: Graphical Notation for Matching BT Nodes

then creating a static link using its node identifier. This static link can then be utilised during execution, removing the need to locate the target node each time a node operator is executed. Special edges exist for transforming nodes with reference, reversion, branch-kill and synchronisation operators.

Two means exist for defining a group of nodes in the graphical notation. Firstly, a group of nodes can be captured by surrounding the nodes by a node group box. If the group of nodes is an optional part of the pattern, then the box is two brackets to the left and right

of the group of nodes. If the group of nodes is required to be present, then a box with a continuous dotted line is used to surround the nodes of interest. If details of the nodes need to be described in the textual notation, then a node identifier can be associated with the group of nodes. Secondly, a group of nodes can be captured by using the node identifier to indicate a group, e.g.  $N_0 : N_{m-1}$ . The group of nodes that is described depends on the notation to which it is applied as shown in Figure 5.6. If a node group is defined in a node identifier of a child node, then it represents a horizontal group of sibling nodes of the child node type. If a node group is defined in a node identifier of a middle node, then it represents a vertical group of node descendants of the child node type.

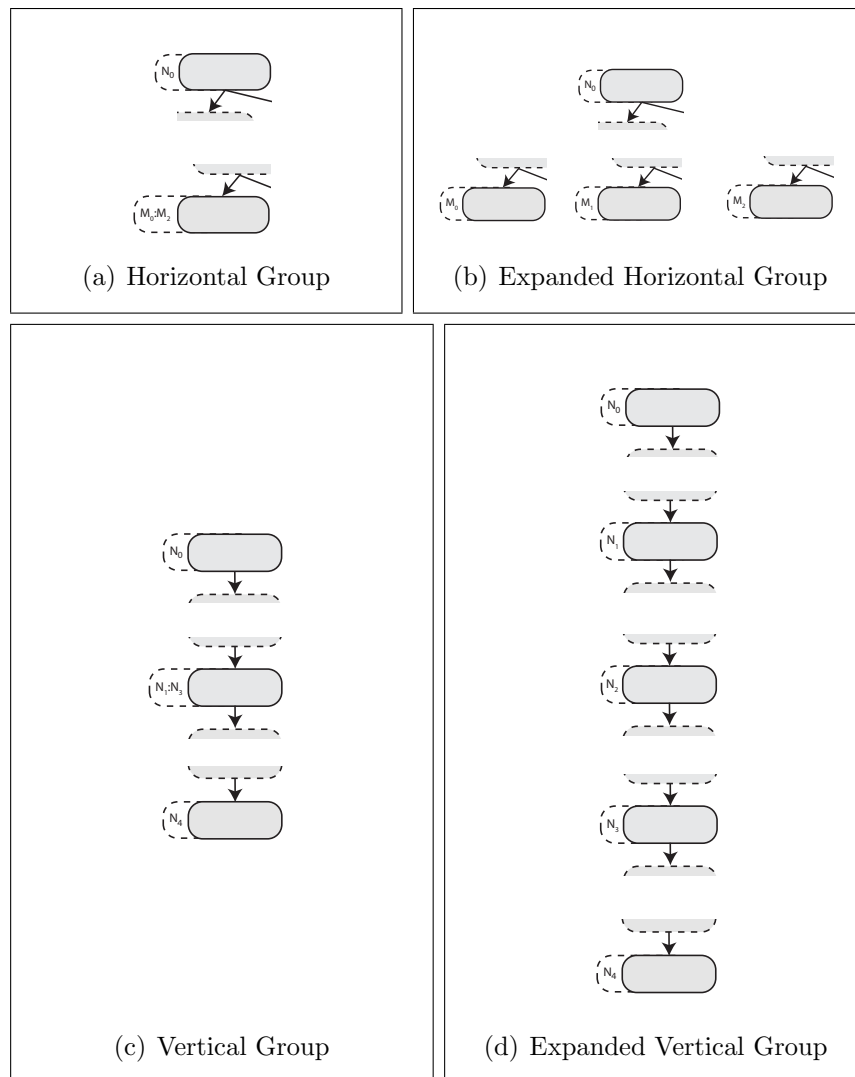


FIGURE 5.6: Matching Groups of Nodes

Transformation rules are applied by using the graphical and textual notation to define a *from pattern* and a *to pattern*. The *from pattern* describes the pattern that must be matched by the BT being transformed for the rule to be applied. The *to pattern* shows the changes that are made when the transformation rule is applied. The transformation rules are partially refining transformations, so details that remain the same between the from and to patterns are implicitly copied. Also, any details that are not defined in the from pattern will match anything i.e. any group of nodes that occur before or after the pattern. We will now discuss the Synchronisation transformation rule to demonstrate the BT M2M transformation language. Appendix D contains the complete set of BT M2M transformations defined in the BT M2M transformation language.

### The Synchronisation Transformation Rule

The transformation rules of all the operators with special edges (reversion, reference, branch-kill) except synchronisation consist of matching a node with an operator and its target and substituting a special edge. The Synchronisation transformation rule shown in Figure 5.7 is more complex due to multiple target nodes that must be matched. The graphical portion of the from pattern matches against multiple groups of nodes  $Q_0 : Q_{m-1}$ , each of which contains a node  $O_{m-1}$  with a synchronisation operator. The textual portion of the from pattern adds the further restriction that the details of the  $O_{m-1}$  node in each  $Q$  group of nodes must be the same.

The to pattern describes the addition of synchronisation special edges to the  $O_{m-1}$  node of the  $Q_0$  group of the nodes, which link to each of the  $O_{m-1}$  nodes in the  $Q_1 : Q_{m-1}$  groups. This allows the nodes that are participating in a synchronisation event to be located quickly during execution.

### 5.2.2 Implementing M2M Transformations in ATL

BT M2M transformations are implemented using the ATLAS Transformation Language (ATL) developed by INRIA, which now forms part of the Eclipse M2M project. ATL is used to describe a mapping between a source metamodel and a target metamodel to enable

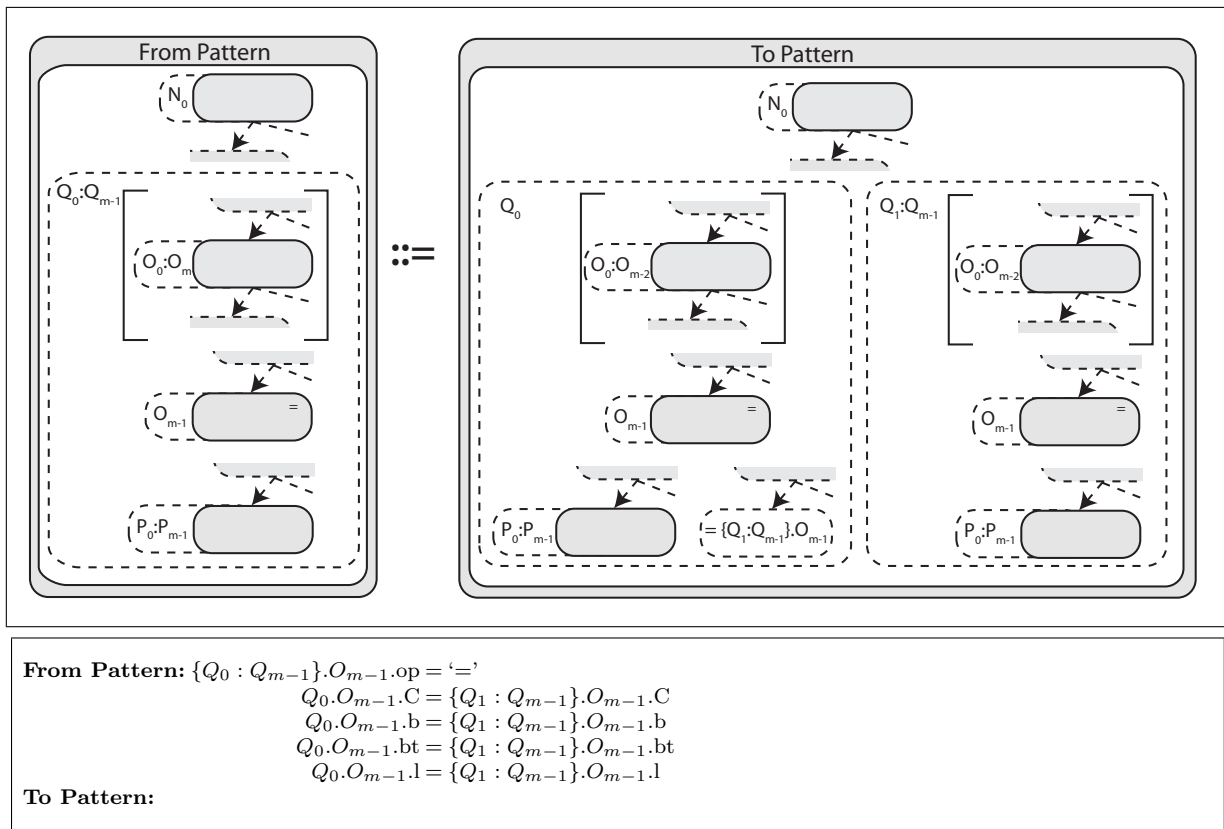


FIGURE 5.7: Synchronisation Transformation Rule

the elements of a source model to be transformed into the target model. In the case of BE M2M transformations, the source and target metamodel are the same, so transformations refine elements of the source model into the desired configuration of the target model.

Though ATL supports a refining transformation mode, it is not currently implemented in the engine for transforming Eclipse ecore models. In order to circumvent this, a rule is required for each element to copy it into the target model, even if it is not being altered. To support the development of ATL transformations, a library of BE matched rules called rules and helpers was developed. Matched rules are used to copy the contents of the source model into the target model without change. Called rules are used to allow new model elements to be added to the target model, such as various types of BT nodes and edges. Helpers are used to perform tasks such as finding the destination of a node operator.

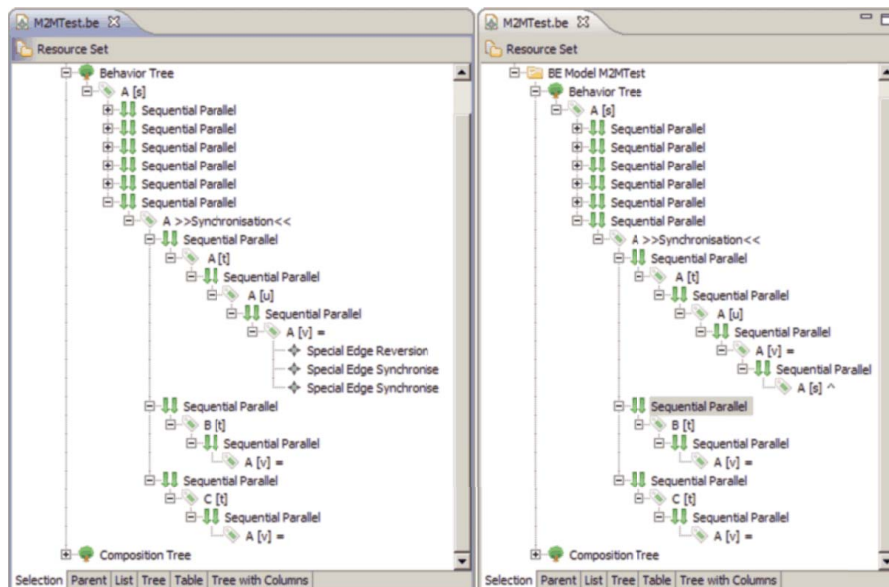
Figure 5.8 shows the synchronisation M2M transformation implemented in ATL and the result of applying the M2M transformation to a BT.

```

-- Synchronisation Rule
rule Synchronisation {
  from s : IN!OperatorNode(
    if (s.colIsTypeOf(IN!OperatorNode))
    then
      if (s.operator = #Synchronise)
      then
        true
      else
        false
      endif
    else
      false
    endif
  )
  using {
    found : Boolean = false;
  }
  to t : OUT!OperatorNode(
    index <- s.index,
    id <- s.id,
    component <- s.component,
    behavior <- s.behavior,
    BehaviorType <- s.BehaviorType,
    traceabilityLink <- s.traceabilityLink,
    TraceabilityStatus <- s.TraceabilityStatus,
    operator <- s.operator,
    edge <- s.edge->flatten(),
    specialEdge <- s.specialEdge->flatten()
  )
  do {
    found <- let synchNodes : Sequence(Integer) = thisModule.synchGroup in
    synchNodes->iterate(n; found2 : Boolean = false |
    if (n = s.index) then true else if found2 = true then true else false endif endif);
    if (found = false){
      thisModule.synchGroup <- thisModule.synchGroup->union(s.getDestination());
      for(d in s.getDestination()){
        if (d = t.index){
          else{
            t.specialEdge <- thisModule.addSpecialEdge(d, #Synchronise);
          }
        }
      }
    }
  }
}

```

(a) Synchronisation rule in ATL



(b) Comparison of before and after synchronisation rule is applied

FIGURE 5.8: Implementing the Synchronisation Rule



## 5.3 Model-to-Text Transformations

After the Model-to-Model transformations have been completed, Model-to-Text (M2T) transformations are applied to the BE model to convert it into source code. M2T is the MDE term for what is otherwise known as code generation. M2T transformations convert the BT and the CT into C header files which are then used by the eBRE. The M2T transformations are written using java emitter templates (JET) which form part of the M2T eclipse project.

Figure 5.9 shows part of the JET template for generating part of the BT header file and the result of generating the BehaviorTree.h C header file for the One Minute Microwaver case study. We will now discuss the representation of the BT and CT in the C header files used in the eBRE.

### 5.3.1 M2T transformation of the Behavior Tree

The C language representation of the BT of the BE model is represented using an interleaved double-linked list of nodes and edges as is shown in Figure 5.10.

The node elements of the double-linked list contain the details of the BT node and references to the edges that link it to its parent node and child nodes. The ID of the node element is a unique identifier assigned to each BT node using a singleton series. The ID enables the quick determination of whether one node is the ancestor of another. The Component term is a pointer to the Component element from the M2T transformation of the Composition Tree. The Behavior term stores an index of an enumeration of the possible states of the component. The Behavior type stores an index which represents the node's behavior type, selected from an enumeration of the possible behavior types. The Parent Edge contains an index which is used to link to the node's parent edge in the edge array. The Child Edges are similar except as a BT node can have multiple child edges, an array of indices is used. To allow the array to be navigated, the number of child edges is also stored.

The edge elements of the double-linked list contains the composition and branching of the BT edge selected from enumerations of the possible types. Each edge element also contains two indices to link to its parent node and child node.



```

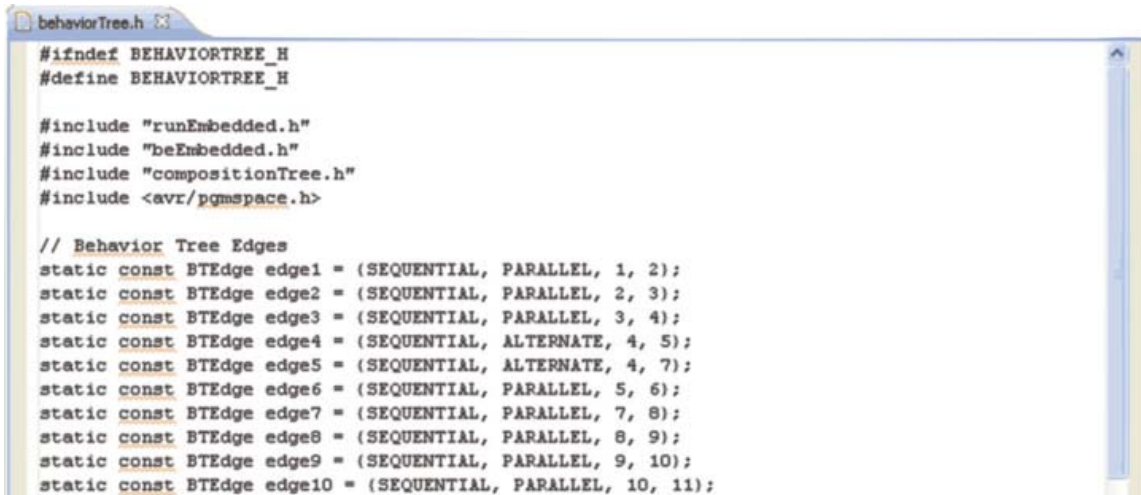
behaviorTree.jet
<!-- Embedded Behavior Run-time Environment (eBRE) -->
<!-- File: behaviorTree.jet -->
<!-- Date: 12 December 2008 -->
<!-- Notes: Run M2M Transformations First -->
#ifndef BEHAVIORTREE_H
#define BEHAVIORTREE_H

#include "runEmbedded.h"
#include "beEmbedded.h"
#include "compositionTree.h"
#include <avr/pgmspace.h>

<!-- $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$ -->
<!-- Describe Edges -->
<!-- $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$ -->
// Behavior Tree Edges
<!-- ITERATE OVER EDGES -->
<c:setVariable var="edgeIndex" select="1"/><!-- EDGE # --><!-- PARALLEL / ALTERNATE --><!-->
</c:if><c:if test="$edge/childNode">static const BTEdge edge<c:get select="$edgeIndex"/> = {<c:
</c:if><c:set select="$edge" name="edgeIndex"><c:get select="$edgeIndex"/></c:set><c:setVariab.
<!-- EDGE ARRAY -->
<c:setVariable var="first" select="'true'"/>
static const PGM_VOID_P edge[] = {<c:iterate select="//rootNode/edge | //childNode/edge | //ro

```

(a) JET Template for the Behavior Tree



```

behaviorTree.h
#ifndef BEHAVIORTREE_H
#define BEHAVIORTREE_H

#include "runEmbedded.h"
#include "beEmbedded.h"
#include "compositionTree.h"
#include <avr/pgmspace.h>

// Behavior Tree Edges
static const BTEdge edge1 = {SEQUENTIAL, PARALLEL, 1, 2};
static const BTEdge edge2 = {SEQUENTIAL, PARALLEL, 2, 3};
static const BTEdge edge3 = {SEQUENTIAL, PARALLEL, 3, 4};
static const BTEdge edge4 = {SEQUENTIAL, ALTERNATE, 4, 5};
static const BTEdge edge5 = {SEQUENTIAL, ALTERNATE, 4, 7};
static const BTEdge edge6 = {SEQUENTIAL, PARALLEL, 5, 6};
static const BTEdge edge7 = {SEQUENTIAL, PARALLEL, 7, 8};
static const BTEdge edge8 = {SEQUENTIAL, PARALLEL, 8, 9};
static const BTEdge edge9 = {SEQUENTIAL, PARALLEL, 9, 10};
static const BTEdge edge10 = {SEQUENTIAL, PARALLEL, 10, 11};

```

(b) Generated C Header File of the Behavior Tree

FIGURE 5.9: Generating Code with Java Emitter Templates (JET)

### 5.3.2 M2T transformation of the Composition Tree

The C code representation of the CT shown in Figure 5.11 consists of the components that form the composition tree and the system context. These elements have the dual purpose of representing the deployment CT (DpyCT) and the dynamic CT (DynCT). The current

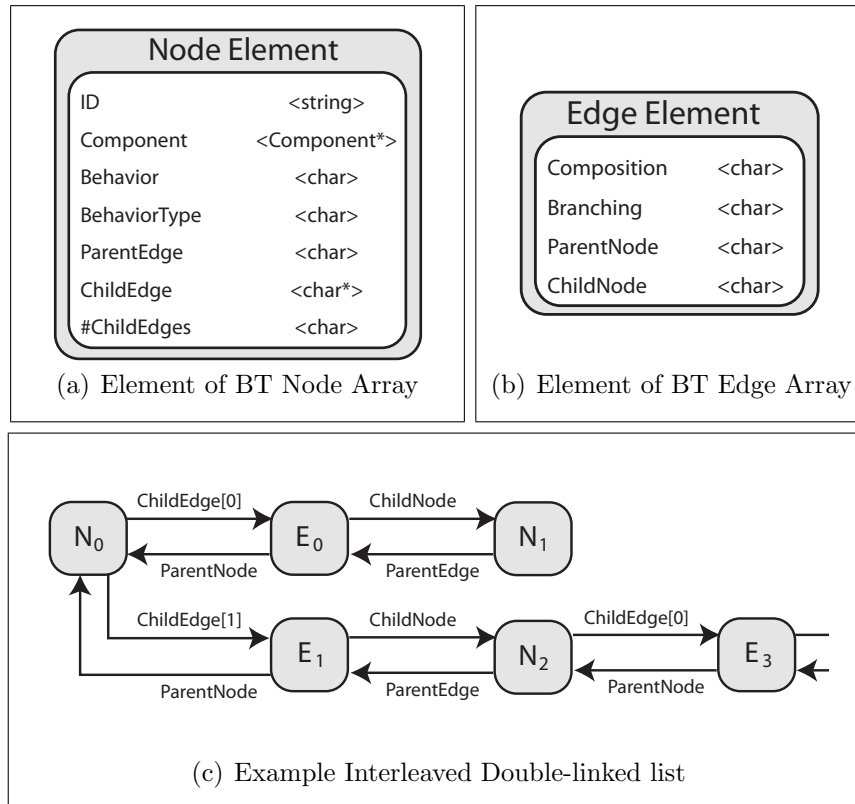


FIGURE 5.10: Code Representation of a Behavior Tree

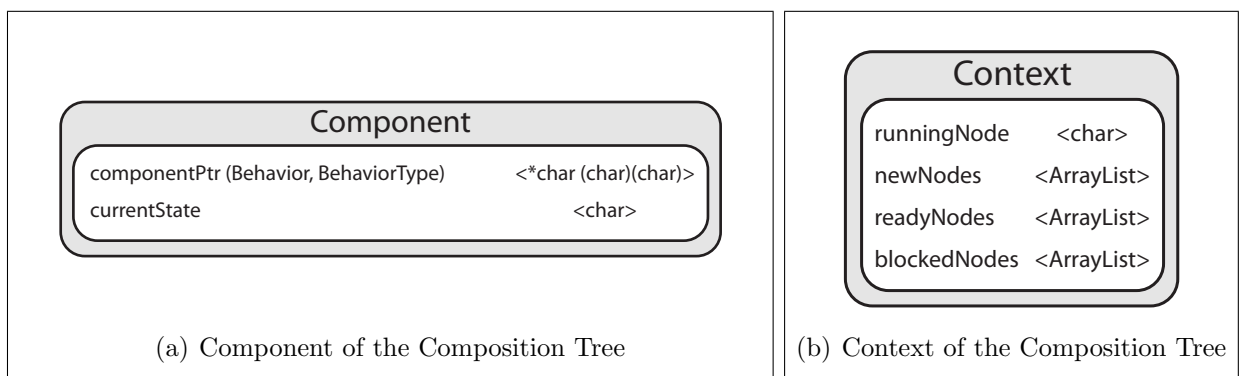


FIGURE 5.11: Code Representation of a Composition Tree

state of the Component and the initial states of the system's context form the DpyCT which is used to configure the initial context of the system. Once the system begins execution, the current state is used to store the component's state and the context stores the current context of the system forming the DynCT.

## 5.4 The Process Control Model

At the core of the eBRE is the five-state process control model in Figure 5.12 that has been modified to execute BTs. Process control models are commonly used in operating systems (OS) to interleave processes, providing the illusion that multiple processes are executing concurrently on a single processor. The interleaving of multiple processes is known as time sharing, and is performed by regulating the amount of time each process is executed according to a scheduling algorithm. When a process uses its allocated amount of execution time, it is quickly switched with a waiting process that is in the ready state. If this is performed with sufficient speed and reliability, it provides the illusion of concurrent execution.

The modified process control model considers each BT node as an individual process which it sets to the new, ready, blocked or running state based upon the BT semantics. The exit state is not strictly necessary for execution of the eBRE, as any process that is placed in the exit state may simply just be removed from the process control model. Another modification is warranted by the smaller granularity of a BT node process in comparison to an OS process. The small piece of encapsulated functionality that is represented by a BT node does not normally justify the need for time sharing, where running nodes are blocked during execution and switched with a waiting ready process. Instead, a BT node can follow

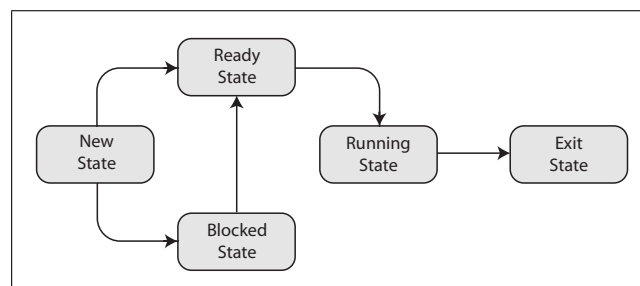


FIGURE 5.12: The Five-State Process Control Model

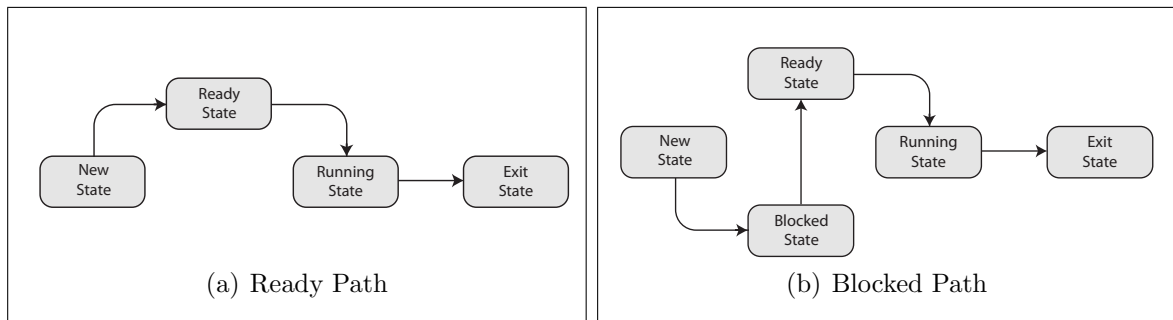


FIGURE 5.13: The two possible paths for a BT Node through the process control model

one of two possible paths through the process control model, as shown in Figure 5.13. The path a BT node takes through the scheduler, and the behavior it exhibits, is determined by its behavior type, the presence of any special edges, and the composition and branching which connects it to its parent node.

For example, a state realisation node<sup>2</sup> will take the ready path through the process control model. When the state realisation node is running, the scheduler will call the function pointer of the component associated with the node, and any user-defined encapsulated computation which the state realisation refers to will be executed. After this, the children of the state realisation node are added to the scheduler in the new state and the state realisation node is removed from the process control model. By default, the running behavior of all nodes includes being removed from the process control model and adding child nodes to the scheduler in the new state.

Comparatively, a guard node will take the blocked path. While the guard node is in the blocked state, it is periodically evaluated, and once it evaluates to true, the node is updated to the ready state. Unlike the state realisation node, when a guard node is run, it does not call a function pointer but the children of the node are still added to the scheduler (in the new state) and the guard node is removed.

Table 5.2 shows the paths that each type of BT node takes through the process control model. Selections and outputs follow the ready path, with the exceptions that selection node children are only added if the expression evaluates to true. Running an internal output results in the message being placed in the eBRE, and running an external output results

<sup>2</sup>In the following discussion a BT node is sometimes referred to based on its defining characteristic. For example, a BT node with a state realisation behavior type is referred to as a state realisation node.

	Node Type	Path	Running/Blocked Behavior
<b>Behavior Type</b>	State Realisation	Ready	Call component function pointer and execute user-defined code.
	Selection	Ready	Evaluate Expression. Only add children to new if expression is true.
	Guard	Blocked	Evaluate Expression. Unblock if expression is true.
	Input	Blocked	Wait to receive message from eBRE (Internal) or a component or the environment (External)
	Output	Ready	Output message to eBRE (Internal) or environment (External)
<b>Operator Edges</b>	Reference	Ready	Add Destination to process control model with new state
	Reversion	Ready	Remove all descendants of Destination from process control model regardless of state. Add Destination node to process control model with new state.
	Branch-Kill	Ready	Remove all descendants of Destination from process control model regardless of state.
	Synchronisation	Blocked	Check if all destination nodes are in blocked state, if so unblock.
<b>Parent Edge</b>	Atomic Composition	-	Child nodes are added with ready state at start of FIFO queue, instead of with new state.
	Alternative Branching	-	If node is unblocked or becomes the running node then remove all siblings from process control model regardless of state.

TABLE 5.2: The Behavior of BT nodes through the Process Control Model

in the message being sent out to the environment. Inputs follow the blocked path and are unblocked (move from the blocked state to the ready state) when a matching message is sent to the eBRE (from either an internal output node, a component, or the environment).

Mapping the semantics of node operators and parent edges into the process control model is more complex, but allows BTs with multiple threads to be executed on a single processor. Nodes with reference, reversion, branch-kill and synchronisation node operators are all statically transformed into special edges prior to being executed by the eBRE. Reference special edges are treated the same as normal edges with the destination node being added to the process control model in the new state. Branch-kill special edges result in the scheduler removing any descendants of the destination node from the process control model. Reversion special edges perform the tasks of a branch-kill followed by a reference. Nodes with a synchronisation node operator remain blocked until all the destination nodes

1.	$Blocked \Rightarrow Ready$
2.	$New \Rightarrow Ready \mid New \Rightarrow Blocked$
3.	$Running \Rightarrow Exit$
4.	$Ready \Rightarrow Running$

TABLE 5.3: Order of operation of the process control model

are in the process control model in the blocked state after which they are added to ready. The destination nodes are determined by the synchronisation nodes which have synchronisation special edges to each of the participating synchronisation nodes. When the nodes with a Synchronisation node operator are unblocked and eventually enter the run state, the synchronisation special edges are ignored and child nodes are added to the process control model in the new state.

If the parent edge of a node is sequential composition or parallel branching, then no further modifications are necessary to the previously described behavior. When nodes with alternative parent edges are unblocked or evaluate successfully in the running state, however, all the sibling nodes in the process control model must be removed by the scheduler. This ensures that only one alternative branch will be executed. Nodes with a parent edge that has atomic composition are added to the process control model with a ready state and the scheduler prioritises them to be the next node to change to the running state. Also, if an atomic block of nodes (group of nodes connected with atomic composition) are composed of guard nodes, and any of the guard nodes evaluates as false, the first node of the atomic block is added again to the process control model with a blocked status to be re-evaluated.

In one cycle of the eBRE, only one group of nodes in the process control model in the same state may transition state. The change of the state of nodes in the process control model is managed by an order of operation shown in Table 5.3. The eBRE will first attempt to find any nodes in the blocked state which are ready to run and unblock them. If no nodes can be unblocked, then any nodes in the new state are moved to the ready or blocked state based upon the path the node takes through the process control model. If no new nodes exist, then the node with a running state, known as the running node, may be executed. If a running node is not assigned, then a node in the ready state may be selected to become a running node using a First-In First-Out (FIFO) scheduler (with the exception of atomic

composition). Also, in each cycle messages are cleared.

## 5.5 Deployment of the One-Minute Microwaver



FIGURE 5.14: The BE One-Minute Microwaver

Figure 5.14 shows the deployment of the BE One-Minute Microwaver in a mock-up microwave oven. The BE model of the microwave oven is implemented on an AVR atmega128 operating at 16 MHz. The total size of the project, complete with manually created component code, was 5.5 kilobytes. Approximately 20% of the total code was manually created and another 20% was automatically generated from the BE model. The remaining 60% is accounted for by the eBRE.

Interactions across the system-environment boundary are implemented using interrupts from the microcontroller which add the associated external input in the DBT. The interrupt code adds a message to the eBRE using the *addExternalMessage* function. The system only responds to the external input if an external input node is waiting to receive the message in a blocked state in the process control model of the eBRE.

Interactions across the system-component boundary are implemented using functions named after each component. These function accept the value of the behavior and behavior type of the node that is responsible for the component being accessed. Before a behavior is sent to a component, however, the eBRE has a simple expression parser which attempts to evaluate selections and guards against the current state of a component stored in the DynCT.



The expression parser accesses the encapsulated computation of a component to evaluate state realisations and selection and guards with more complex expressions. Computation associated with a state realisation behavior type returns a boolean value to indicate if it was successfully computed. The same boolean value is used to return the result of selection and guard evaluations to the eBRE. Components can also execute behavior external to the eBRE by setting interrupts that are associated with the component. This external behavior can only interact with the eBRE by using external inputs which are sent using the *addExternalMessage* function.

Figure 5.15 describes how interactions between the system and the components are implemented in the eBRE. The figure shows the implementation of the Timer component of the One Minute Microwave and how it interacts across the system-component boundary. The Timer component consists of three pieces of encapsulated behavior which correspond to the state realisations *OneMinute*, *ExtraMinute* and *Stopped*. The *OneMinute* behavior sets the value of a seconds attribute and enables a timer interrupt associated with the Timer component. The *ExtraMinute* behavior increases the value of the seconds attribute. The *Stopped* behavior disables the timer interrupt associated with the Timer component. The timer interrupt occurs once every second and decreases the value of the seconds attribute. If the seconds attribute reaches zero, then a *Timed Out* message is returned to the eBRE from the timer interrupt.

The deployment of the microwave oven can be easily altered by substituting a different *CompositionTree.h* header file. This allows different starting contexts and initial states to be described. For example, currently the microwave oven starts at the OVEN [Open] node in the Behavior Tree and the Powertube component is set to off and Light component is set to on. The microwave oven could easily be changed to be deployed to start in the OVEN [Idle] node of the BT with the Light component being initialised to off. Even more complex deployment contexts are possible, with the system starting with several nodes in the new, blocked, ready and running states.

The deployed microwave oven can also be easily altered by substituting different components. For example, testing can be performed more quickly by using a timer that times out after five seconds rather than one minute. This can be easily achieved by substituting

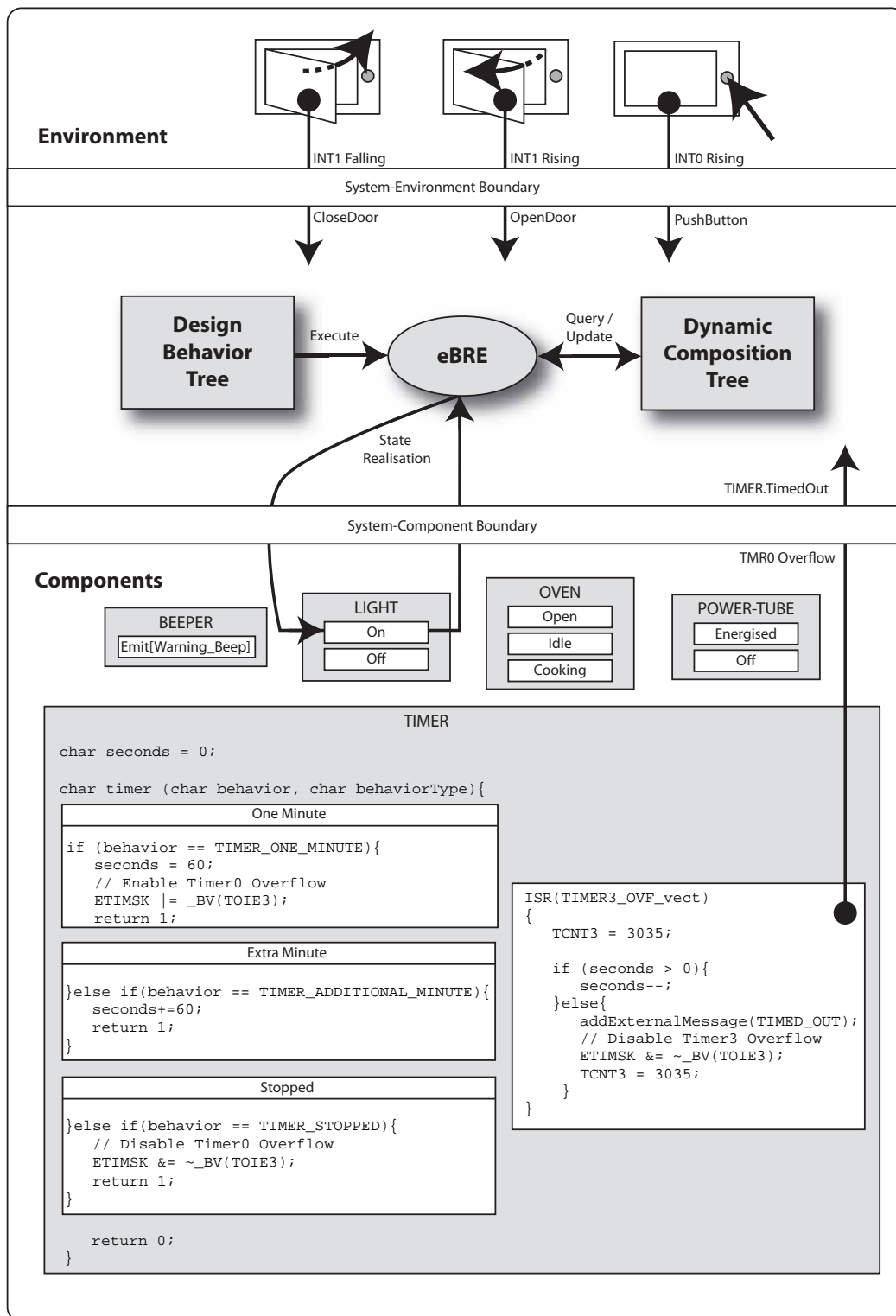


FIGURE 5.15: BE Boundaries of the One Minute Microwave

the five-second timer component code with the one-minute timer component code during testing.

## 5.6 Discussion

The approach described in this chapter only addresses the implementation of a BE design on an embedded system consisting of a single microcontroller. Research in embedded systems is increasingly focusing on deploying designs on embedded systems composed of a networked group of microcontrollers. Future work could focus on using an approach such as algorithm architecture adequation (AAA) to distribute a BE design over several microcontrollers. Another possibility is to investigate the feasibility of executing BTs on specialised hardware by translating BTs to VHDL.

The BT M2M transformation language described in this chapter is limited in that it requires manual implementation in a transformation language such as ATL. Future work could address this limitation by investigating the feasibility of automatically generating ATL code from the BT M2M transformation language.

Finally, as stated previously, the one-minute microwaver case study does not highlight the ability of BE to develop embedded systems with quite complex integrated behavior and multi-threaded control. The use of predefined components is also not demonstrated in the case study, although it is possible.

## 5.7 Conclusion

The increasingly ubiquitous nature of embedded systems is causing greater expectations of the functionality that can be provided. This increase in functionality causes increases in the size and complexity of the code of embedded systems. Despite this, around 60% of embedded systems programmers are not taking advantage of graphical modeling approaches such as UML, state machines and block diagrams. These approaches address complexity by capturing the design of embedded systems graphically and decompose the task of coding into smaller functional blocks. Using BE to design embedded systems, however, differs from these

approaches by providing a clear path from natural language requirements to an embedded design consisting of an integrated set of components. The BE approach creates an embedded system using a scaleable methodology that minimises the complexity of the task. The design graphically captures the complex integrated behavior of the components in the BML. The design is executed by an embedded system using a small footprint run-time environment that integrates the components, removing the need to manually code the integrated behavior of the components. The remaining component code that has to be manually defined has a clear and concise purpose and is separated from any integrated behavior. The contributions of this chapter are summarised below:

1. The BE design stage is supported by a toolset that utilises the Eclipse Modeling Project to enable models to be deployed as an embedded system. The toolset consists of:
  - (a) An editor for describing BE models.
  - (b) Model transformations to make BE models executable.
  - (c) A virtual run-time environment for executing BE models tailored specifically to embedded systems
2. The eBRE is a virtual run-time environment that executes BE designs for embedded systems. At the core of the eBRE is a process control model commonly used in operating systems. Using a process control model provides the possibility of real-time extensions to the eBRE in the future by modifying the scheduler.
3. The deployment of the one-minute microwaver as an embedded system highlights several benefits of the eBRE:
  - (a) Component integration is separated and generated independent of manually defined component code.
  - (b) The remaining component code that has to be manually defined has a clear and concise purpose and is simplified by the absence of any integrated behavior.
  - (c) Interactions between the environment and the embedded system are easily added using interrupts. The only coding required to use an interrupt is to manage their

configuration and to add the associated messages to the eBRE when the interrupt occurs.

- (d) Testing and debugging is aided by the DynCT which provides the current system context consisting of the state of each of the BT nodes in the scheduler. Manually defined component code can be added to the system in stages, allowing the system to be built incrementally and allowing issues with component code to be isolated.
- (e) The low coupling of the manually defined component code increases its potential for being reused in future projects.



## Part III

# Integrated Modeling





# 6

## Co-modeling

Interactive modeling is the second type of application of MDE as categorised by Bèzivin [BBJ07]. Interactive modeling consists of a system and a model co-existing with each other with any changes in one influencing the other. In this chapter, we demonstrate how interactive modeling can be used to investigate the impact of design decisions on a system composed of integrated software and hardware components. We refer to the resulting early-stage strategy as co-modeling, in reference to the established practice of co-design, which has similar goals but is applied later in the development process.

Our approach to co-modeling [MFD08, MFD09] uses multi-view modeling to separately model the software and hardware aspects of the system. To achieve this separation, our approach exploits and integrates BE with Modelica, an equation-based, object-oriented mathematical modeling language suited to modeling complex physical systems in multiple domains. We demonstrate our co-modeling approach using the case study of an automated

train protection system. The case study demonstrates how to construct a co-model, which is then used to investigate and document numerous co-modeling scenarios.

Co-modeling addresses the *software/hardware integration problem*, which is caused by the increasingly co-dependent nature of software and hardware in large-scale systems. This co-dependency is unlikely to be reflected in the set of functional requirements for an integrated software/hardware system, since the requirements often do not describe the interdependencies among the software and hardware components that will impact the construction of a successful integrated design. This omission impacts the early stages of development of both the software and hardware aspects of a system. The software specification created from the requirements will often lack the quantified and temporal information that is needed when focusing on software/hardware integration. Determining this information requires details of the hardware components to be specified early in development and involves determining the characteristics of sensors, actuators, and the architecture on which the software will be deployed. Thus, at the early stages of system development many decisions must be made about how the system will be realised as a combination of integrated software and hardware components. The software/hardware integration problem involves the need to make these decisions early in development when they are poorly informed and the impact upon both the software and hardware design of the system is unclear.

There is an increased risk of incompatibility between software and hardware components and the introduction of defects if the software/hardware integration problem is not adequately addressed. Delaying addressing the software/hardware integration by developing software and hardware specifications independently also increases the risk of incompatibility due to the likelihood of contradicting assumptions as to how integration will occur. Determining the design that best integrates software and hardware components can also be a wicked problem: a problem with complex interdependencies such that the solution of one aspect of the problem often creates or reveals previously unidentified problems in other aspects. Co-modeling can be used to investigate software/hardware integration that can help identify and resolve software/hardware integration problems at an earlier stage when they are easier and cheaper to fix. This is not only beneficial for locating potential

incompatibilities and defects, but can also help locate software and hardware configurations that combine to create a design that is superior at meeting the needs of the system.

The remainder of the chapter is structured as follows. The next section describes co-modeling in more detail and provides a comparison with the current established practice of co-design. We then describe how to construct a co-model using a case study of an automated train protection system. The co-model is used to simulate several scenarios to investigate how different software/hardware partitions affect the interactions, timing and behavior of individual components and the system as a whole. We conclude the chapter with a discussion of how co-modeling can be used to support the development of cyber-physical systems, an emerging form of large-scale systems composed of a large network of embedded devices.

## 6.1 From Co-design to Co-modeling

Co-design is the current established practice of dealing with the design of integrated software/hardware systems. Co-design involves mapping the functionality of an embedded system onto a hardware platform that is composed of a combination of software operating on general purpose microprocessors (such as a CPU or DSP) and specialised hardware integrated circuits (such as an ASIC or FPGA). The goal of co-design is to find a mapping that maximises the satisfaction of constraints important to embedded systems development such as timing, weight, power-consumption, reliability and cost [Wol03].

Two tasks are required to be performed to find the most suitable co-design mapping. Firstly, the functionality of the embedded system must be partitioned to operate either in software or hardware. Secondly, the composition of microprocessors and integrated circuits which form the hardware platform must be determined. As with the software/hardware integrated problem mentioned previously, these two tasks also constitute a wicked problem. The appropriate software/hardware partitioning is hard to determine without access to the completed hardware platform. Likewise, development of a suitable hardware platform requires knowledge of the partitioning of the functionality into software and hardware.

This problem is addressed in co-design by concurrently investigating software/hardware partitions and potential hardware platforms using a virtual platform. Tool support such as

Mathwork's Real-Time Workshop and Simulink HDL Coder supports co-design by automatically generating C and hardware definition language (HDL) code from MATLAB/Simulink models. The effectiveness of the partitioned functionality can then be quickly investigated by emulation on a virtual platform. Different hardware platforms can also be investigated by changing the model of the hardware platform on which the partitioned functionality is emulated.

The success of co-design at managing the design of integrated software/hardware systems is due to two restrictions. Firstly, co-design restricts the definition of software and hardware to functionality executing on general-purpose microprocessors or specialised hardware ICs. This restricted definition of software and hardware places a second restriction on co-design. Co-design of an integrated software/hardware system can only occur late in development when the functionality of a discrete embedded system is well defined.

Co-modeling removes these restrictions to apply the principles of co-design to the software/hardware integration problem. Software and hardware are considered in a broader sense when performing co-modeling. Hardware components consist of mechanical, electrical, electronic, hydraulic, and thermal components that form part of the system being built. Software components consist of a modular grouping of functionality operating on a hardware platform. This broader definition of software and hardware allows co-modeling to be performed earlier in development.

Despite being performed earlier in development, however, co-modeling still uses the concept of investigating software/hardware partitions on a virtual platform. Co-modeling begins after the requirements have been formalised and integrated. The resulting specification is used to perform software/hardware partitioning by identifying potential software/hardware interactions. Different hardware components consisting of either sensors or actuators are then investigated to determine the best software/hardware partitioning to meet the constraints of the system. Due to the broader definition of hardware and software, a virtual environment is used in place of a virtual platform, which simulates the environment where the integrated software/hardware system will be deployed.

Figure 6.1(a-b) shows the software/hardware partitioning performed by co-design and co-modeling respectively. Comparison of the two figures highlights the different stages of

development at which the two approaches are applied. Co-design partitions functionality into software and hardware that is deployed on a hardware platform. Co-modeling partitions a specification built from the integration of the formalised requirements into software and hardware components. Later in development the software components that are deployed on a hardware platform could be further partitioned using co-design to execute on general purpose microprocessors and specialised hardware ICs.

Another important point to note from the figures is that co-design does not directly map the functionality to particular microprocessors or ICs. Instead, languages such as C are used for mapping software, and HDLs such as Verilog and VHDL are used for mapping hardware. Similarly, co-modeling uses modeling languages as an intermediary to model the software and hardware components. Our approach to co-modeling partitions the software and hardware aspects of the system into the BE and Modelica languages respectively.

The choice of Behavior Engineering (BE) and Modelica is central to our co-modeling approach, which consists of the process shown in Figure 6.1(c). Unlike the functionality that is partitioned in co-design, the requirements used for co-modeling are written in natural language and must be formalised prior to partitioning. BE is ideal for this task, and is used to formalise and integrate the natural language requirements to form an executable specification. The BE specification contains information about software components, hardware components and the environment in which the system will be deployed. The next step is to use this specification to determine an appropriate software/hardware partition.

Software/hardware partitioning of the BE model is performed as described in Chapter 4. When performing co-modeling, however, components that interact with the system across the system-environment boundary and components that interact with the system across the system-component boundary are now described in the Modelica modeling language. Modelica is also used to capture how the software represented by the DBT interacts with the hardware components. In order to investigate these interactions, various scenarios are also described in the Modelica model which simulate software/hardware interactions by executing the BE model. The simulations produce a graphical, documentable result which can be analysed to determine the effect different software/hardware partitions have on the timing, performance, and complexity of individual components, and on the system behavior.

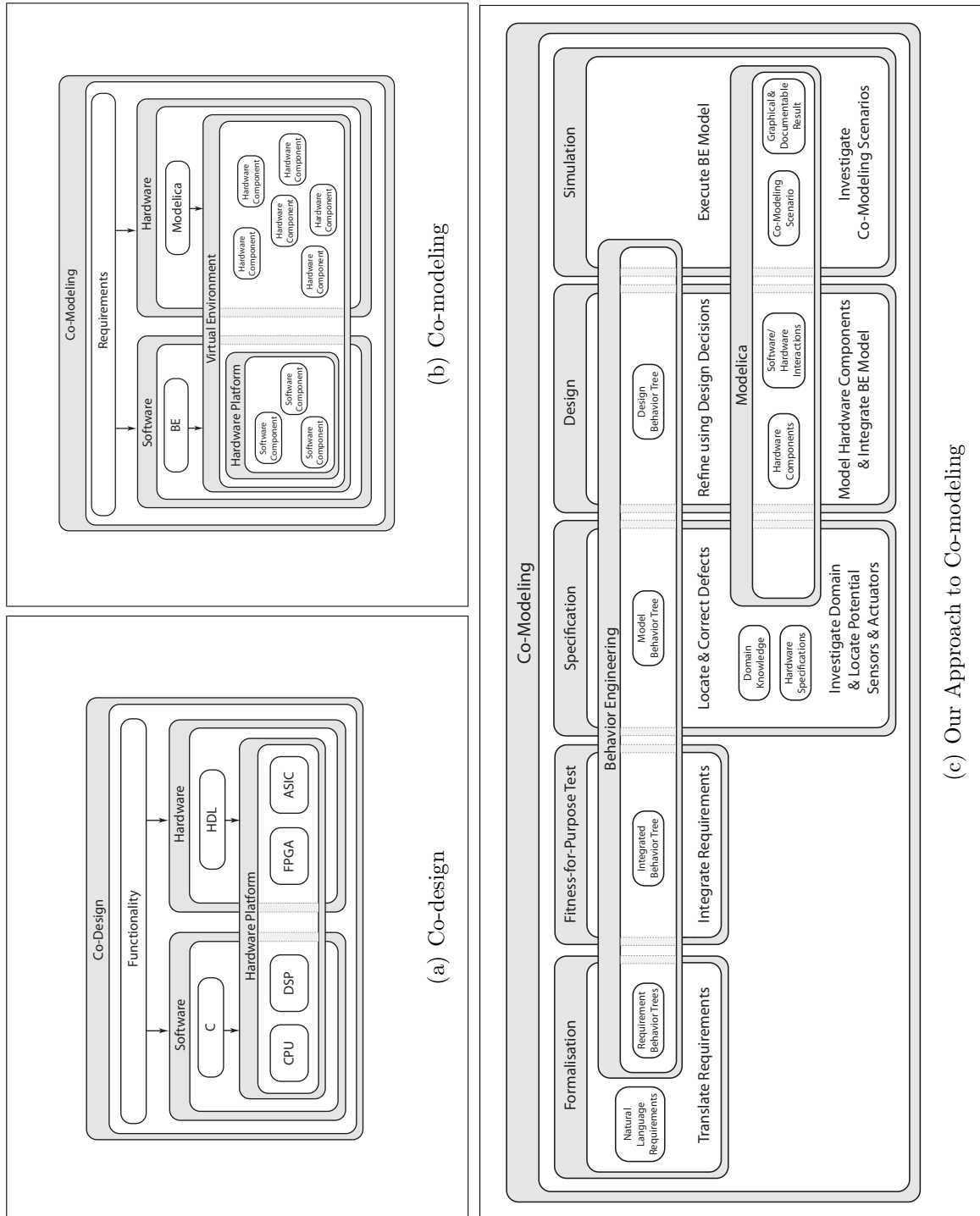


FIGURE 6.1: From Co-design to Co-modeling

## 6.2 Constructing a Co-Model

In order to demonstrate our co-modeling approach, we will now describe the co-modeling of an automated train protection (ATP) system. Most rail systems have some form of train protection system that use track-side signals to indicate potentially dangerous situations to the driver of the train. The simplest train protection systems consist of signals with two states: green to continue along the track and red to apply the brake to stop the train. More sophisticated systems include detailed information such as speed profiles for each section of the track.

Accidents still occur using a train protection system when a driver fails to notice or respond correctly to a signal. ATP systems are used to reduce the risk of these accidents. These systems automate the response of the train to the track-side signals by sensing each signal and monitoring the reaction of the driver. If the driver fails to act appropriately, the ATP system takes control of the train and responds as required.

The ATP system used in this case study has three track-side signals: proceed, caution and danger. When the ATP system receives a caution signal, it monitors the driver's behavior to ensure the train's speed is being reduced. If the driver fails to decrease the train's speed after a caution signal, or the ATP system receives a danger signal, the train's brakes are applied. The complete requirements of the ATP system can be found in Table 6.1.

These requirements were originally used as an assignment on high integrity software development in which students develop components of the ATP system in SPARK, a safety critical subset of the ADA language [Ire08]. The requirements have also been used to demonstrate the composition of components using exogenous connectors<sup>1</sup> [LLW06].

### 6.2.1 Modeling the Requirements

Co-modeling begins by modeling the software requirements using BE by progressing through the BMP stages of formalisation, the fitness-for-purpose test and specification. It is not until design decisions are applied to the specification, however, that the software / hardware partitioning of the system begins to be determined. In the following section we discuss the

---

<sup>1</sup>For further details on exogenous connectors and a comparison with BTs see Appendix A.

Requirement	Description
R1	The ATP system is located on board the train. It involves a central controller and five boundary subsystems that manage the sensors, speedometer, brakes, alarm and a reset mechanism.
R2	The sensors are attached to the side of the train and detect information on the approach to track-side signals, i.e. they detect what the signal is displaying to the train driver.
R3	In order to reduce the effects of component failure three sensors are used. Each sensor generates a value in the range 0 to 3, where 0, 1 and 2 denote the danger, caution, and proceed signals respectively. The fourth sensor value, i.e. 3, is generated if an undefined signal is detected, e.g. may correspond to noise between the signal and the sensor.
R4	The sensor value returned to the ATP controller is calculated as the majority of the three sensor readings. If there does not exist a majority then an undefined value is returned to the ATP controller.
R5	If a proceed signal is returned to the ATP controller then no action is taken with respect to the train's brakes.
R6	If a caution signal is returned to the ATP controller then the alarm is enabled within the driver's cab. Furthermore, once the alarm has been enabled, if the speed of the train is not observed to be decreasing then the ATP controller activates the train's braking system.
R7	In the case of a danger signal being returned to the ATP controller, the braking system is immediately activated and the alarm is enabled. Once enabled, the alarm is disabled if a proceed signal is subsequently returned to the ATP controller.
R8	Note that if the braking system is activated then the ATP controller ignores all sensor input until the system has been reset.
R9	If enabled, the reset mechanism deactivates the train's brakes and disables the alarm.

TABLE 6.1: Requirements of the ATP system

modeling of the ATP system using BE.

## Formalisation

Figures 6.2 and 6.3 show the requirement behavior trees resulting from formalising the requirements of the ATP system. The issues list that accompanies the RBTs is shown in Table 6.2.

RBT3 and RBT4 make use of the BT set notation extension to model a set composed of multiple sensors. In RBT3, the forAll symbol,  $||$ , indicates to execute an instance of the behavior below for each instance of the three sensors. In RBT4, the set notation is used to define a set called MAJORITY, which stores the majority of the values of the sensors as calculated by the sensor set.



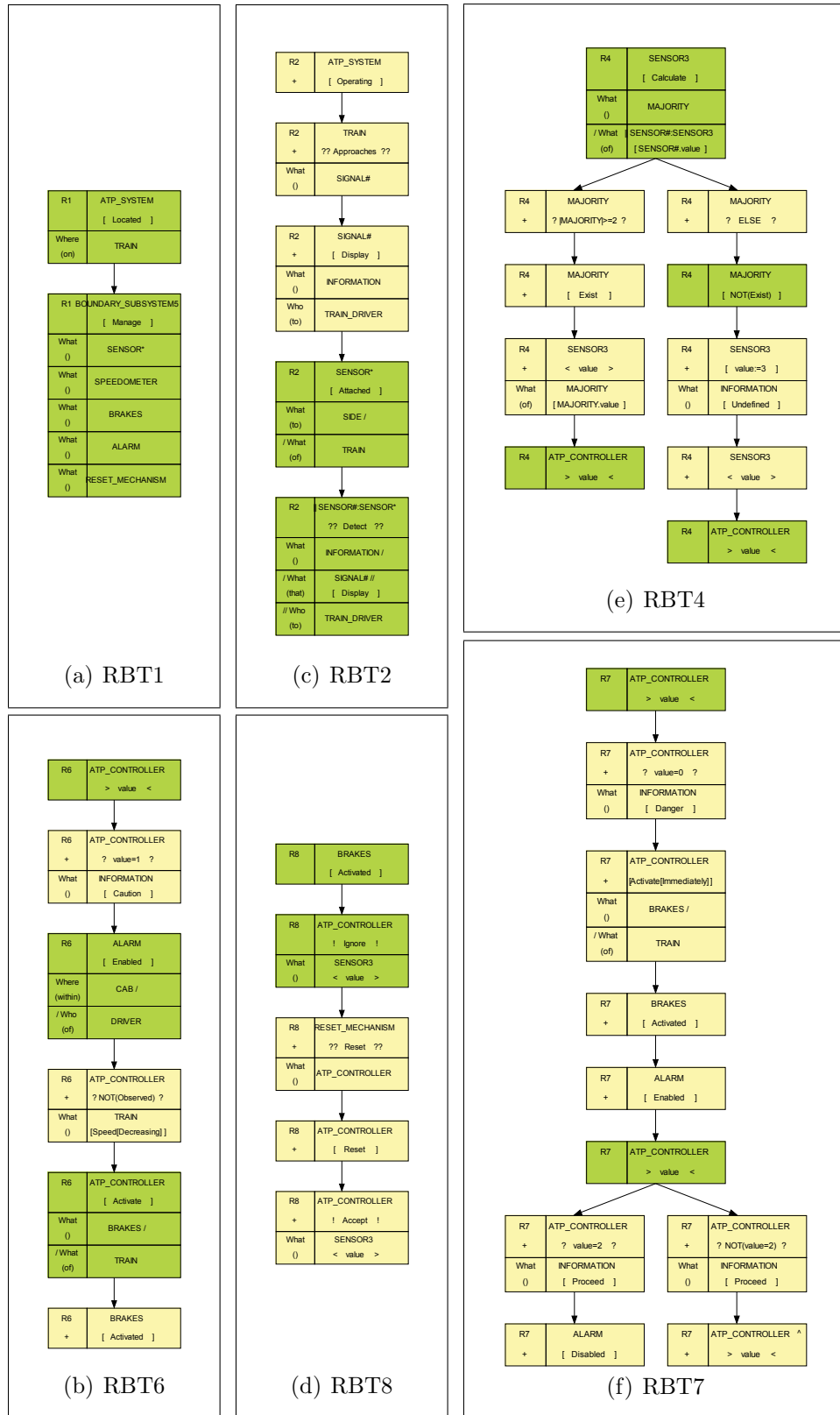


FIGURE 6.2: The Requirement Behavior Trees of the ATP System (R1,R2,R4,R6,R7,R8)



FIGURE 6.3: The Requirement Behavior Trees of the ATP System (R3,R5,R9)

Requirement	Issue
R2	The ATP must be operating for the sensors to detect information
	The sensors detect information on the approach of the <i>train</i> to the signal
	The signal displays <i>information</i> to the train driver
R3	The generated sensor value is derived from the information detected by each of the sensors that is displayed by the signal
	If an undefined signal is detected it may or may not correspond to noise existing between the signal and the sensor
R4	A majority is two or more of the same sensor values
	An undefined value corresponds to the numeral 3
	The undefined value is returned to the ATP controller by the sensor group
	If a majority exists then the sensor value of the majority is returned to the ATP controller by the sensor group
R5	A proceed signal corresponds to numeral 2
R6	A caution signal corresponds to numeral 1
	The ATP controller observes if the speed of the train is decreasing or increasing
	The ATP controller activating the brakes of the train causes the brakes to be activated
R7	A danger signal corresponds to the numeral 0
	The braking system of the train
	Activating the braking system of the train causes the brakes to be activated
	A proceed signal corresponds to numeral 2
	If a signal other than proceed is returned to the ATP controller, keep waiting to receive a proceed signal
R8	The reset mechanism is responsible for resetting the ATP controller and causes the ATP controller to be reset
	After the ATP controller has been reset it can accept sensor input
R9	If the reset mechanism is enabled
	The reset mechanism deactivating the train's brakes results in the brakes being deactivated
	The reset mechanism disabling the alarm results in the alarm being disabled

TABLE 6.2: Issues found during translation of requirements of the ATP System

### Testing Fitness for Purpose

Figure 6.4<sup>2</sup> shows the IBT resulting from integrating the RBTs of the ATP system. Note that RBT8 and RBT9 are integrated twice to form the IBT. The associated ICT is shown in Figure 6.5<sup>3</sup>.

The following integrations were made to form the IBT:

- The behavior of RBT1 precedes the behavior of RBT2.

<sup>2</sup>To view this figure in more detail, please refer to the electronic version included as part of the supplementary material.

<sup>3</sup>To view this figure in more detail, please refer to the electronic version included as part of the supplementary material.

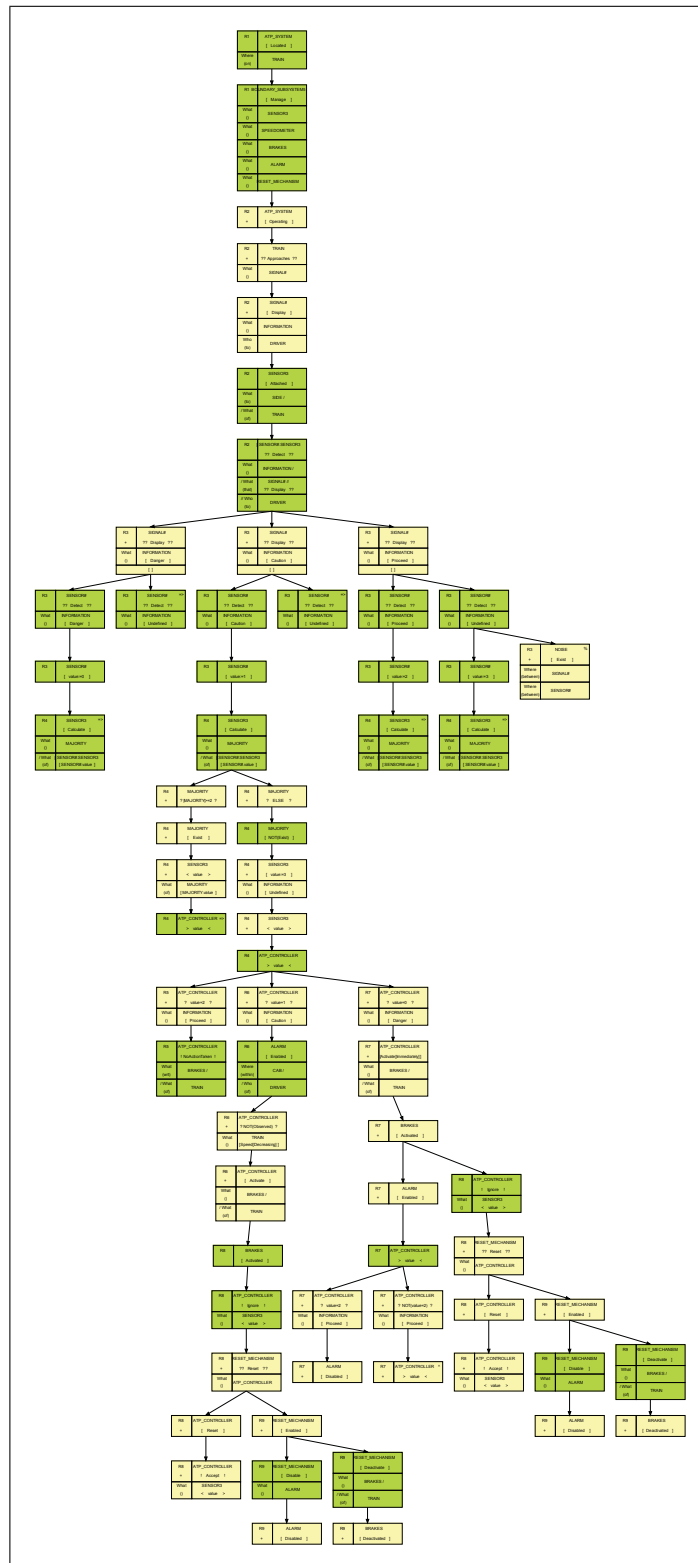


FIGURE 6.4: The Integrated Behavior Tree of the ATP System

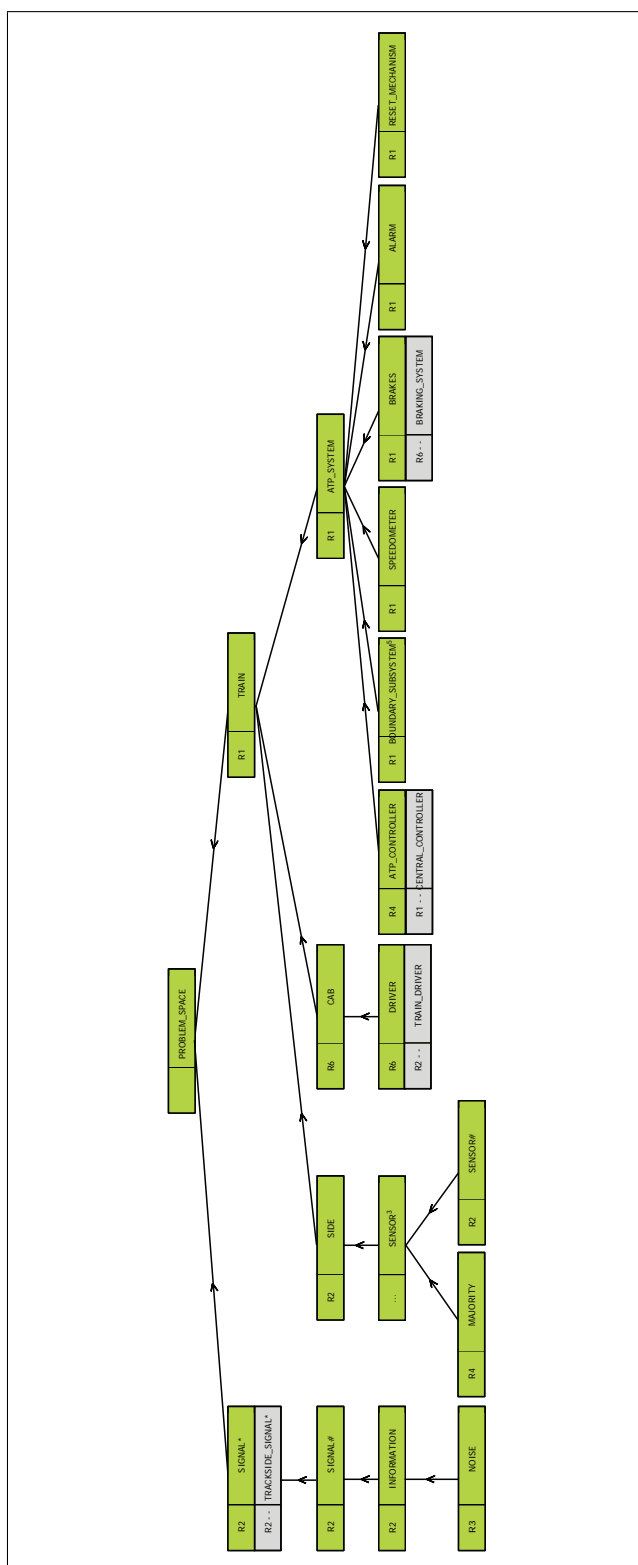


FIGURE 6.5: The Integrated Composition Tree of the ATP System

- RBT2 and RBT3 share the integration node  $||\text{SENSOR\#} : \text{SENSOR}^3$ .
- The behavior of RBT3 precedes the behavior of RBT4 because the  $\text{SENSOR\#}$  values are used to calculate a majority.
- RBT4, RBT5, RBT6 and RBT7 share the integration node  $\text{ATP\_CONTROLLER} >\text{value}<$ .
- RBT6 and RBT8 share the integration node  $\text{BRAKES} [\text{Activated}]$ .
- RBT7 and RBT8 share the integration node  $\text{BRAKES} [\text{Activated}]$ .
- The behavior of RBT9 follows the behavior of RBT8 where the  $\text{RESET\_MECHANISM}$  resets the  $\text{ATP\_CONTROLLER}$ .

## Specification

Figure 6.6<sup>4</sup> shows the partially completed MBT after the system component was chosen and looping behavior was resolved. The following actions were applied to the IBT:

- $\text{ATP\_SYSTEM}$  is a system component, any node in the MBT where it is the primary component uses a double-line border.
- Leaf node  $\text{R5 ATP\_CONTROLLER !NoActionTaken! } [What(wrt)] \text{ BRAKES } / [What(of)] \text{ TRAIN}$  reverts to  $\text{ATP\_SYSTEM} [\text{Operating}]$  to wait for the train to approach another signal.
- Re-order the causality of R8 and R9 so that the brakes are deactivated first, followed by the alarm being disabled, followed by the ATP controller being reset.
- Leaf node  $\text{R8 + ATP\_CONTROLLER !Accept! } [What()] \text{ SENSOR}^3 >\text{value}<$  reverts to  $\text{ATP\_SYSTEM} [\text{Operating}]$  to wait for new sensor input.
- Reposition  $\text{R8 ATP\_CONTROLLER !Ignore! } [What()] \text{ SENSOR}^3 >\text{value}<$  to occur after  $\text{R7 + ALARM} [\text{Enabled}]$ .

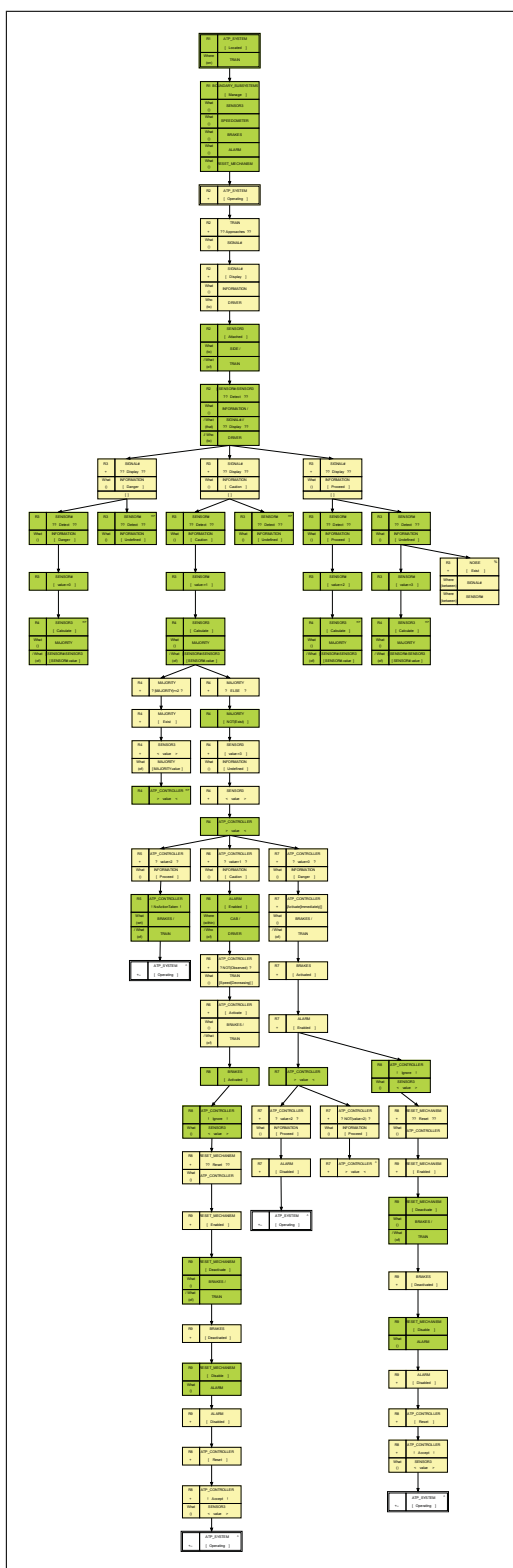


FIGURE 6.6: Partial Model Behavior Tree #1 of the ATP System

Figure 6.7<sup>5</sup> shows a second stage of the partially completed MBT after missing behavior was identified by checking constraints, events and initial states. The following actions were applied:

- The MBT does not current specify what behavior occurs when an undefined signal is returned to the ATP controller. Assume the ATP controller ignores the undefined signal and waits for a new signal to be detected by the sensors.
- The behavior that occurs if the ATP controller observes the train's speed decreasing after a caution signal has been received is missing. Behavior is added so that: if a danger signal is received the brakes are immediately activated; if a proceed signal is received the alarm is disabled; and if the train's speed increases before either of these signals are received then the ATP controller should activate the train's brakes.
- Requirements R7, R8 and R9 have conflicting behavior. R7 states that after the braking system is activated a proceed signal disables the alarm. This conflicts with R8 which states that after the braking system is activated all sensor input is ignored until the ATP is reset. It also conflicts with R9 which gives the reset mechanism unconditional responsibility for disabling the alarm. This conflicting behavior is resolved by giving R8 and R9 priority over R7. That is, a proceed signal can only disable the alarm after the alarm has been enabled but prior to the brakes being activated. After the brakes have been activated all sensor input is ignored until the ATP controller is reset. Also, resetting the ATP controller after the brakes have been activated causes the train's brakes to be deactivated and disables the alarm.
- The Reset mechanism enters the Enabled state, but never enters another state thereby indicating a possible missing state. The node RESET\_MECHANISM [Disabled] is added after the Alarm is disabled with a missing traceability status.

---

<sup>4</sup>To view this figure in more detail, please refer to the electronic version included as part of the supplementary material.

<sup>5</sup>To view this figure in more detail, please refer to the electronic version included as part of the supplementary material.



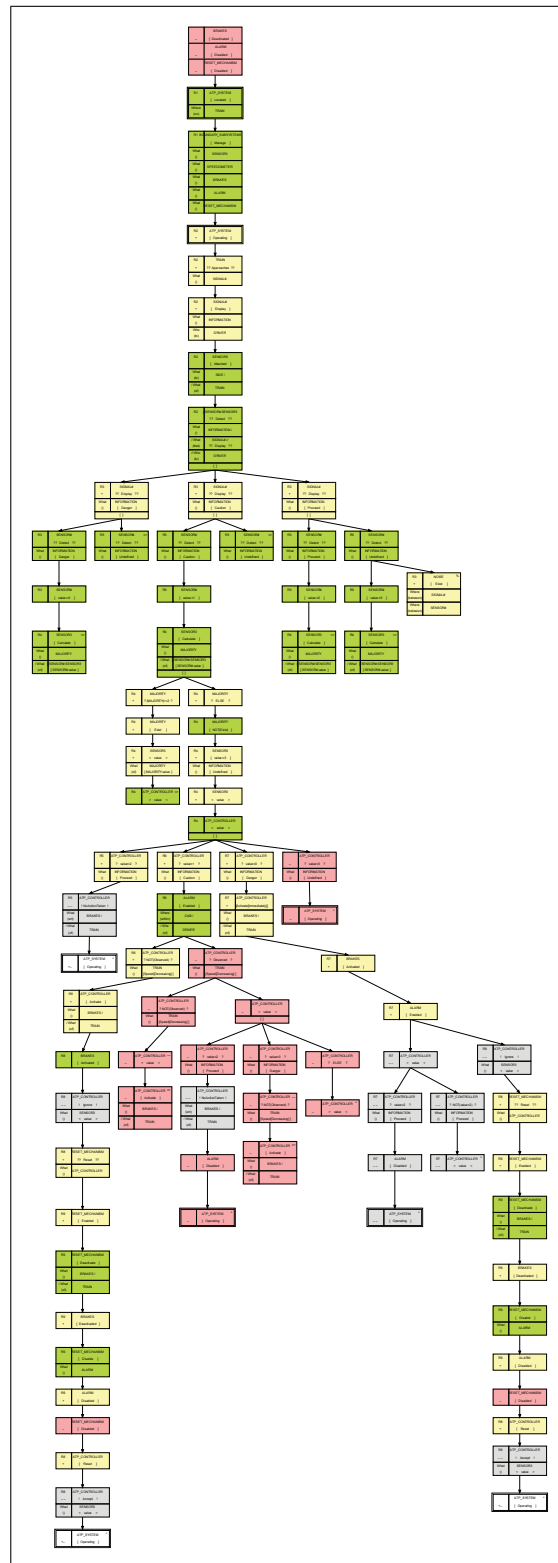


FIGURE 6.7: Partial Model Behavior Tree #2 of the ATP System

- The parent edge of the R3 + SIGNAL# ??Display?? *[What()]* INFORMATION sibling nodes is changed to alternative branching. This reflects that the signal can only display one signal at any one time.
- The parent edge of the R4 + MAJORITY ?|MAJORITY| >=2? and R4 + MAJORITY ?ELSE? sibling nodes is changed to alternative branching.
- The child edges of the R4 ATP\_CONTROLLER >value< node and the - ATP\_CONTROLLER >value< node are changed to alternative branching. This reflects that only one signal value can be received and acted upon by the ATP controller at any one time. This ensures that the R8 ATP\_CONTROLLER !Ignore! *[What()]* SENSOR >value< and R8 + ATP\_CONTROLLER !Accept! *[What()]* SENSOR >value< assertions are satisfied which is shown by giving the nodes a deleted traceability status. The ATP\_CONTROLLER !NoActionTaken! *[What(wrt)]* BRAKES/ */[What(of)]* TRAIN assertions are now also satisfied and are given a deleted traceability status.
- The components Brakes, Alarm and the Reset Mechanism should also have initial states. Brakes should begin in the Deactivated state, Alarm should begin in the Disabled state and the Reset mechanism should begin in the Disabled state.

The MBT is completed by decoupling environmental components and by resolving events. The finalised MBT is shown in Figure 6.8<sup>6</sup>. The Train, Signal, Sensor and Reset mechanism components are decoupled and placed in concurrent environmental threads of behavior. This is to reflect that the Train, Signal, Sensor and Reset mechanism can interact with the environment at any time regardless of the current state of the ATP system. The following actions were taken to finalise the MBT:

- TRAIN >>Approaches<< *[What()]* SIGNAL causes an ApproachSignal internal output message.
- The Signal environmental thread is dependent on receiving the ApproachSignal message. The SIGNAL ??Display?? *[What()]* INFORMATION event is changed to a

---

<sup>6</sup>To view this figure in more detail, please refer to the electronic version included as part of the supplementary material.

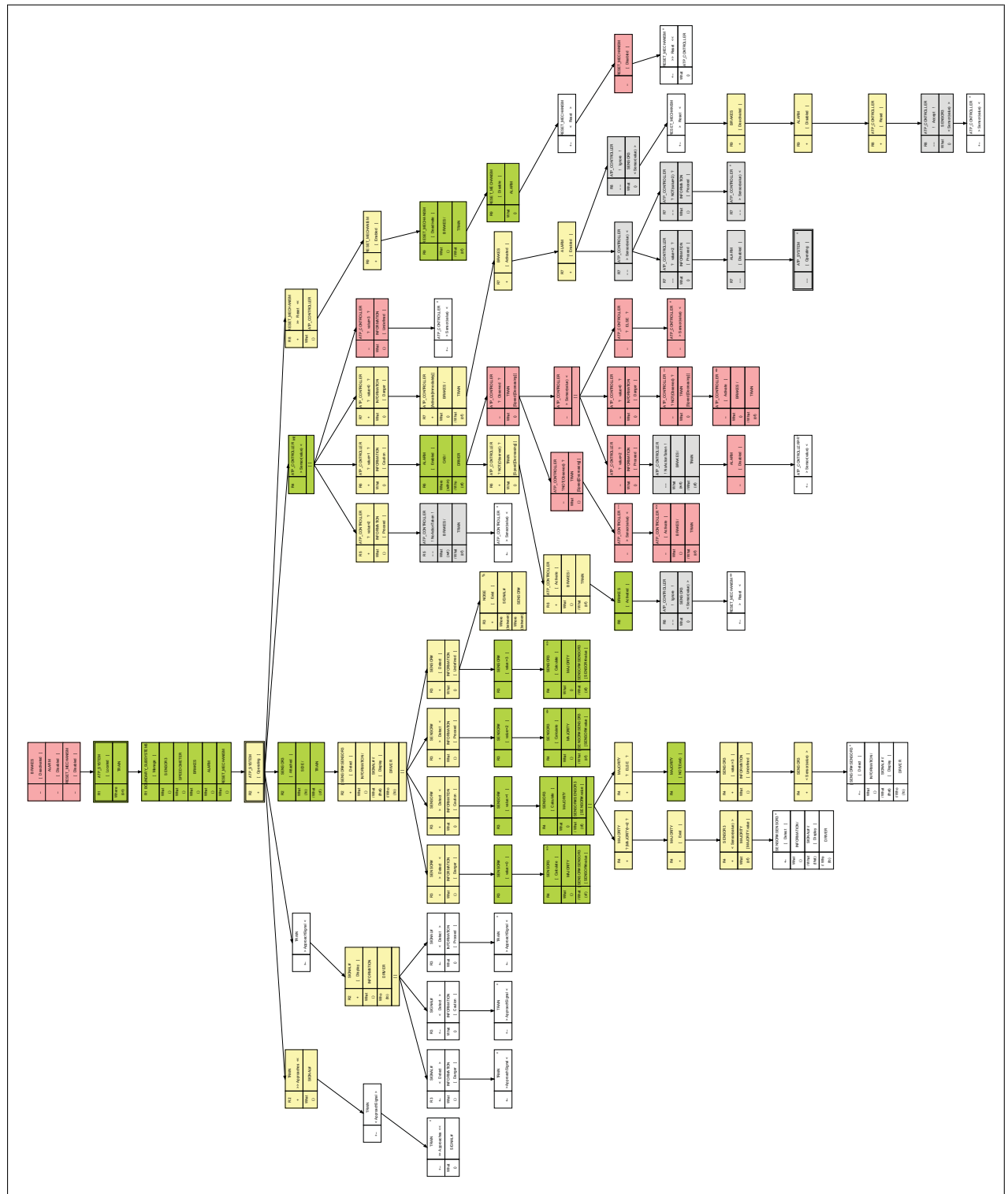


FIGURE 6.8: Model Behavior Tree of the ATP System

Detect internal output to match the SENSOR ??Detect?? [*What()*] INFORMATION event.

- The Sensor ??Detect?? [*What()*] INFORMATION event is changed to a Detect internal input with the exclusion of the INFORMATION [Undefined] event which remains a state realisation. This allows the non-deterministic semantics of alternative branching to ensure that a Sensor will occasionally erroneously detect an undefined signal.
- RESET\_MECHANISM >>Reset<< causes a Reset internal output message.
- The ATP\_SYSTEM [Operating] reversions are replaced by ATP\_CONTROLLER >value< reversions. To distinguish between two possible ATP\_CONTROLLER >value< reversion origins the ancestor ATP\_CONTROLLER >value< is given the label init.

## Design

The DBT that is used for co-modeling is still created by applying design decisions to the MBT. The design decisions that are made, however, now also impact upon the creation of a virtual environment which is modeled in the Modelica modeling language. The DBT interacts with this virtual environment at key points which are indicated by placing a (M) in the tag of relevant nodes in the DBT. We will now discuss the design decisions that were applied to the MBT of the ATP system.

We begin the design of the ATP system begins by determining the location of the System-Environment boundary. The Train, Signal, the individual Sensors, Driver's cab, Reset mechanism, Boundary subsystem and Noise components are outside the boundaries of the DBT. The ATP system does not control these components and is unable to determine the state of these components directly.

Figure 6.9 shows the partially completed DBT after the System-Environment boundary is resolved. The following actions were applied to the MBT:

- The Train component only consists of external input, so these are reassigned to the system component with a refinement traceability status. The environmental threads

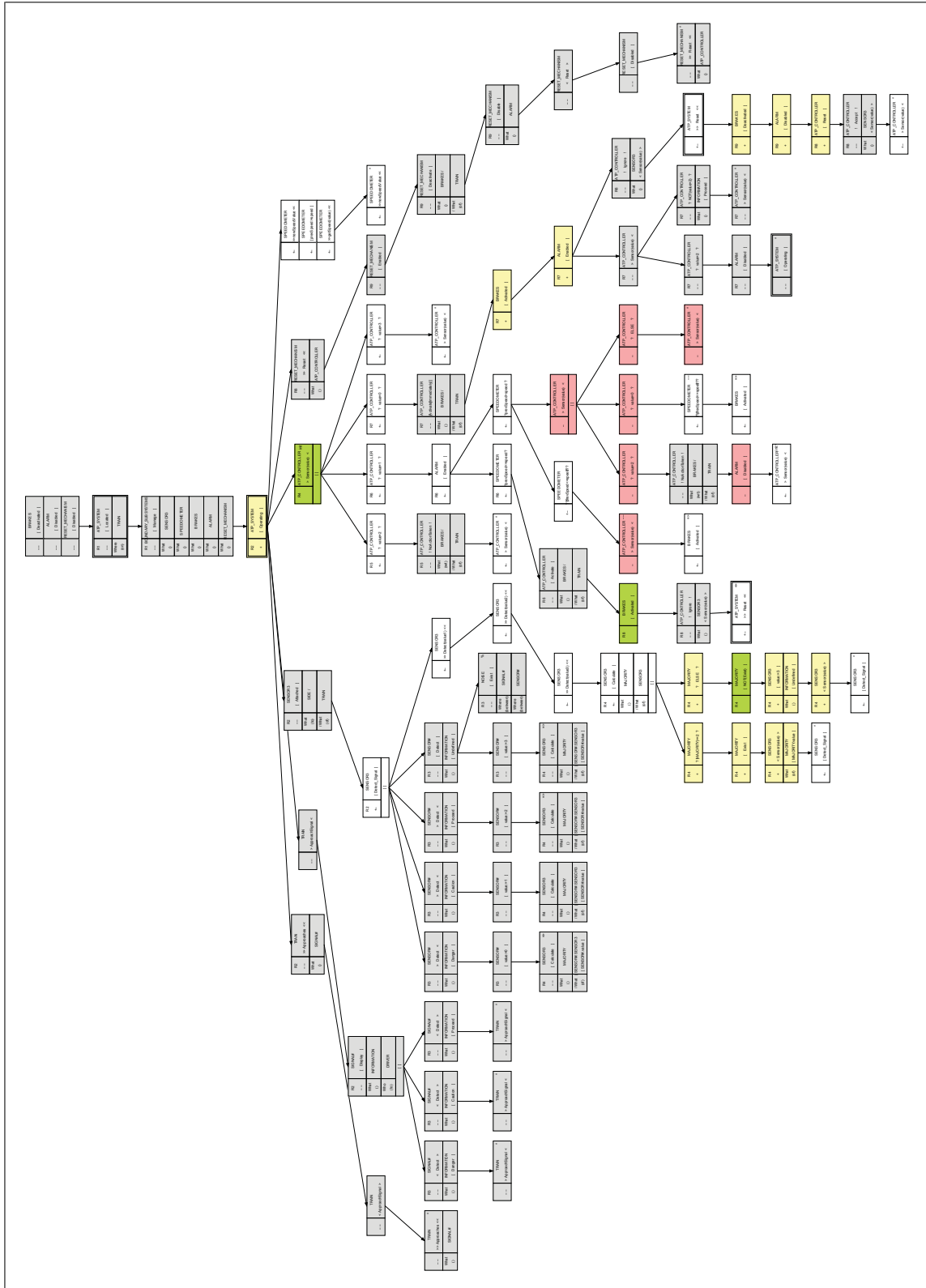


FIGURE 6.9: Partial Design Behavior Tree of the ATP System

of the Train component are given a deleted traceability status.

- The Signal component only consists of behavior. This behavior is detected by the Sensor components, so the nodes of the Signal component are given a deleted traceability status. The detect message of the Sensor components also becomes external input. The ApproachSignal external input is no longer required and is also given a deleted traceability status.
- The individual Sensor components consist of both behavior and external input. The behavior is not required elsewhere in the DBT so it may be removed. The external input is reassigned to the Sensor group component, which now individually receives each of the sensor values from the three sensors.
- The Reset mechanism consists of both behavior and external input. The external input is reassigned to the system component. The behavior of the Reset mechanism is not required elsewhere in the DBT, the alarm can be disabled and brakes can be deactivated after the reset message is received by the system component.
- A Speedometer component is required to receive the train's speed and store the previous value so that changes in the speed of the train may be determined. Atomic composition is required for when the speedometer component's speed value is updated. This is because for a small period of time the current speed equals the previous speed causing the `prevSpeed<=speed` guard to evaluate to true, regardless of the new speed value.

The DBT is completed by resolving the System-Component boundary and the initial states of the ATP. The finalised DBT is shown in Figure 6.10. The following actions were taken to finalise the DBT:

- The SENSOR<sup>3</sup> group component is renamed to SENSOR as the individual sensors are no longer captured in the DBT.
- The behaviors of the ATP\_CONTROLLER and ATP\_SYSTEM are combined to form the system component and is renamed ATP.

- The branch after the ATP receives the majority value of the sensors is simplified by combining ATP ?value=2? and ATP ?value=3? into ATP ?ELSE?.
- A DpyCT is created with an ATP[Operating] starting node and the BRAKES [Deactivated], ALARM[Disabled], SPEEDOMETER[prevSpeed:=0], SPEEDOMETER[speed:=0] initial component states.

### 6.2.2 Building a Virtual Environment

Modelica facilitates co-modeling by capturing a virtual environment that is used to investigate software/hardware interactions in different scenarios. Modelica is ideal to capture this virtual environment as it is envisioned as the major next generation language for modeling and simulation of applications composed of complex physical systems. The equation-based, object-oriented and component-based properties allow easy reuse and configuration of model components without manual reprogramming in contrast to today's widespread technology, which is mostly block/flow-oriented modeling or involves hand-programming.

The virtual environment is captured in Modelica using an object-oriented component-based approach ([Fri04], p383). The virtual environment is built from a combination of information specified in the MBT, interactions with the DBT and domain knowledge of the hardware components and the environment. This information is used to build the Modelica model in three stages which involve: (1) identifying components by investigating the domain and locating potential sensors and actuators; (2) defining the integration of the components; and (3) defining the components themselves.

Building of the Modelica model begins by identifying the components that will compose the virtual environment. The BE specification can be leveraged for this task because in addition to software components, it also specifies hardware components and the environment in which the system will be deployed. This is done by adding the environmental components that are removed when creating a BE design into the Modelica virtual environment. The Modelica model can also be used to define the behavior of the sensors and actuators that operate on the system/environment and system/component boundaries of the BE design.



FIGURE 6.10: Design Behavior Tree of the ATP System (with deleted nodes hidden)



Finally, domain knowledge is used to identify any hardware components or details of the environment that have not been described in the BE specification or design. During this stage, it is important to ensure that abstractions of the virtual environment are defined at a level of complexity that maintains a balance between realism and simulation speed.

Once the components that form the virtual environment have been identified, the integration of the components is defined. Component integration in the virtual environment is implemented in the Modelica model using connectors which link the external ports of the components together. As with the identification of components, information in the BE specification can be leveraged so that it specifies how hardware components and the environment integrate. This information is supplemented by domain knowledge of the hardware components being used as well as the environment the system is being deployed in. Software/hardware partitioning is also captured in the virtual environment during component integration by integrating a component representing the software system.

The final stage of building the virtual environment is to define the internal behavior of each of the components. Often this will just involve defining several parameters, ports, and equations to specify the behavior of the component. If the behavior of the component is complex, however, it may be beneficial to decompose the component into another integrated set of components. Another possibility for defining the behavior of complex components is to reuse existing components from the Modelica Component Library [Mod09].

One possible design of the virtual environment of the ATP system is shown in Figure 6.11. The figure shows how the ATP system represented by a BE design interacts with the virtual environment using several hardware components from the BE specification. Interactions between the components are implemented by connectors which link the external ports of the components together.

The code of the train component is shown in Listing 6.1 to demonstrate how a component is defined in the virtual environment. The train component consists of parameters, variables, connectors and equations. Parameters define the characteristics of the component that can be modified when investigating co-modeling scenarios (e.g. *m*). Variables define local values belonging to the component (e.g. *maxFa*). Connectors define values shared between components through their external ports (e.g. *fa*). Each of these types of stored values are

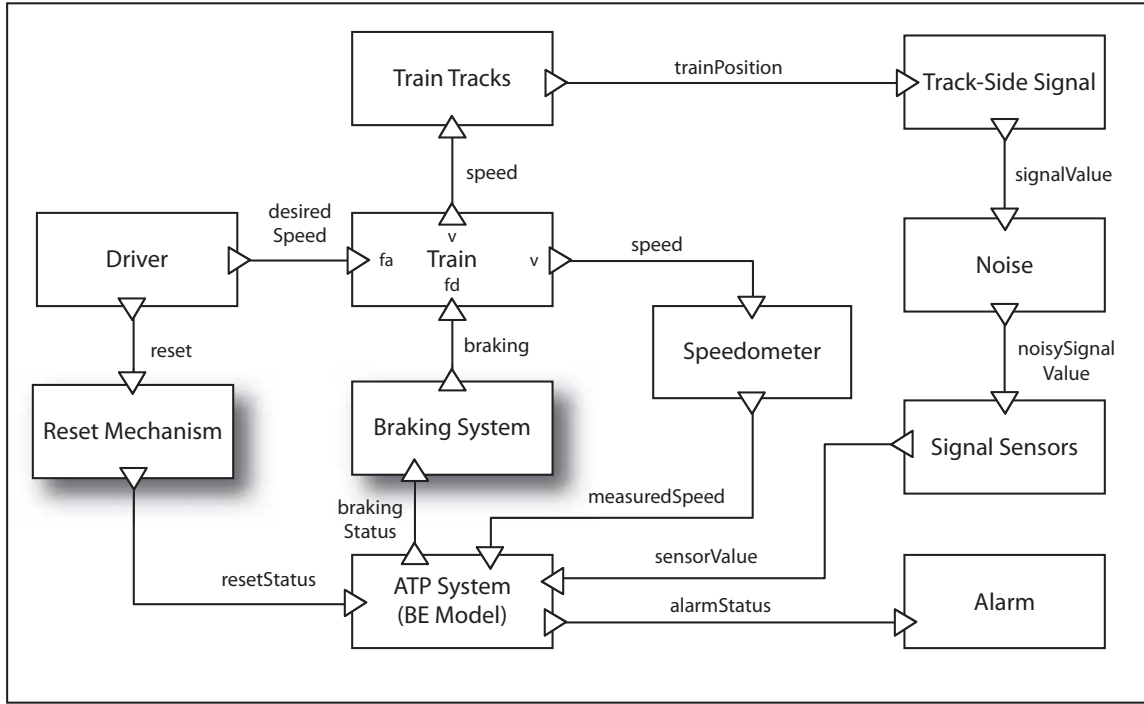


FIGURE 6.11: Component Diagram of the Modelica model of the ATP System

used to define equations that describe the behavior of the component.

```

model Train
  parameter Real m(unit='kg') 'Mass of the train';
  parameter Real maximumSpeed(unit='m/s') 'Maximum speed of the train';
  parameter Real maximumEnginePower(unit='W') 'Maximum Engine Power';
  parameter Real engineEfficiency(unit='%') 'Engine Efficiency in %';
  Real maxFa 'Maximum Accelerating Force at current velocity';
  Real limFa 'Accelerating Force limited by Maximum Accelerating Force';
  Real limFd 'Decelerating Force limited by velocity < 0';
  ForceSignal fa 'Accelerating Force, connector';
  ForceSignal fd 'Decelerating Force, connector';
  VelocitySignal v 'Train Speed, connector';
equation
  m * der(v.val) = limFa - limFd; // F=ma
  maximumEnginePower / (engineEfficiency / 100) = maxFa / v.val; // P=F/v
  limFa = if (maxFa < fa.val) then maxFa else fa.val; // Limit fa
  limFd = if (v.val <= 0) then 0 else fd.val; // Limit fd
end Train;

```

Listing 6.1: The Train Component of the Modelica Model

### 6.2.3 Implementing Software/Hardware Interactions

The final stage of constructing the co-model is to link the virtual environment built in Modelica with the software design defined by the DBT of the BE model. Our co-modeling approach defines the software/hardware interactions in the Modelica model using C external functions. This approach has two implications. Firstly, BE components must be defined in C++ to allow them to be executed externally by Modelica. Secondly, Modelica manages all the interactions between the BE model and the Modelica model. In the future, it may be possible to increase the flexibility when co-modeling using Modelica and BE by creating a BT Modelica library to allow BE models to be executed natively in Modelica.

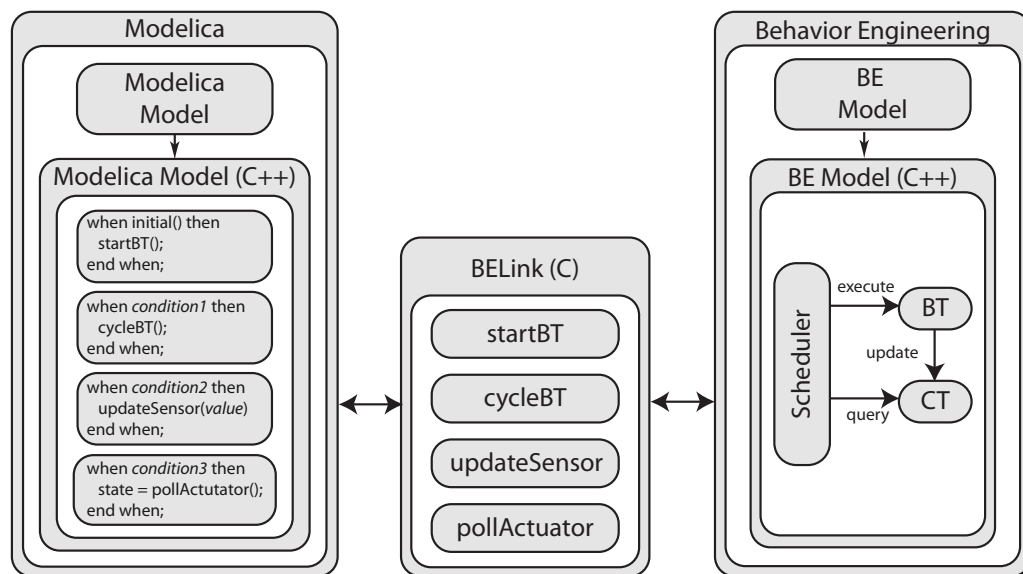


FIGURE 6.12: Interactions between Modelica and BE Models

Our current co-modeling approach which is shown in Figure 6.12 implements software/hardware interactions in three parts: (1) the compiled Modelica model (C++); (2) the BELink (C); and (3) the BE model (C++). The compiled modelica model initiates interactions with the BE model by running external functions in a C source file called BELink. BELink is responsible for mapping the function calls from Modelica to interact with the BE Model. Interactions between BELink and the BE Model are used to initiate and cycle the BRE scheduler; to send messages to the external inputs in the BT; and to query the state of components in the DynCT.

BELink consists of functions which are automatically generated from the DBT using Eclipse Java Emitter Templates (JET). Each instance of BELink consists of the functions *startBT()* to begin execution of the BE model and *cycleBT()* to execute one cycle of the BE scheduler. The JET code generation template also creates a function for each BT node in the DBT with a tag marked (M). Each unique external input node generates an *updateSensor()* function and each state realisation node generates a *pollActuator()* function. Each function is annotated with the name and behavior of the node to distinguish each updateSensor function and pollActuator function. For example, the node ATP >>reset<< causes the generation of the function *updateATP\_reset()* and the node SENSOR >>detect(value1)<< causes the generation of the function *updateSENSOR\_detect\_value1()*. Similarly, the node BRAKES[Activated] causes the generation of the function *pollBRAKES\_Activated()* and the node ALARM[Disabled] causes the generation of the function *pollALARM\_Disabled()*.

The cycleBT, updateSensor, and pollActuator functions can be used to simulate the behavior of three types of interactions. The cycleBT function can be used to simulate the frequency of the execution of the BE model relative to the Modelica model to simulate the performance capabilities on which the BE model will be deployed. The updateSensor functions can be used to simulate the sampling frequency of the sensors by defining the frequency of interactions between the Modelica model and the BE model. The pollActuator functions can similarly be used to simulate the response time of actuators.

The interactions between the Modelica model and the BE model are defined in the Modelica model. The definition of these interactions is quite flexible and they may be either periodic, aperiodic or linked to the state of the virtual environment. Listings 6.2-6.4 provide an example of each of these three types of interactions defined in Modelica. The periodic interaction demonstrates the BT scheduler being cycled once every 10 milliseconds using the cycleBT function. The aperiodic interaction demonstrates the Sensor component being sent the detect message at irregular intervals. The final interaction demonstrates the ATP component being sent a reset message when the simulation has been executing for 250 seconds.

---

```

    discrete Boolean periodic = sample(0.01,0.01);
equation
    when(periodic)
        cycleBT();
    end when;

```

---

Listing 6.2: A Periodic Interaction

---

```

Real[5] aperiodic = {0.01, 0.023, 0.029, 0.37, 0.45}
Real returnValue;
equation
    returnValue = if time = aperiodic[1] then
        updateSENSOR_detect_value1(SignalSensors.sensor1reading);
    elseif time = aperiodic[2] then
        updateSENSOR_detect_value1(SignalSensors.sensor1reading);
    elseif time = aperiodic[3] then
        updateSENSOR_detect_value1(SignalSensors.sensor1reading);
    elseif time = aperiodic[4] then
        updateSENSOR_detect_value1(SignalSensors.sensor1reading);
    elseif time = aperiodic[5] then
        updateSENSOR_detect_value1(SignalSensors.sensor1reading);
    else
        0;
    endif;

```

---

Listing 6.3: An Aperiodic Interaction

---

```

Real returnValue;
equation
    returnValue = if time > 250 then updateATP_Reset() else 0;

```

---

Listing 6.4: Interaction that is linked to the state of the Virtual Environment

## 6.3 Exploring the Possibilities

The BE and Modelica models can now be integrated to form a co-model which is used to investigate and document numerous co-modeling scenarios. A co-modeling scenario is

an instance of the virtual environment that defines one simulation of the environmental components using a particular software/hardware partition. The instance of the virtual environment is created by populating the parameters of the Modelica components and defining the behavior they exhibit during the simulation. The software/hardware partition is defined by adding the quantified and temporal information to the software and hardware interactions that occur between the Modelica and BE models.

The main purpose of simulating the co-model is to investigate the behavior of different software/hardware partitions in various environmental conditions. This may involve modeling the specifications of available sensors and actuators, or determining the quantified and temporal information for a sensor or actuator that needs to be developed. The simulation of the co-model is controlled by Modelica which provides plots that graphically show the results of the interactions between the software and hardware components in reference to time. This allows the investigation and documentation of co-modeling scenarios in a clear way.

A simple co-modeling scenario should be defined when first simulating the co-model to verify the resulting behavior is as expected. The scenario we defined has the driver gradually accelerating the train to 50 km/h, and then reducing the train's speed in response to a caution signal. The next signal indicates danger, causing the ATP controller to automatically activate the brakes. After the hazard has been removed, the driver resets the ATP controller and the train continues on.

This simple scenario already includes several assumptions about the virtual environment and the partitioning of the software and hardware components. The train model is based on a British Rail Class 57 diesel locomotive, which has a mass of 120 tonnes, a maximum speed of 120.7 km/h, a maximum brake force of 80 tonnes and a power at rail of 1860 kW with an assumed 80% efficiency due to losses in pressure and friction. It is also assumed that the signal sensors and the speedometer transmit data once every second, which effectively determines that the ATP controller has one second to respond to each signal.

The result of simulating this co-modeling scenario was the discovery of a small defect in the original DBT of the BE model. The original DBT was updated to add atomic composition for the small period of time when the speedometer component's speed value is being updated.

As stated previously, if atomic composition is not used then for a small period of time the current speed equals the previous speed causing the  $\text{prevSpeed} \leq \text{speed}$  guard to evaluate to true, and the brakes to be activated regardless of the new speed value.

Figure 6.13(a) shows the simulation of this co-modeling scenario after the discovered defect has been corrected. Analysis of the figure shows the co-model is now behaving as expected. This co-model can now form a basis on which to compare the impact of changes to the virtual environment and software/hardware partitioning that are made in other co-modeling scenarios. We will now discuss these co-modeling scenarios.

Figure 6.13(b) documents how changes to the virtual environment can influence the operation of the co-model. The change shows how the addition of carriages weighing 1500 tonnes affects the train's ability to brake.

Figure 6.13(c) investigates a software/hardware partitioning by changing the brake component of the train to inferior brakes with less brake force. The results of these investigations could be used to specify the minimum required brake force and maximum load of a train using the ATP system.

Figure 6.13(d) documents the investigation of another software/hardware partitioning involving the signal sensors. Previously, the signal sensors only updated their value as they passed directly by the track-side signal. This indicates a signal that can only be detected at close range, such as an electronic transponder in-between the tracks. In this scenario, different signal sensors are used which can detect the signals from 50m away using cameras. This results in the signals being detected a few seconds earlier allowing the ATP controller to react earlier.

Figure 6.13(e) is an example of a change in the software/hardware partitioning that can have a significant impact on the behavior of the system. In the previous scenarios, the modelled speedometer component was very precise and could detect minute changes in the train's speed. In this scenario, we used a speedometer that can only detect changes of 1 m/s. The result is that when the caution signal is detected, the train is not decreasing quickly enough to be detected by the speedometer, resulting in the train's brakes being activated.

Figure 6.13(f) shows the result of changing to a hardware platform on which the software executes five times slower. The result is an increased delay of three seconds in the time taken

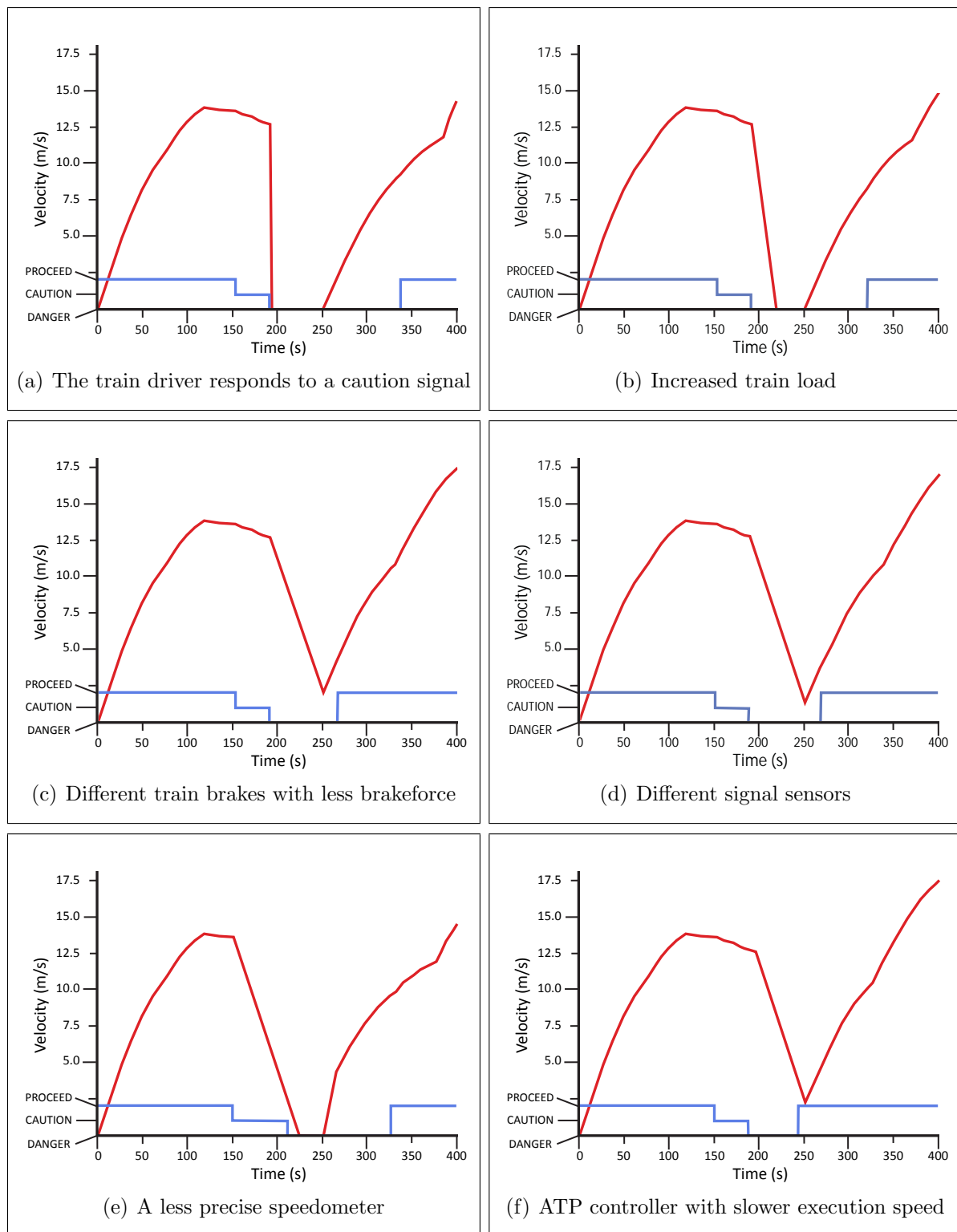


FIGURE 6.13: Simulation of several co-modeling scenarios using the co-model of the ATP System



to activate the train's brakes in response to a danger signal.

## 6.4 Towards Co-modeling of Cyber-Physical Systems

Co-modeling can assist with the task of decomposing a large-scale system into smaller decoupled systems with partitioned software and hardware components. Also, as both BE and Modelica have been used independently to model large-scale systems [Fri04][Pow07] their combination for co-modeling may also be able to handle large-scale systems.

The development of a large-scale system is often a task which is too large to be undertaken by a single company. Large-scale systems often begin with a tender, where clients provide requirements in natural language describing the needs of the system they want developed. System integrators investigate the requirements and estimate the time and cost of the project. The system integrator that succeeds in the tender brings in sub-contractors to develop portions of the complete system. The final stage of development involves integrating the systems developed by the sub-contractors to form the completed large-scale systems. In order to assure integration occurs smoothly, it is necessary to ensure that the system boundaries are both carefully defined at the early stages of development and adhered to during the development of the sub-contracted systems.

The co-modeling process can be used to investigate different system boundaries and different software/hardware partitions. Each system that is sub-contracted is described by a BE model and a Modelica model. The system developed by the sub-contractors must conform to the BE model and integrate with the Modelica model.

Large-scale co-modeling would potentially have several benefits. The systems that integrate to form the large-scale system could be developed independently and concurrently with hardware/software partitions and system boundaries that are most suited to the system. In addition, if any system boundaries or hardware/software partitions needed to be changed during development, the new boundaries/partitions could quickly be provided to all affected sub-contractors. Finally, if the system conforms to the BE and Modelica models, there is a much lower risk of problems occurring during integration. This risk can be further mitigated

by using Modelica models to simulate some components of the system and integrating sub-contracted systems one at a time to test for any incompatibilities. This plug-and-play approach to large-scale systems integration using co-modeling may be able to simplify the jobs of both sub-contractors and system integrators.

## 6.5 Discussion

The integration of BE and Modelica described in this chapter could be improved by executing the BE model natively in Modelica rather than externally in C++ using external functions. This future work would involve creating a BT library for Modelica similar to the StateGraph library [OÅD05].

Another limitation of the co-modeling approach is that investigating several software/hardware configurations creates a significant burden of documentation that must detail the parameters of each configuration and the outcomes of several co-modeling scenarios. This limitation could be addressed in future work by augmenting the current co-modeling approach with the testing language, TTCN-3. The resulting three-tier co-modeling approach would utilise TTCN-3 to define parameters of different co-models and co-modeling scenarios. TTCN-3 would then perform the required simulations using the Modelica and BE models and collate the results.

Finally, the application of co-modeling to cyber-physical systems by system integrators who sub-contract out individual systems has not been validated, even with a small case study. Therefore, it is unknown whether co-modeling could be beneficial for managing system boundaries in this application.

## 6.6 Conclusion

Applying the principles of co-design at earlier stages of development using a co-modeling approach provides a way to systematically investigate how different software/hardware partitions can be used to meet a system's constraints. Different configurations of software and hardware components can be evaluated and compared in the early stages of development

when they are easier and cheaper to fix. Co-modeling also has the potential to improve our ability to develop large-scale systems by supporting their specification, decomposition, development and integration.

The contributions of this chapter are summarised below:

1. A new approach called Co-modeling is introduced which aims to address issues due to the increasing co-dependency of software and hardware in large-scale systems. Co-modeling has the following benefits:
  - (a) It provides a framework to investigate the suitability of design decisions applied to a specification consisting of a mixture of hardware and software components.
  - (b) It helps locate and resolve defects across the software/hardware boundary early in development when they are cheaper and easier to fix.
  - (c) Simulation of co-modeling scenarios provides a graphical and documentable result that can be used to determine the specifications of sensors and actuators operating in a range of environmental conditions.
2. We demonstrate our realisation of co-modeling using a multi-view modeling approach that exploits and integrates BE and Modelica. Our approach to co-modeling provides the following contributions:
  - (a) The integration of BE and Modelica is supported by the BE link which is automatically generated from the DBT of the BE model.
  - (b) A wide range of software/hardware interactions can be captured by our co-modeling approach including periodic, aperiodic and interactions caused by the state of the virtual environment.
  - (c) The approach has the potential in the future to be used to develop large-scale cyber-physical systems by aiding the modeler to manage the complexity of building a large system composed of tightly integrated software and hardware components.



## Part IV

# Reverse Modeling



# 7

## Behavior Engineering of Digital Circuits

Reverse modeling is the third type of application of MDE as categorised by Bèzivin [BBJ07]. Reverse modeling creates a model from an existing system. In this chapter, we introduce an approach to reverse modeling that uses BTs as an intermediate representation to migrate a legacy system to a new platform. The approach is demonstrated using a real-world case study from industry. The case study requires outdated circuitry implemented in transistor-transistor logic (TTL) integrated circuits (ICs) to be replicated in a field programmable gate array (FPGA). The only documentation available for this task is a circuit diagram which describes the implementation of the hardware by the integration of several TTL ICs.

Modeling the behavior of the hardware using BTs requires a domain-specific extension to capture the behavior of digital circuits. This has a number of benefits. Using this domain-specific extension to capture the legacy hardware in BTs allows the functionality of the system to be more easily understood and analysed by a wider audience than just engineers

with a hardware background. It also provides access to the existing BE toolset, including an editor, simulator, and a model-checker that can be used to perform failure modes and effects analysis (FMEA). The domain-specific BTs are used to generate VHDL, a hardware language that can synthesise a gate-level design that is deployable on an FPGA. Finally, we also transform the domain-specific BT to perform FMEA to justify the benefits of using BTs as an intermediate.

Using BTs as an intermediate to replicate the TTL functionality on an FPGA is a migratory approach. To justify the selection of a migratory approach, we first must discuss why legacy systems occur and what approaches exist to deal with legacy systems. A system becomes a legacy system when the benefits of upgrading or extending a system are outweighed by the risk involved over a long period of time. Eventually when the balance reverses, the legacy system is often in a state where: it is deployed on obsolete hardware; it has software written in an arcane language; it is described by little or no documentation; and the original developers have long left the organisation. Legacy systems are especially common in safety-critical application areas such as aviation, medicine, the military and space exploration where failure can have life-threatening consequences. They are also prevalent in information systems requiring high availability such as financial systems, reservation systems, and personnel management systems.

The consequences of failure in these areas makes the task of replacing or modifying a legacy system a high risk undertaking. Three approaches are used: wrapping, migration, and re-development [BLWG99]. A summary of the advantages and disadvantages of each of these approaches is shown in Table 7.1.

Wrapping involves adding an interface to the legacy system to increase its reusability. The previous chapters described mechanisms to enable reuse that are also suited to the wrapping approach of dealing with legacy systems. For example, a legacy embedded system can be encapsulated as a component and reused by integrating it with other components to form a new system. Similarly, co-modelling supports the integration of pre-existing hardware components which may also be legacy systems. In information systems, wrapping often takes the form of adding a graphical user interface (GUI) or web-based front-end to the legacy system. Although wrapping is widely used, it only has a short-term impact and often



Approach	Advantages	Disadvantages
Wrapping	Low Risk	Increases maintenance costs
	Makes system extendable	
Migration	New platform reduces costs and makes maintenance easier	May not be the most efficient solution
	Aims for identical functionality and traceability	Risk is increased if new functionality is added during migration
	Leads to a better understanding of the system	
Re-development	New platform built using modern techniques	High risk of failure due to the introduction of defects and functionality that differs from the legacy system
	Resulting system is much easier to extend and maintain	May become a legacy system in the future

TABLE 7.1: Three approaches to dealing with Legacy systems

increases maintenance costs rather than reduces them.

Re-development deals with legacy systems by replacing the system with a system built upon a new platform using modern techniques. Re-development is a risky approach as it is difficult to minimize the introduction of defects and to ensure the functionality of the new system and the legacy system is the same. Even if re-development is successful, it is likely the new system will itself become a legacy system in the future.

Migration represents the middle-ground for dealing with legacy systems where the legacy system is adapted to operate on a new platform in order to reduce costs and for ease of maintenance. Migration has the goal of maintaining identical functionality by keeping traceability with the legacy system. Migration also has the benefit of aiding developers to better understand the system during the migration process. However, the newly migrated system may not be the most efficient solution due to the need to maintain identical functionality and traceability to the legacy system. Also, if there is a desire to extend functionality it must wait until the newly migrated system has first been tested for conformance.

The rest of this chapter is structured as follows. The next section introduces a domain-specific extension to allow BTs to capture digital circuits. This extension is then used to

describe how a legacy system can be replicated using a migratory approach with a BT intermediate. The domain-specific BT is used for two applications. Firstly, it is used to demonstrate the benefits of a BT intermediate by using the BE specification to perform FMEA. Secondly, it is used for simulation and synthesis of an FPGA using VHDL code generated from the BE design.

## 7.1 Capturing Digital Circuits with BTs

Using BE to model digital circuits differs significantly from modeling software requirements. Software requirements are defined in natural language so it is necessary to deal with imperfect knowledge. A digital circuit, conversely, is an existing implementation which can be assumed to be verified and validated. The process of translating a digital circuit defined by a circuit diagram to another representation, however, can introduce defects. BE helps to avoid the introduction of defects by providing a scaleable methodology which enables small portions of the digital circuit to be captured using a rigorous translation process.

The difference between digital circuits and software requirements necessitates two modifications to the BML. The first modification made to the BML is to modify traceability to match the traceability information recorded in circuit diagrams. Circuit diagrams record traceability by assigning each hardware component a unique part identifier. Each hardware component also has a number of pins which are associated with sending or receiving particular signals. This traceability information is recorded in the BML by using a part identifier similar to the unique identifier that is assigned to each software requirement. If the behavior is associated with a particular pin of the hardware component, the pin number is appended to the part identifier with a period used as a delimiter. For example, U1.1 provides a traceability link to the first pin of the U1 component.

The second modification made to the BML is to enable the behavior of a digital circuit to be represented in BTs. This modification must meet two goals. Firstly, it must enable the generation of VHDL code from BTs for the synthesis of an FPGA. This is achieved by creating a domain-specific extension to BTs that captures the concepts of VHDL. Secondly, it must enable the BT representation of the digital circuit to be model checked, thereby

allowing it to be used to perform FMEA. This is achieved by mapping the VHDL domain-specific extension of BTs to the existing BT syntax.

Behavior is represented in VHDL using digital signals which consist of four values: high, low, rising and falling. Signals can be set to either high or low. Rising edges (low to high) and falling edges (high to low) are automatically generated when a signal transitions from either high to low or low to high. Most of these VHDL concepts can be mapped directly to the existing BT syntax as shown in Table 7.2.

Behavior	Behavior Tree	VHDL
Set a signal	<div>U1</div> <div>COMPONENT [signal := High]</div>	signal <= '1';
	<div>U1</div> <div>COMPONENT [signal := Low]</div>	signal <= '0';
Check a signal	<div>U1</div> <div>COMPONENT ? signal = High ?</div>	if (signal = '1') then ...
	<div>U1</div> <div>COMPONENT ? signal = Low ?</div>	if (signal = '0') then ...
Check for a signal change	<div>U1</div> <div>COMPONENT ? signal = Rising ?</div>	if (RISING_EDGE(signal)) then ...
	<div>U1</div> <div>COMPONENT ? signal = Falling ?</div>	if (FALLING_EDGE(signal)) then ...
Signal Passing	<div>U1.1</div> <div>COMPONENT &lt; U1.1(signal) &gt;</div>	Used for port mapping
	<div>U1.1</div> <div>COMPONENT &lt;&lt; U1.1(signal) &gt;&gt;</div>	
	<div>U2.1</div> <div>COMPONENT &gt; U1.1(signal2) &lt;</div>	
	<div>U2.1</div> <div>COMPONENT &gt;&gt; U1.1(signal2) &lt;&lt;</div>	

TABLE 7.2: Mapping VHDL functionality to Behavior Trees

The digital signals of VHDL can be captured in BTs using attributes which store the current value of the signal. BTs must be able to use these attributes to emulate three key functions of signals with VHDL: setting the value of a signal; checking the value of a signal; and passing the value of a signal between two or more components. Signals can be set to high or low by assigning the desired value using an attribute assignment. The value of a signal can be determined using a conditional test associated with a selection behavior type. Signal passing between components can be modeled in BTs by using parameterised message passing.

VHDL concepts that do not map directly to BTs are added with a domain-specific extension. Three VHDL concepts still need to be defined: (1) generating a rising and falling signal state; (2) receiving a signal from multiple sources; and (3) inverting a signal value. The first domain-specific extension is required for BTs to generate a rising or falling value. This is required to capture the implicit behavior of VHDL that during the transition of a signal from a low value to a high value, the signal briefly has a rising value. Similarly, when a signal transitions from a high value to a low value, the signal briefly has a falling value. Figure 7.1 shows the BT mapping to generate the rising and falling signals. The mapping ensures that the signal is transitioning from high to low or low to high before generating the rising or falling signal value. If the signal is being assigned the same value, then a reference is used to skip the generation of the rising or falling signal value.

The second domain-specific extension is required for BTs to capture the VHDL semantics for receiving a signal from multiple sources. When multiple signals are received on the same pin in a digital circuit, they are implicitly combined using exclusive or semantics. This ensures that a pin only receives a high value if all the signals it receives are high, otherwise a low value is received. A group of atomic nodes is used to indicate that multiple signals are received at the same pin. The mapping in Figure 7.2 transforms the group of atomic nodes to model the semantics for receiving multiple signals in BTs. The mapping assumes that the signals are all sent one after the other and will block if all the signals are not sent (unless a low is received prior).

The third domain-specific extension is required for BTs to capture the semantics of inverting a signal, a common function in VHDL. The invert behavior is captured in BTs as

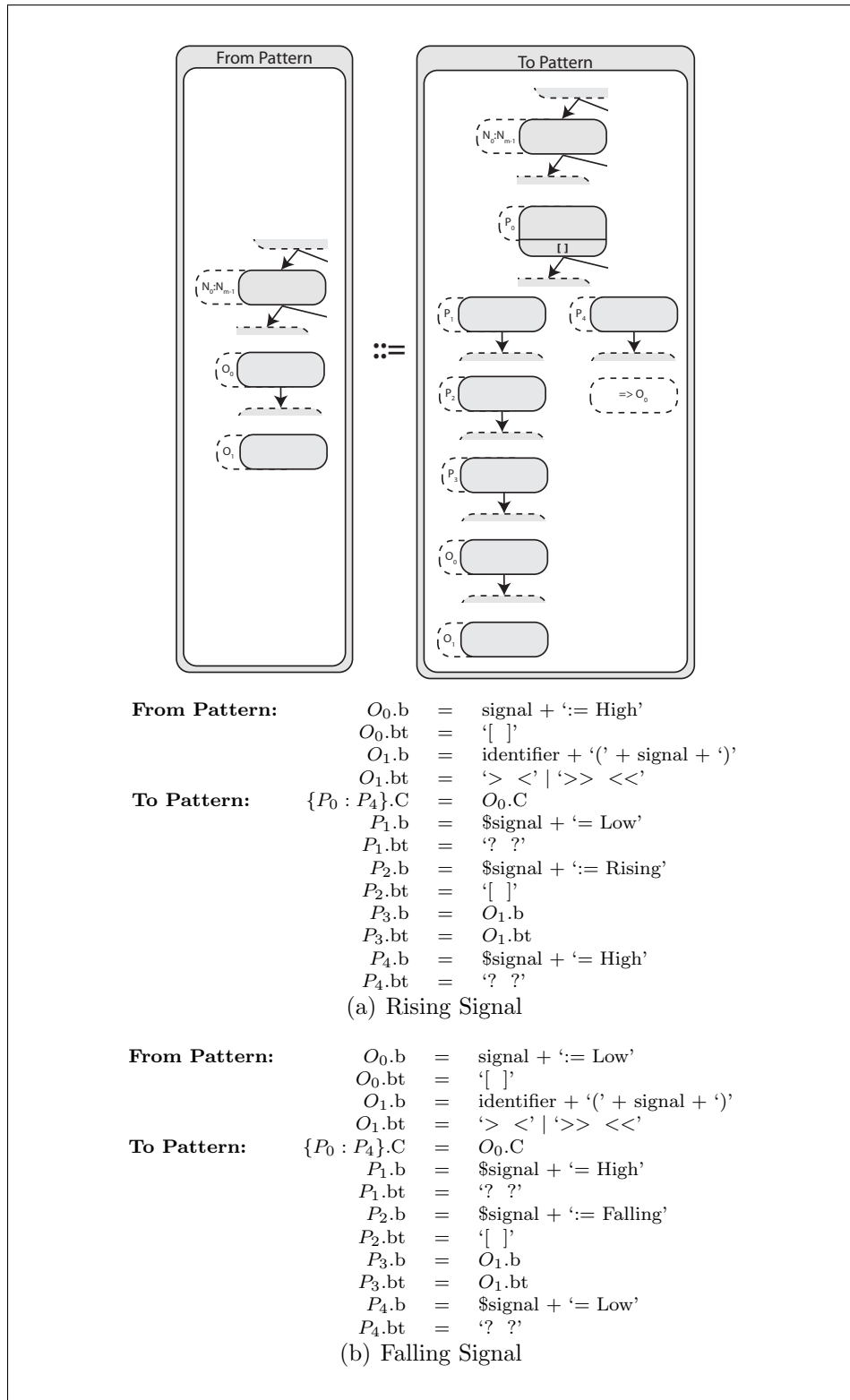
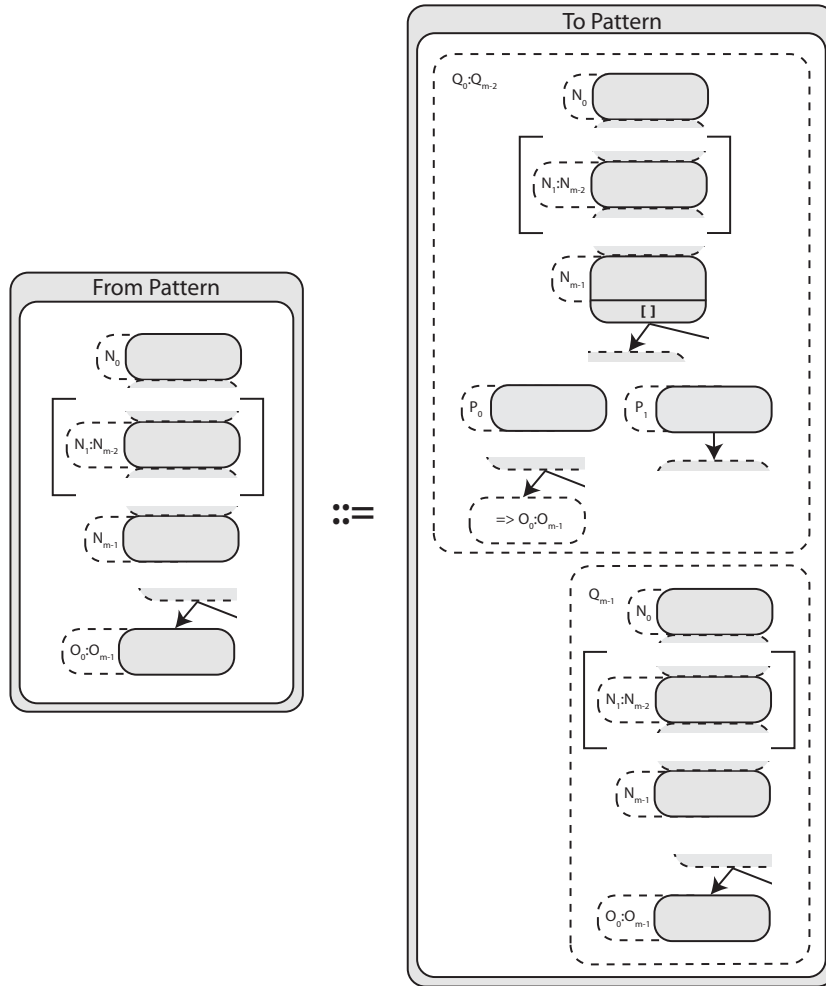


FIGURE 7.1: Mapping for Generating a Rising and Falling Signal State



**From Pattern:**

**To Pattern:**

$\{N_0:N_{m-1}\}.bt$	=	'> <'   '>> <<'
$N_0.b$	=	'(' + signal + ')'
$Q.m$	=	$N.m$
$\{Q_0:Q_{m-1}\}.\{N_0:N_{m-1}\}.op$	=	' '
$\{Q_0:Q_{m-1}\}.P_0.C$	=	$N_0.C$
$\{Q_0:Q_{m-1}\}.P_0.b$	=	\$signal + '= Low'
$\{Q_0:Q_{m-1}\}.P_0.bt$	=	? ?
$\{Q_0:Q_{m-1}\}.P_1.C$	=	$N_0.C$
$\{Q_0:Q_{m-1}\}.P_1.b$	=	'ELSE'
$\{Q_0:Q_{m-1}\}.P_1.bt$	=	? ?

FIGURE 7.2: Mapping for Receiving a Signal from Multiple Sources

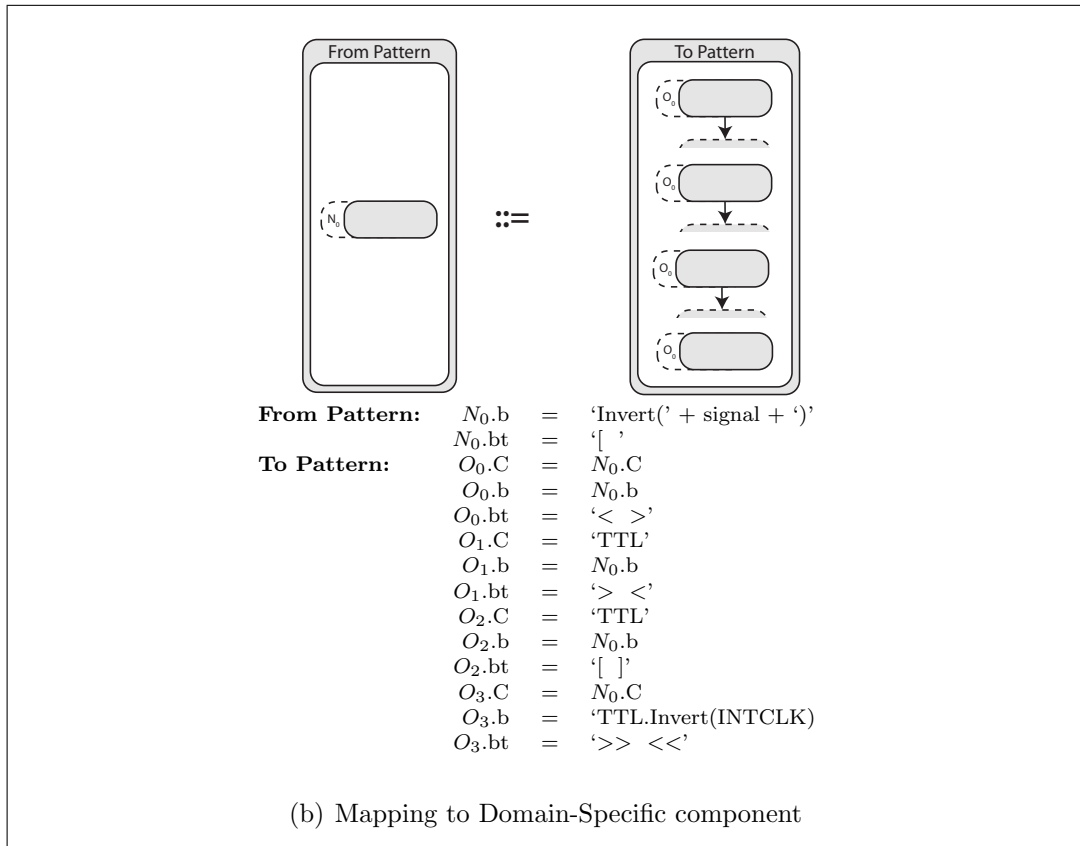
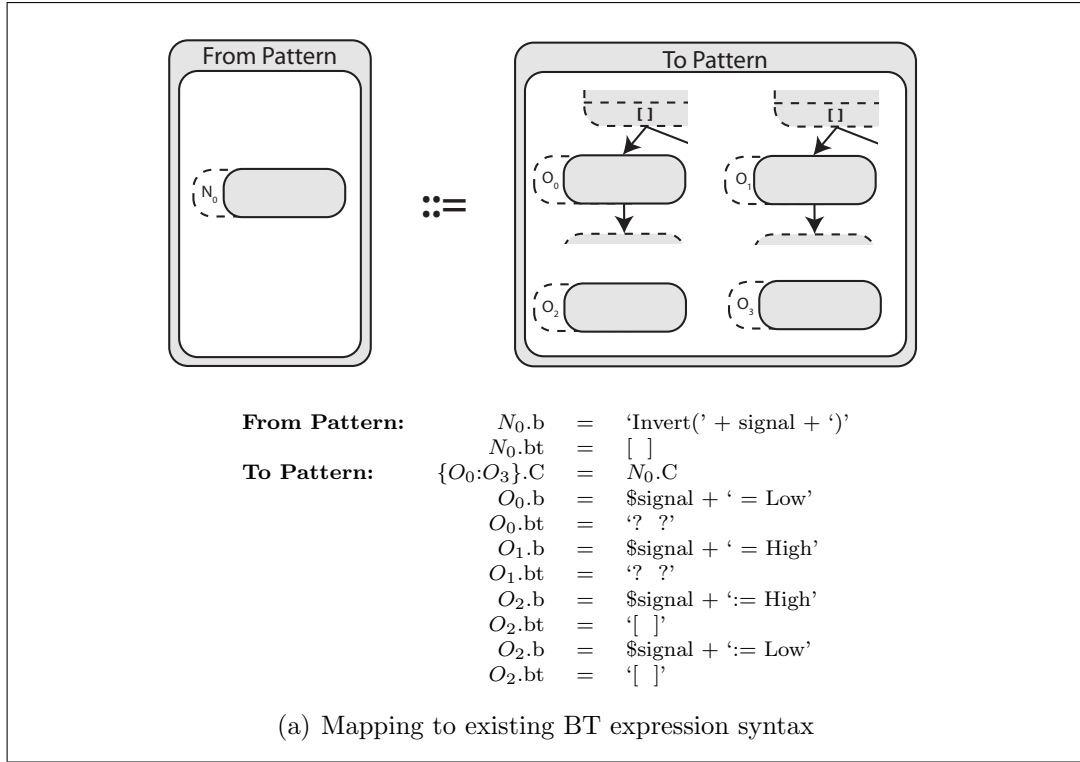


FIGURE 7.3: Two Mappings for Inverting a Signal

a state realisation. Figure 7.3 shows two mappings for capturing the semantics of the invert behavior using a domain-specific extension. Mapping to a domain-specific component is not suitable in this case because the model-checker cannot execute encapsulated computation. By mapping to the existing BT expression syntax using condition checks and attribute assignment, however, the invert behavior can be model-checked.

## 7.2 Replicating Legacy Hardware with BE

In this section, we demonstrate how the VHDL domain-specific extension can be used to capture digital circuits in the field of avionics. The case study involves a real-time system that is currently in service. The authorised maintenance organisation found that the maintenance of the system was becoming increasingly difficult due to the use of outdated hardware components. A particular concern was a timing unit which forms the central communication link between the computer systems operating on a multiplexed 1553b bus and interfaced hardware of the avionics system operating on a non-multiplexed 1553 bus [Dep87]. Maintenance of the timing unit was becoming increasingly difficult as its circuitry contained Transistor-Transistor Logic <sup>1</sup> (TTL) components which were no longer manufactured. Repairs of the timing unit were also time consuming due to the time needed to locate the cause of a failure from amongst a large number of discrete components. To deal with the increasing costs of maintenance, it was decided to replicate the functionality of the A2 controller card of the timing unit on an FPGA. FPGAs are customisable ICs used to deploy digital circuits that are defined in a hardware definition language. In comparison to TTL ICs, FPGAs have several benefits including a smaller size and weight, and the ability to be easily replaced or upgraded. Because of these benefits, and prior to the decision decommissioning of the system, it was planned that the outdated hardware of several circuit cards of the timing unit and other circuitry of the real-time system would be replicated on FPGAs.

The maintenance organisation contracted a third party to re-develop the A2 controller card on an FPGA. At the same time, the maintenance organisation also wanted to investigate

---

<sup>1</sup>For a brief introduction to Transistor-Transistor Logic refer to Appendix C.



To demonstrate this approach, we use a small case study consisting of a subsystem that is part of the A2 controller card. The A2 controller card is represented using a circuit diagram, with the portion used for this case study shown in Figure 7.4<sup>2</sup>. The migration process utilizes the BMP by considering each small indivisible portion of the hardware circuit as a verified and validated requirement. This strategy allows the migration process to follow the BMP steps of formalisation, fitness for purpose test, specification and design using the domain-specific extension to the BT representation. This allows BE to be used both to: (1) perform FMEA using the MBT resulting from the specification stage; and (2) generate VHDL using the DBT resulting from the design stage.

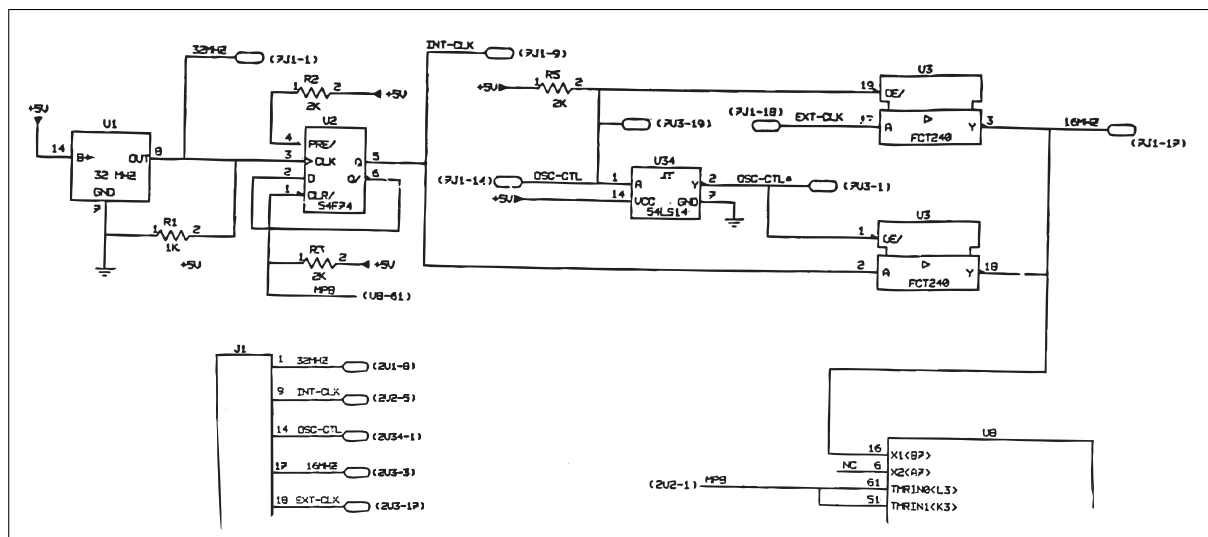


FIGURE 7.4: The Sub-System of the A2 Controller Card

<sup>2</sup>To view this figure in more detail, please refer to the electronic version included as part of the supplementary material.

### 7.2.1 Translating a Hardware Requirement

When applied to digital circuits, the formalisation stage of the BMP does not actually formalise digital circuits. The hardware requirements have already been deployed and can therefore be assumed to be verified and validated. Instead the first stage of the BMP is used

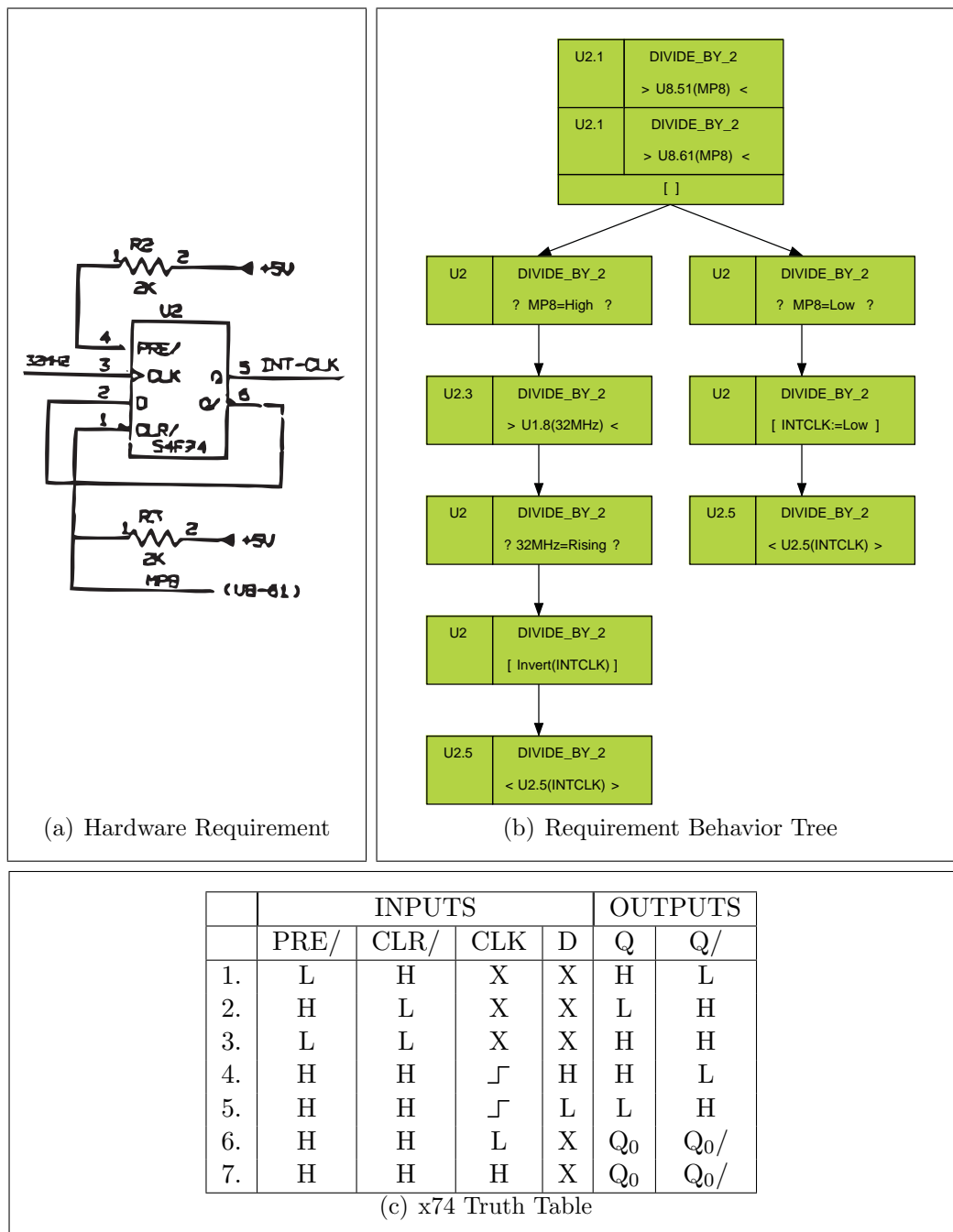


FIGURE 7.5: Translating a Hardware Requirement

to translate small indivisible portions of the circuit diagram, which we refer to as hardware requirements, into RBTs. Figure 7.5 shows the result of translating a hardware requirement extracted from the hardware subsystem (see Figure 7.4) into an RBT by using a truth table which describes the logic of the TTL IC. In the remainder of this section we will describe how this hardware requirement is translated into the resulting RBT.

The hardware requirement is composed of a 54F74 TTL IC, which is a military standard (54), fast-switching (F), dual d-type positive edge triggered flip-flop with preset and clear (74). The functionality of TTL ICs is captured using truth tables which describe the values of the digital output resulting from each possible combination of the values of the digital input. Symbols are used in the truth tables to describe the state of the digital signal: L is a digital low; H is a digital high; X can be either low or high;  $\lceil$  is a rising edge;  $\rfloor$  is a falling edge; and a subscript zero before a signal name (e.g.  $Q_0$ ) is the value of the signal before the inputs changed. A naming convention for signals is to include a forward slash (/) after the name of an input or output to indicate it is inverted. For example, when signal Q is high, signal Q/ is low, and when signal Q is low, signal Q/ is high. The functionality of the x74 series of TTL ICs is described by the truth table in Figure 7.5(c)<sup>3</sup>. The x74 series of TTL ICs has seven possible combinations of the values of the input and output signals.

The wiring that connects the pins of the 54F74 TTL IC determines which combinations of the input and output values described in the truth table can occur in the hardware requirement. The configuration of the 54F74 TTL IC is shown in Figure 7.5(a) where the signals (MP8, 32MHz, +5V, and INTCLK) are connected to the pins of the IC (PRE/, CLK, D, CLR/, Q, and Q/). While analysing each signal, we indicate which combinations of input and output values are affected in the truth table by surrounding the line number by a parenthesis e.g. (1).

Let us begin the analysis with the +5V signal received by the PRE/ input pin. The PRE/ input pin is held high by the +5V signal passed through the 2k $\Omega$  R2 resistor, so the two combinations where PRE/ is low (1 and 3) cannot occur in the configuration of this hardware requirement.

---

<sup>3</sup>The third line of the truth table is a nonstable state because both Q and Q/ are outputting high. As soon as either PRE/ or CLR/ go high the output of Q or Q/ will change.

The CLR/ input is also held high by a +5V signal over the  $2k\Omega$  R3 resistor, however, the CLR/ input is also connected to the MP8 signal. R3 is a pull-up resistor that ensures when the MP8 signal is high, the voltage is pulled up to +5V. This reduces noise at the CLR/ input, which could cause the wrong signal value to be detected. When the MP8 signal is low, however, the lower resistance of this signal path still results in a digital low at the CLR/ input. This results in two possibilities. If the CLR/ input is low (2), then the value of the Q output will be low. Alternatively, if CLR/ is high (4-7), then the value of the Q output depends upon the CLK input.

The CLK input is connected to a 32MHz signal that oscillates from digital low to high and back to low once every  $1/3200000$  seconds. When the CLK input is low (6) or high (7), then the value of the Q output remains unchanged. On the rising edge at the CLK input (4 and 5), the value of the Q output depends on the D input.

In the hardware requirement, the D input is connected to the output of Q/. This connection inverts the Q output whenever there is a rising edge at the CLK input. The reasoning for this is as follows. If the output value of Q/ is high and the CLK does not have a rising edge, then the input value of D is also high. Because the value of Q/ is high, the value of Q must currently be low. When there is a rising edge at the CLK input and D is high (4), the value of Q changes to high<sup>4</sup>. Similarly, when the value of Q/ and D are low, the value of Q must currently be high. When there is a rising edge at the CLK input and D is low (5), the value of Q changes to low.

In summary, two behaviors can be observed at the INTCLK signal connected to the Q output pin. When the MP8 signal is low, the INTCLK signal is low. When the MP8 signal is high, the value of the INTCLK signal inverts on the rising edge of the 32MHz signal. This gives the INTCLK signal a frequency equal to half of the 32MHz signal as shown in Figure 7.6. Therefore in this hardware requirement, when the MP8 signal is high, the 54F74 is configured to output the INTCLK signal with a 16MHz frequency derived from the 32MHz signal.

Now that the hardware requirement is understood, it can be captured in an RBT. Because

---

<sup>4</sup>This will also cause Q/ to change to low and subsequently cause D to change to low. The signal values are buffered, however, so this change is not acted upon until the next CLK rising edge.

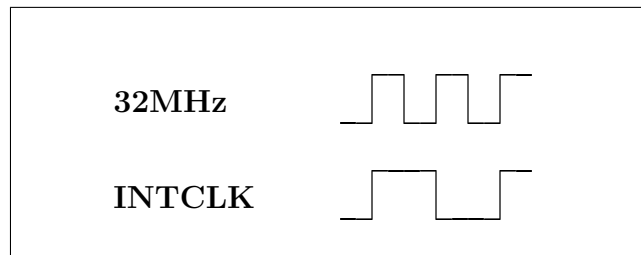


FIGURE 7.6: Comparison of INTCLK and 32MHz signals

the tag of the hardware component, U3, is used for traceability, a component name is chosen to reflect the functionality of the component. The component is given the name `DIVIDE_BY_2` to reflect the behavior that the generated INTCLK output signal is half the frequency of the 32MHz input signal. The PRE/ input pin is always held high, so it does not exhibit any behavior that needs to be captured by the RBT. Therefore, the RBT begins with the input of the MP8 signal on the CLR/ input pin. Tracing the MP8 signal back to the U8 component (see Figure 7.4) reveals there are two sources for the signal, U8.51 and U8.61. This dual source is captured using the notation introduced for signal passing from multiple sources. The behavior of the first node, U8.51(MP8), describes a message containing the MP8 signal which originates from pin 51 of component U8. The traceability link of the node also indicates that the message is recieved by first pin of component U2, the CLR/ input pin of the 54F74 IC.

The subsequent behavior depends on whether the value of the MP8 signal is high or low. This is represented using two alternative branches. Alternative branching indicates that two branches are mutually exclusive<sup>5</sup>; the MP8 signal cannot have a value of both high and low at the same time. The first branch of behavior occurs if the MP8 signal is low, which is determined using a node with a selection behavior type. If this condition is true, the INTCLK attribute is set to low. Because the behavior of these nodes occurs inside the 54F74 IC, they are both given a traceability of U2 without a pin number. The first branch of behavior is finished by outputting the INTCLK signal from the fifth pin of component U2, the Q output pin of the 54F74 IC. The label U2.5 is used for both the message identifier and the traceability link to record the origin of the signal.

<sup>5</sup>Alternative branching is also utilised for VHDL code generation from the BT in Section 7.4.1.

The second branch of behavior occurs if the MP8 signal is high. From our previous analysis, we know that the subsequent behavior requires the 32MHz signal. The 32MHz signal originates from pin eight of component U1, so the behavior of the internal input is U1.8(32MHz). The 32MHz signal is input on pin 3 of component U2, the CLK input pin of the 54F74 IC. The hardware requirement only exhibits behavior on the rising edge of the 32MHz signal, so a condition test ensures the 32MHz signal is rising. When a rising edge is detected, the INTCLK signal is inverted using the Invert(INTCLK) state realisation. Once again, both of these nodes describe behavior that is internal to the 54F74 IC, so they are given a traceability of U2 without a pin number. Finally, the second branch concludes by outputting the inverted INTCLK signal from the fifth pin of component U2. This is captured by using the label U2.5 for both the message identifier and the traceability link of the node.

The partitioning of the A2 subsystem into the 17 hardware requirements is shown in Figure 7.7<sup>6</sup>. The RBTs resulting from translating each of the hardware requirements of the subsystem of the A2 controller card are shown in Figures 7.8 and 7.9.

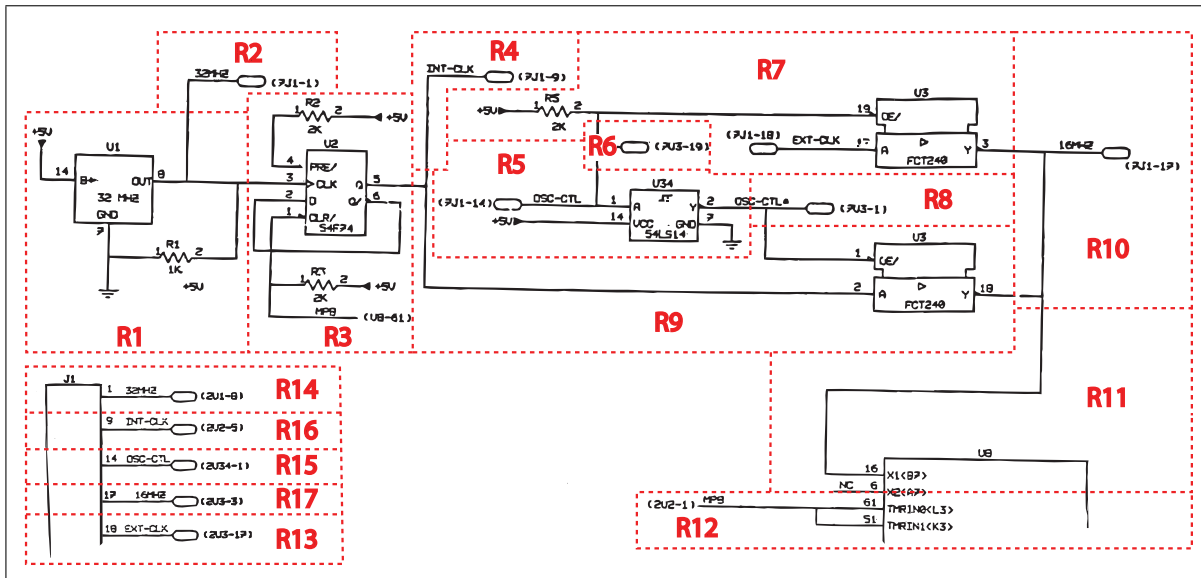


FIGURE 7.7: The A2 sub-system partitioned into the 17 hardware requirements

<sup>6</sup>To view this figure in more detail, please refer to the electronic version included as part of the supplementary material.

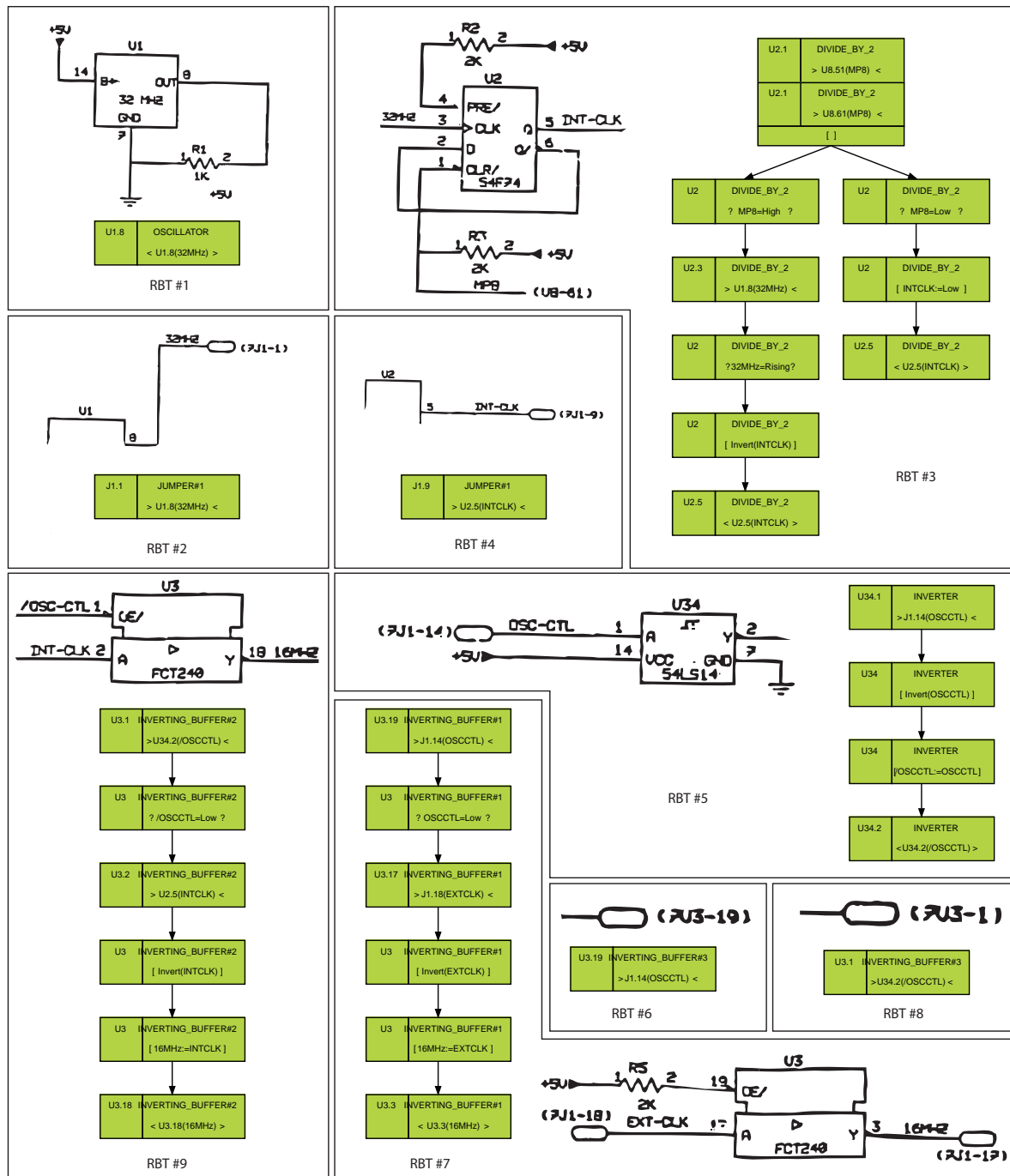


FIGURE 7.8: The Requirement Behavior Trees of the Sub-System (R1-R9)

### 7.2.2 Testing Fitness for Purpose

As with the formalisation stage, the testing fitness for purpose stage of the BMP has a different purpose when applied to translating digital circuits. The identification of

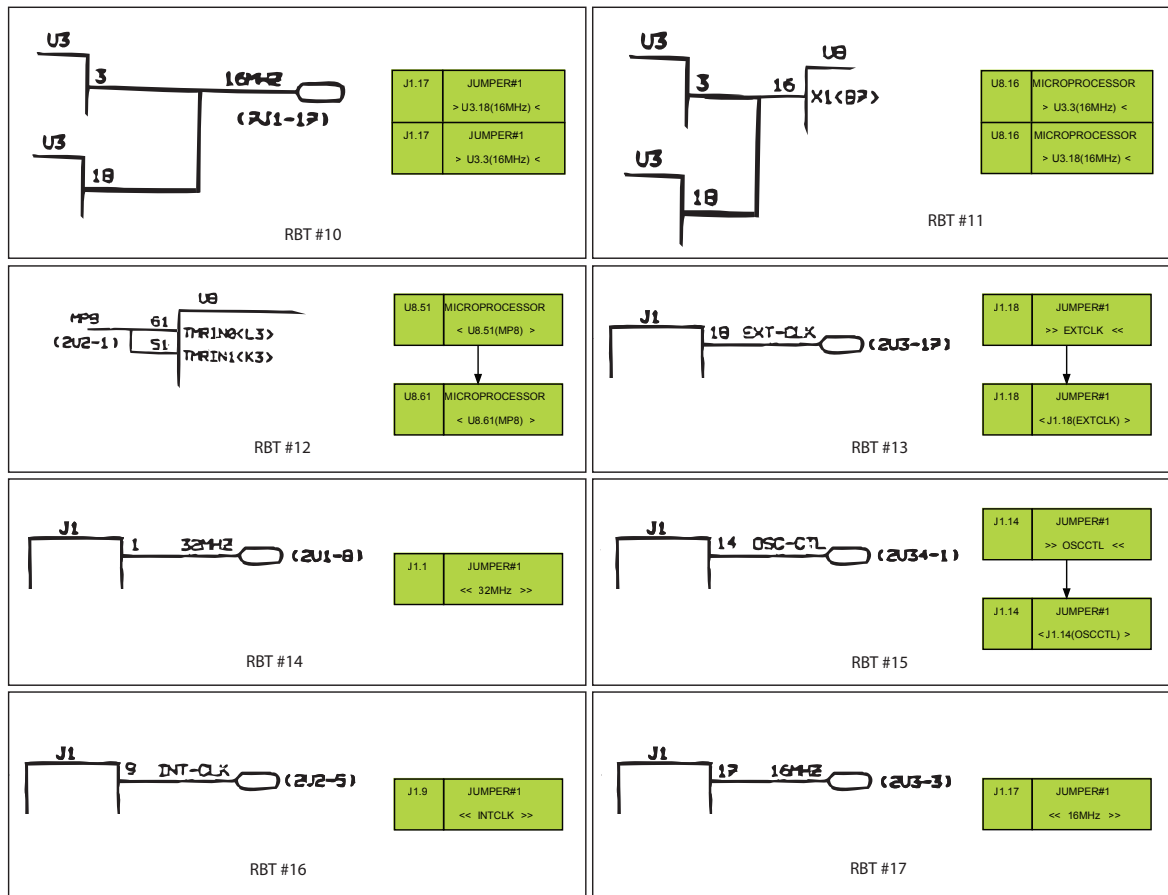


FIGURE 7.9: The Requirement Behavior Trees of the Sub-System (R10-R17)

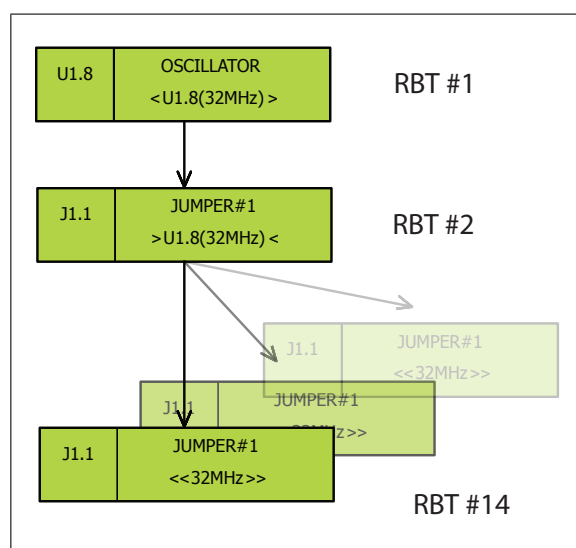


FIGURE 7.10: Integrating RBT1 with RBT2 followed by integration with RBT14



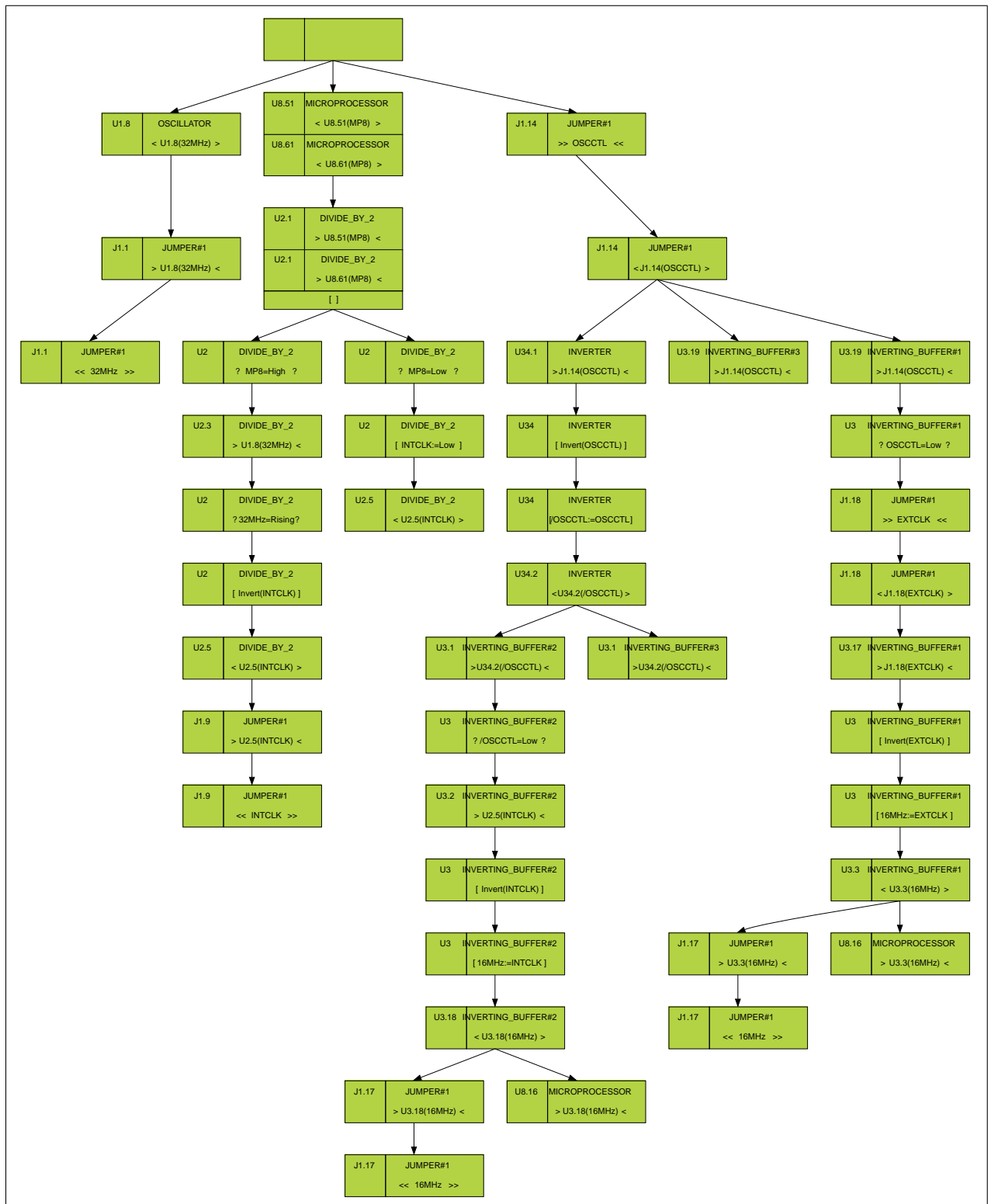


FIGURE 7.11: The Integrated Behavior Tree of the Sub-System

inconsistency and redundancy that emerges from integrating the translated requirements is not required. The stage is still useful, however, for identifying if any hardware requirements are missing, i.e. if a section of the hardware subsystem has not been translated. Integrating the hardware requirements also provides the emergent property of clarifying the intent of the subsystem.

The interaction axiom and the precondition axiom still apply to the digital circuits captured in BTs. BTs of digital circuits do have the distinction in comparison to their software counterparts that integration will always occur during message passing. This is because VHDL is a dataflow language that integrates components by exchanging signals. Figure 7.10 shows an example of integrating of RBT1 with RBT2, and the integration of the result with RBT14.

The resulting IBT is shown in Figure 7.11. Using message passing as the sole means of integration causes an issue during integration of RBTs when the message is only received under specific conditions. In this case, the two RBTs cannot be integrated together but must remain in separate branches of behavior. A complete IBT is formed by using an empty node as the root node, allowing these separate branches of behavior to be integrated. The IBT of the hardware sub-system is composed of three branches of behavior integrated by an empty root node. These three branches are a result of the integration problem which occurs when receiving messages under specific conditions. The message, U1.8(32MHz), that is sent by the Oscillator is only received by the Divide\_by\_2 if MP8 has a value of high. Similarly, the message, U2.5(INTCLK), that is sent by the Divide\_by\_2 is only received by the Inverting\_Buffer#2 if /OSCCTL has a value of low.

### 7.2.3 Specification

In contrast to software requirements it is not necessary to perform some tasks when creating a digital circuit specification because the integrated behavior tree is derived from verified and validated hardware requirements. Three of the seven tasks required for the specification stage with software requirements are not required for modeling digital circuits with BE. They are checking for requirement completeness and correctness; resolving event types; and managing

concurrency. The remaining tasks required to be performed are: decoupling components; refining causal behavior; checking component initialisation; and checking for reversion nodes. As with the previous stages, the specification is modeled using the domain-specific extension that captures VHDL concepts.

The task of decoupling moves components that are not part of the subsystem into separate environmental threads of behavior. During this task the Oscillator, Microprocessor and Jumper#1 components are moved to separate environmental threads for the following reasons. The Oscillator is part of the environment as an FPGA is unable to generate a clock signal that is not based upon the system clock. The Jumper#1 and Microprocessor components are not fully described in the subsystem. The Jumper#1 can be included in the FPGA later by translating its sub-system separately. The Microprocessor would most likely be included in the FPGA as a purchased VHDL IP core rather than by translation using BE.

The second task is to refine causal behavior. This task is facilitated by understanding the purpose of the subsystem from the partially formed MBT. The sub-system is a clock selector that consists of two threads of behavior. The first thread controls an internal clock that is generated by the Oscillator. The internal clock is generated if MP8 is high, or stopped if MP8 is low. The second thread selects between using this internal clock or using an external clock provided by Jumper#1. Annotating the MBT with a Clock\_Selector system component with states that indicate the purpose of the following segments of behavior. This allows the purpose of the subsystem to be quickly conveyed. The empty root node is replaced by Clock\_Selector [On] state realisation. The following state realisations of the Clock\_Selector are placed at appropriate places throughout the partially formed MBT: Internal\_Clock\_Ready, Generate\_Internal\_Clock, Stop\_Internal\_Clock, Select\_Clock, External\_Clock, and Internal\_Clock.

Another refinement to causal behavior is to remove the Inverter component by modifying the Inverting\_Buffer#2 component. The inverter component currently inverts the OSCCTL signal so the Inverting\_Buffer#2 can perform a condition test on the /OSCCTL signal having a low value. The Inverter component can be removed if the Inverting\_Buffer#2 performs a condition test on the OSCCTL signal having the value high. The Inverter component is

removed by giving the nodes it is associated with a deleted traceability status. The two nodes of the `Inverting_Buffer#3` can also be given a deleted traceability status because they result in no behavior being exhibited by the subsystem.

The third task is to check component initialisation. The only initialisation required is to set the `INTCLK` attribute to low. This ensures that if the `Generate_Internal_Clock` branch is taken, the `INTCLK` has a predefined value to invert.

The final task of specification is to add reversion nodes. The two branches involving the internal clock both revert back to the `Internal_Clock_Ready` state of the `Clock_Selector`. The two branches involving selecting between the internal and external clock revert back to the `Select_Clock` state of the `Clock_Selector`. Each of the environmental threads reverts back to ensure the signal can be sent or received again in the future. The final Model Behavior Tree of the Clock Selector is shown in Figure 7.12<sup>7</sup>.

## 7.2.4 Design

The Design Behavior Tree is the artifact used to generate VHDL source code suitable to simulate and synthesise into an FPGA. The design stage when applied to digital circuits proceeds in a similar fashion to the standard BE process. Because the DBT is made from verified and validated hardware requirements, however, few design decisions need to be made. Figure 7.13 shows the resulting DBT.

The following design decisions are made to construct the DBT. The system-environment boundary is clarified by removing the environmental components from the BT. The messages sent and received by the environmental components are replaced by external input and output messages from the system component that are relayed to the internal components that form the system. The system-component boundary is clarified by combining the `Inverting_Buffer#1` and `Inverting_Buffer#2` components into a single `Inverting_Buffer` component. A `DpyCT` is not created, as it is not required for deploying the BT in VHDL.

---

<sup>7</sup>To view this figure in more detail, please refer to the electronic version included as part of the supplementary material.

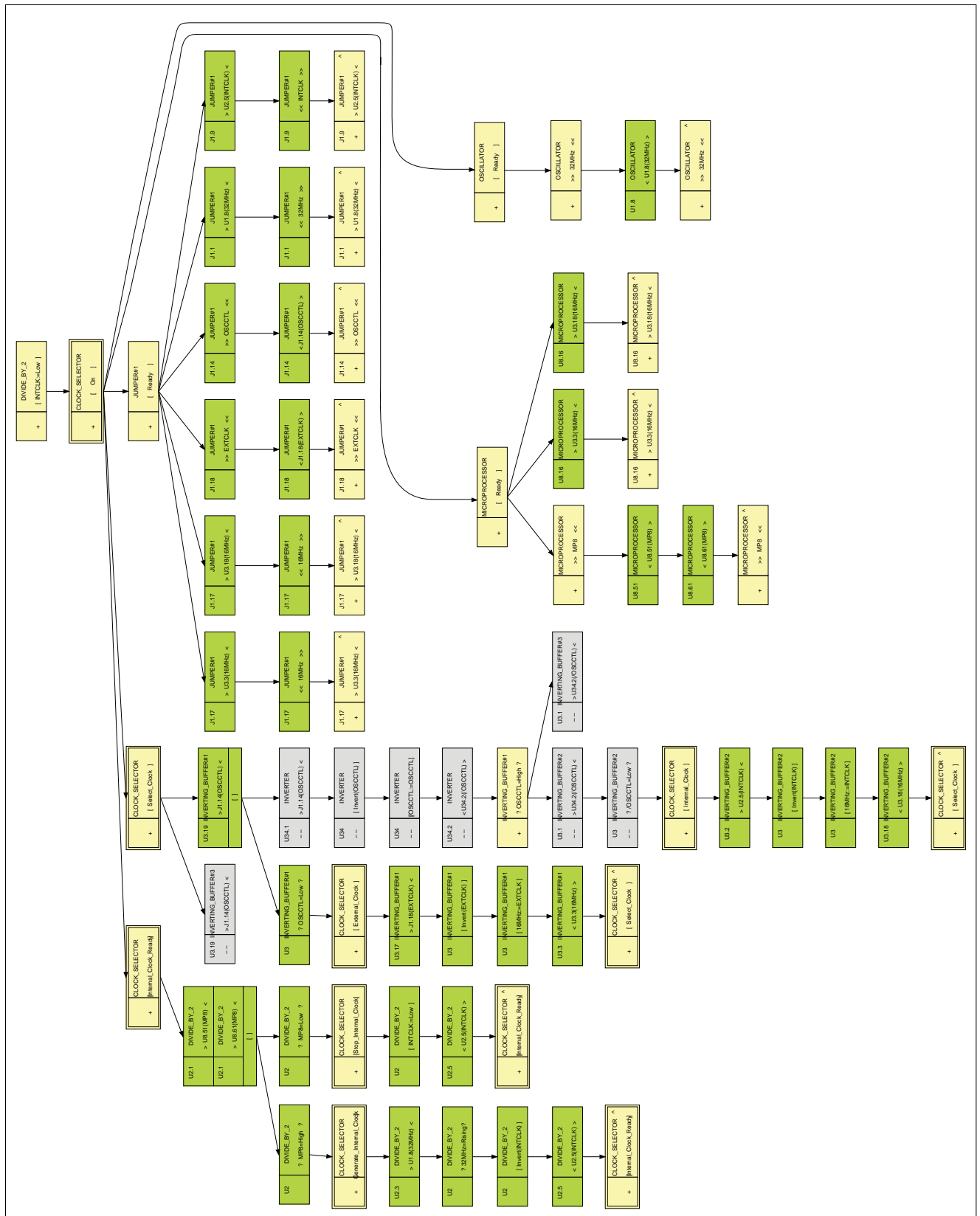


FIGURE 7.12: The Model Behavior Tree of the Sub-System

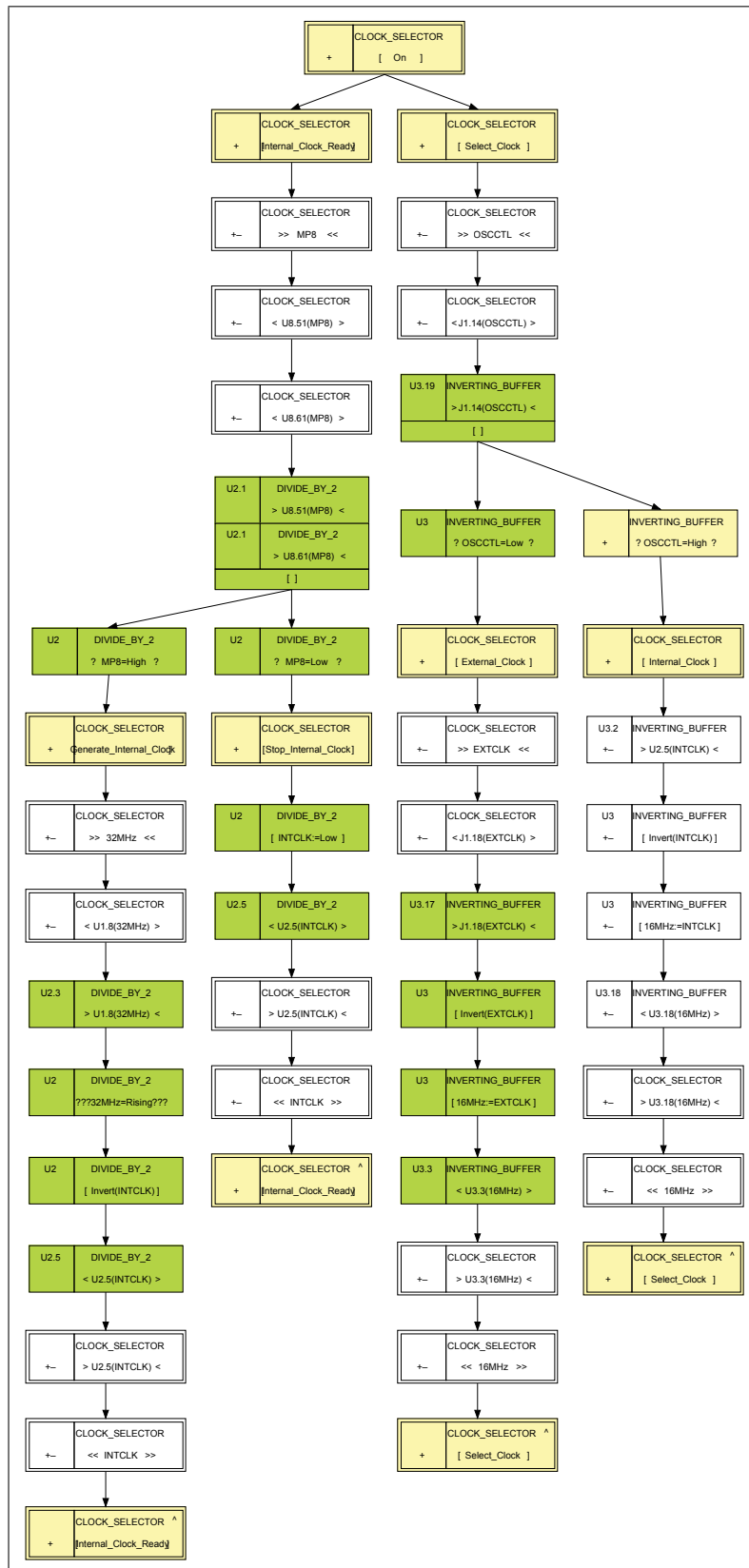


FIGURE 7.13: The Design Behavior Tree of the Sub-System (with deleted nodes hidden)

## 7.3 Performing Failure Modes and Effects Analysis

The MBT created in the specification stage of the BMP cannot be used directly to perform model-checking. This is because the MBT contains domain-specific extensions which cannot be interpreted by the model-checker. In the next section we introduce a series of modifications which allow the domain-specific BT to be used for model-checking. We then discuss the results of performing model-checking and failure modes and effects analysis (FMEA).

### 7.3.1 Modifying the MBT for Performing Model-Checking

Several modifications must be made to the MBT to enable it to be model-checked. Figure 7.14 shows the modified MBT that can be used to perform model-checking. The modifications that are made to the MBT are: (1) removing environmental threads; (2) removing parameterised message passing; (3) transforming domain-specific extensions into existing BT expression syntax; and (4) adding failure views.

The first modification is to remove the environmental threads of behavior. This is necessary because the parallel branching causes a state-space explosion for the model-checker. This modification is not ideal, because when the environmental threads are not present, the messages sent from environmental components cannot be constrained. For example, it would be possible for a high and low value of the 32MHz signal to be sent at the same time. The modified MBT avoids the issues of unconstrained messaging by only receiving messages that are appropriate to the current context.

The second modification is to remove parameterised message passing because it is not supported by the model-checker. Parameterised message parsing is removed by converting the signal attributes to their range of possible values i.e. High, Low, Rising, and Falling. Manual pruning is also performed to remove any unnecessary signal values. For example, the Divide\_By\_2 component only has to receive the Rising value of the 32MHz signal because this is the only value that results in the continuation of behavior. The condition test can also be removed because it is already known that a rising value has been received.

The third modification is to transform any domain-specific extensions into the existing BT expression syntax. In section 7.1, we introduced three BT M2M transformations to

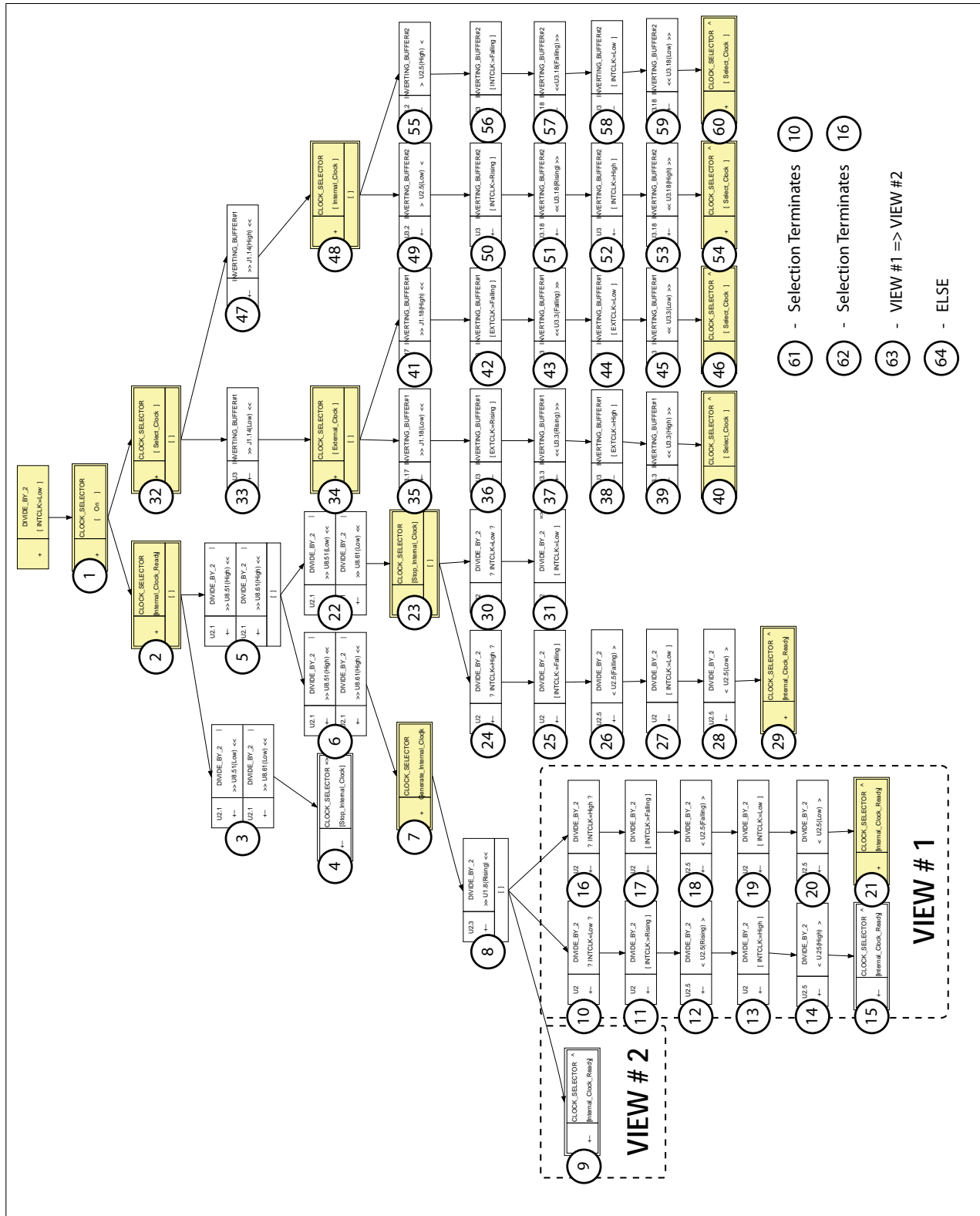


FIGURE 7.14: The Modified MBT used for Model-Checking



describe the semantics for receiving signals from multiple sources, inverting a signal, and generating a rising and falling signal value. Each of these transformations is applied to modify the MBT so it is suitable for model-checking.

Reception of a signal from multiple sources only occurs once in the MBT, when the MP8 signal is sent to U2.1 from both U8.51 and U8.61. The modified MBT shows the transformation of the domain-specific extension combined with the removal of parameterised message passing and manual pruning to remove the condition test that evaluates the value of MP8. The resulting behavior reflects the semantics of digital circuits for receiving a signal from multiple sources. If either of the signals is low, then the MP8 signal is low and the Clock Selector stops the internal clock. This occurs if a low value is received first from U8.51 or U8.61 using a reference. It also occurs if the first value received is high, and the second value received is low. The clock selector only generates the internal clock if a high value is received from both the U8.51 and U8.61 pins.

The invert behavior occurs three times in the MBT, namely when: (1) inverting the INTCLK signal for generating the internal clock; (2) inverting the EXTCLK signal when the external clock is selected to output the 16MHz signal; and (3) inverting the INTCLK signal when the internal clock is selected to output the 16MHz signal. Manual pruning is performed in the second and third instances to remove the condition test which is unnecessary due to the removal of parameterised message parsing.

Transformation of the invert behavior also necessitates the generation of rising and falling signal values when the signal values are set high or low. The invert behavior means that the signal is always changing from low to high or high to low, so a rising or falling signal is always generated. To reflect this, the branch with a reference child node is removed using manual pruning because it cannot occur. The result generates either a rising or falling signal value and outputs the value on the same pin as is used to output the high or low value.

The fourth and final modification to the MBT is to define a failure view. The addition of a failure view to the MBT allows the behavior of the system to be described when a particular component fails. The failure view can then be used together with the model checker to perform failure modes and effects analysis. In Figure 7.14, a portion of the modified MBT is segmented into two views. The first view defines the behavior of the clock selector in normal

operation. The second view defines a potential failure mode of the clock selector, where the oscillator fails so the 32MHz signal is not received. While the 32MHz signal is received in the BT, the failure of the oscillator is modeled by the resulting behavior not occurring.

### 7.3.2 Results of Performing Failure Modes and Effects Analysis

The modified MBT can now be used to perform FMEA using a translation into SAL [dMOR<sup>+</sup>04]. The property of the system we are interested in ensuring is that the internal clock continuously oscillates, that is, it alternately generates a high and low signal value. This can be represented in linear temporal logic (LTL) as:

$$\boxed{\begin{array}{c} G \left( \frac{DIVIDE\_BY\_2}{[INTCLK:=Low]} \Rightarrow F \left( \frac{DIVIDE\_BY\_2}{[INTCLK:=High]} \right) \right) \\ \text{AND} \quad . \quad . \quad . \quad th1 \\ G \left( \frac{DIVIDE\_BY\_2}{[INTCLK:=High]} \Rightarrow F \left( \frac{DIVIDE\_BY\_2}{[INTCLK:=Low]} \right) \right) \end{array}}$$

In LTL, G means always and F means eventually. Therefore, this theorem states that it is always the case that INTCLK will eventually go from low to high and also will always eventually go from high to low.

When a theorem is model-checked, it is either proved or a counter-example (cex) is provided which violates the theorem. Model-checking *th1* results in the cex shown in Table 7.3. The cex uses the numbers annotated on Figure 7.14 to describe the path taken through the BT that violates the theorem. By following the path of the cex through the BT, it is clear that the internal clock does not oscillate if the MP8 signal is set to always stop the internal clock.

This result is only possible if MP8 does not ever have a high value. We are also interested as to whether the internal clock continuously oscillates if all the possible signal values are eventually generated. This requires the addition of fairness to create *th2*:

$$\boxed{\begin{array}{c} G(F(3)) \text{ AND } G(F(5)) \text{ AND } G(F(6)) \text{ AND } G(F(8)) \\ \text{AND } G(F(22)) \text{ AND } G(F(33)) \text{ AND } G(F(47)) \Rightarrow th1 \end{array} \quad . \quad . \quad . \quad th2}$$

This theorem ensures that the nodes 3, 5, 6, 8, 22, 33, and 47 are always reached at some point while proving *th1*. Model-checking this theorem results in it being proved. We can

View #	Theorem	Proved	Runtime (seconds)	Counter-Example (cex)
1	th1	<b>X</b>	6.46	1 32 2 LOOP( 3 4 23 30 31 28 29 )
1	th2	<b>✓</b>	2355.19	
1 and 2	th2	<b>X</b>	2797.1	1 32 2 47 48 3 4 30 31 28 49 50 51 52 53 54 29 33 34 41 42 43 44 45 46 5 33 34 35 36 37 38 39 40 33 34 22 23 30 31 28 29 5 6 7 63 8 9 5 6 7 8 9 3 4 LOOP( 30 31 28 29 41 42 43 44 45 46 47 48 3 4 30 31 28 49 50 51 52 53 54 29 33 34 41 42 43 44 45 46 33 34 35 36 37 38 39 40 5 33 34 22 23 30 31 28 29 5 67 8 9 5 6 7 8 9 3 4 )

TABLE 7.3: Results of Performing Failure Modes and Effects Analysis

now enable the failure view to occur and model-check the theorem again. The resulting cex indicates that a failure of the oscillator causes the INTCLK signal to stop oscillating as is expected.

The modification of *th1* to ensure fairness significantly increases the time taken for the model-checker to provide a result. It is therefore unlikely that this approach to FMEA would scale to larger hardware subsystems or systems. The approach does, however, demonstrate that a domain-specific BT can be transformed and used for applications intended for BTs defined in the existing BT expression syntax.

## 7.4 Simulation and Synthesis

This section describes a second use of the domain-specific BTs to generate VHDL code. This allows the domain-specific BT to be used as an intermediary to migrate outdated digital circuitry onto an FPGA. The VHDL code is generated from the DBT resulting from the BE design stage. The simulation of the generated VHDL code is validated by a VHDL simulation that uses TTL ICs. The VHDL code is also used to synthesise a gate-level design that is suitable to be deployed on an FPGA.

### 7.4.1 Generating VHDL Code

VHDL code is generated from the DBT of a hardware subsystem using a combination of model-to-model (M2M) and model-to-text (M2T) transformations. The M2M transformations create a mapping from the BE metamodel to a VHDL metamodel. The M2M transformations create one VHDL model that captures the integration of components in the subsystem primarily from the message passing involving the system component. It also creates a VHDL model for each component of the system (excluding the system component) that captures the behavior that is described in the DBT. The M2T transformations generate VHDL code from each of the derived VHDL models. This ensures there is one VHDL file for the system component, and one VHDL file for every other component in the DBT.

During mapping from the BE metamodel to the VHDL metamodel, signal names are modified to ensure they adhere to the IEEE 1076.6 standard for VHDL Register Transfer Level (RTL) synthesis [IEE04]. The standard uses a naming convention for signals that consists of a leading letter, followed by one or more letters, numbers or underscores (`[a-z,A-z][a-z,A-Z,0-9,-]*`). This naming convention is violated in the DBT with signals starting with a numeral (e.g. 32MHz) and signals that contain a backslash (/). Signals that start with a numeral in the DBT are prefixed with an 's' (an abbreviation of signal) during mapping to the VHDL metamodel. Any backslashes (/) are removed from signals, and the signal name is appended with 'Neg' (negated) to denote that the signal has been inverted. Signals which are derived from attributes that have the values they store modified (as opposed to just used for parameterised message passing), are prefixed with an 'a' (an abbreviation of attribute).

Figure 7.15(a) shows the generated VHDL code that describes the component integration in the Clock Select subsystem. The VHDL code consists of two sections: the entity section which defines the interface, and the architecture section which defines the behavior of the digital circuit. These two sections are composed of the ports and the port mappings that are derived from the inputs and outputs in the DBT. There are three cases for mapping the inputs and outputs in the DBT to define the ports in VHDL code: (1) an external input and an external output belonging to the system component; (2) an internal input and an internal output sent from a component to the system component; and (3) an internal input

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity CLOCK_SELECTOR is
    Port ( s16MHz : out STD_LOGIC;
          INTCLK : out STD_LOGIC;
          EXTCLK : in  STD_LOGIC;
          OSCCTL : in  STD_LOGIC;
          MP8    : in  STD_LOGIC;
          s32MHz : in  STD_LOGIC);
end CLOCK_SELECTOR;

architecture structural of CLOCK_SELECTOR is
    component DIVIDE_BY_2 is
        Port ( MP8 : in  STD_LOGIC;
              INTCLK : out STD_LOGIC;
              s32MHz : in  STD_LOGIC);
    end component;

    component INVERTING_BUFFER is
        Port ( OSCCTL : in  STD_LOGIC;
              EXTCLK : in  STD_LOGIC;
              INTCLK : in  STD_LOGIC;
              s16MHz : out  STD_LOGIC);
    end component;

    signal aINTCLK : STD_LOGIC;
begin
    cINVERTING_BUFFER: INVERTING_BUFFER port map (OSCCTL, EXTCLK, aINTCLK, s16MHz);
    cDIVIDE_BY_2 : DIVIDE_BY_2 port map (MP8, aINTCLK, s32MHz);
    INTCLK <= aINTCLK;
end structural;

```

(a) Clock Select Subsystem

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity DIVIDE_BY_2 is
    Port ( MP8 : in  STD_LOGIC;
          INTCLK : out STD_LOGIC;
          s32MHz : in  STD_LOGIC);
end DIVIDE_BY_2;

architecture Behavioral of DIVIDE_BY_2 is
begin
    process (MP8, s32MHz)
        VARIABLE aINTCLK: STD_LOGIC;
    begin
        if (MP8='0') then
            aINTCLK := '0';
            INTCLK <= aINTCLK;
        elsif (MP8='1') then
            if (RISING_EDGE(s32MHz)) then
                aINTCLK := not (aINTCLK);
                INTCLK <= aINTCLK;
            end if;
        end if;
    end process;
end Behavioral;

```

(b) Divide By 2 Component

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity INVERTING_BUFFER is
    Port ( OSCCTL : in  STD_LOGIC;
          EXTCLK : in  STD_LOGIC;
          INTCLK : in  STD_LOGIC;
          s16MHz : out  STD_LOGIC);
end INVERTING_BUFFER;

architecture Behavioral of INVERTING_BUFFER is
begin
    process (OSCCTL, INTCLK, EXTCLK)
    begin
        if (OSCCTL='0') then
            s16MHz <= not (EXTCLK);
        elsif (OSCCTL='1') then
            s16MHz <= not (INTCLK);
        end if;
    end process;
end Behavioral;

```

(c) Inverting Buffer Component

FIGURE 7.15: VHDL Code generated from the DBT

and an internal output sent and received between two components not involving the system component.

In the first case, external input and external output belonging to the system component is used to generate the entity section. The entity is given the name of the system component and the external inputs and external outputs define the ports of the entity. External input is defined as input ports (in) of type STD\_LOGIC and external output is defined as output ports (out) of type STD\_LOGIC.

In the second case, internal input and internal output sent from a component to the system component is used to generate component definitions at the start of the architecture section. Each component is given the name of the component in the DBT. Internal input is defined as input ports of type STD\_LOGIC and internal output is defined as output ports of type STD\_LOGIC. These component definitions are used later in the architecture section for port mapping

In the third case, internal input and internal output is sent and received between two components, neither of which is the system component. The ports of these two components cannot be directly connected using port mapping. The signal that is sent using the internal input and internal output is instead defined as a signal in the architecture section and prefixed with an 'a' to indicate it is derived from an attribute.

The generated ports and signals are now connected in the architecture section by mapping the ports of the entity and shared signals to the ports of the components. Each component is initialised using the name of the component prefixed with a 'c'. The ports of the entity and the shared signals are mapped based on the order they are defined in the port mapping. For example, the OSCCTL input of the CLOCK\_SELECTOR entity is mapped to the first port of the INVERTING\_BUFFER component, which is also the OSCCTL input. When a signal is used to map a port between two components, and the same signal needs to be mapped to the port of the entity, then this signal indirectly connects the port of the entity to the port of the component. This occurs in Figure 7.15(a), where the INTCLK port of CLOCK\_SELECTOR is connected to the DIVIDE\_BY\_2 and INVERTING\_BUFFER components using the aINTCLK signal.

The behavior of the components defined in the architecture section of the subsystem's

VHDL code is defined in separate VHDL code generated from each component's CBT. Figure 7.15(b) shows the VHDL code generated from the CBT of the Divide\_by\_2 component and Figure 7.15(c) shows the VHDL code generated from the CBT of the Inverting Buffer component. The name of the entity and the port definition is the same as described for the VHDL code of the subsystem. Each signal received on an input port is also added to the sensitivity list of the process statement. This ensures that the behavior of the process statement is re-evaluated whenever an input changes. If the value of a signal needs to be stored then a variable is defined which is given the name of the signal prefixed with an 'a' to indicate it is derived from an attribute. The body of the process is generated by mapping the CBT to VHDL code using the BT-VHDL mapping defined in Section 7.1.

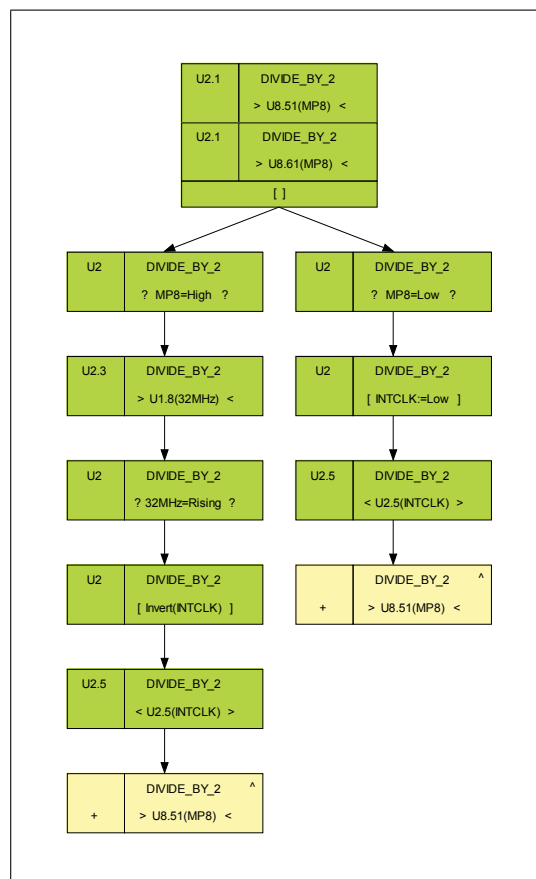
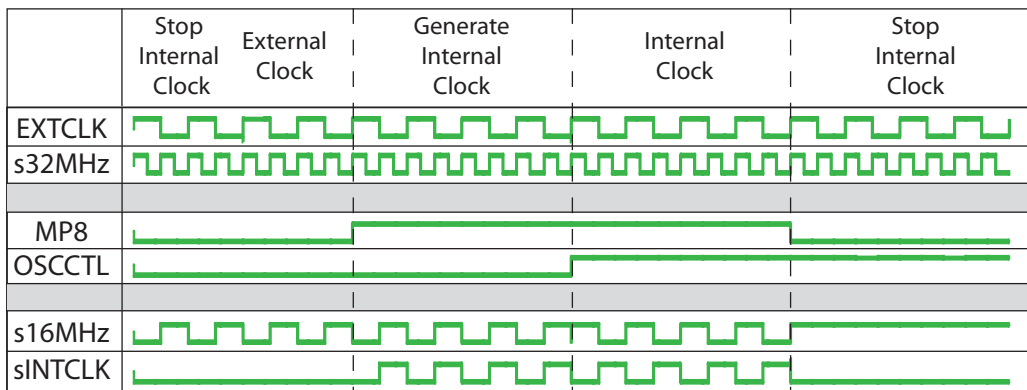


FIGURE 7.16: Component Behavior Tree of Divide By 2 Component

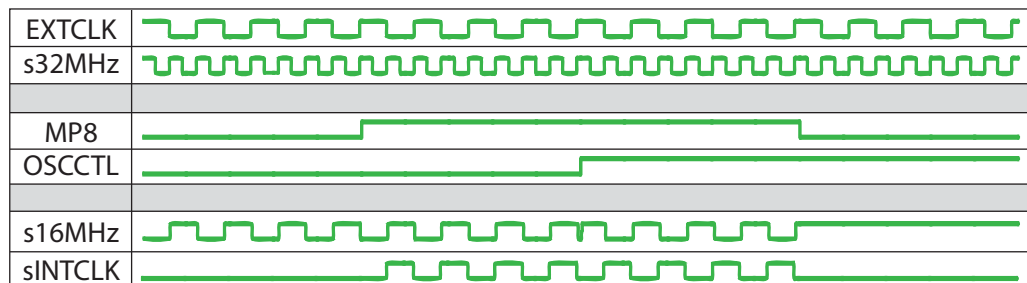
Figure 7.16 shows the CBT of the Divide by 2 component of the Clock Select subsystem. The mapping of the CBT to the body of the process in VHDL is as follows. The first

branch starts with a condition test on the MP8 signal having a low value which is mapped to *if (MP8='0') then*. The attribute assignment of the MP8 signal to low is mapped to *aINTCLK:='0'*. The aINTCLK variable is also mapped to the INTCLK output port. The second branch starts with a condition test on the MP8 signal having a high value. Because the branches are connected by alternative branching, the mapping is *elsif (MP8='1')* *then*. The condition check on the 32MHz signal having a rising value is mapped to *if (RISING\_EDGE(s32MHz)) then*. The 32MHz signal is prefixed with 's' to meet naming conventions. Applying the invert behavior to INTCLK is mapped to *aINTCLK <= not(aINTCLK)*. The aINTCLK variable is also mapped to the INTCLK output port. Finally, two *endif;* statements are added to finish the process body.

### 7.4.2 Simulation



(a) Behavior Engineering version



(b) TTL version

FIGURE 7.17: Simulation of the Clock Select System

The generated VHDL can be used to simulate the clock selector subsystem. This requires



a testbench that provides: a 16MHz signal to EXTCLK; a 32MHz signal to s32MHz; and toggles MP8 and OSCCTL. Simulating the clock selector subsystem using the testbench results in the output values of the s16MHz and the INTCLK signals. Figure 7.17(a) shows the result of simulating the clock select subsystem derived from the BE model.

The VHDL generated from the BE model is validated by VHDL code based on the original TTL ICs in the circuit schematic. The STND library [Fre09] provides simulatable versions of the 74xx/54xx series of ICs. The library makes use of the VITAL\_Timing library to decrease the simulation time, however, this also means the VHDL code cannot be used for synthesis. Figure 7.18 shows the VHDL code made using the STND library. The components from the STND library are used that match the type of TTL ICs used in the subsystem and are integrated together according to the wiring in the circuit diagram.

Figure 7.17(b) shows the result of simulating the clock select subsystem using the VHDL code made from the STND library. It is clear that the simulation of the VHDL generated by the DBT is equivalent to the simulation of the TTL ICs using the STND library.

### 7.4.3 Synthesis

The second use of the generated VHDL code is to synthesise a gate-level design that can be deployed on an FPGA. Figure 7.19 shows the gate-level design of the clock select system that is synthesised from the generated VHDL code. The figure is separated into a portion derived from the INVERTING\_BUFFER component and a portion derived from the DIVIDE\_BY\_2 component to simplify its comprehension.

The gate-level design is connected to input and output signals through input and output buffers. These buffers temporarily store data to accommodate different rates of data flow. The buffered s32MHz and MP8 signals are sent to the DIVIDE\_BY\_2 portion of the gate-level design, which outputs the INTCLK signal. The D flip-flop with clear (FDC) is used to store the previous value of the aINTCLK variable. If the MP8 signal has a high value, then the value of the aINTCLK is inverted on each rising edge of the s32MHz signal by using the inverted output of Q as an input for D. If the MP8 signal has a low value, when it is inverted to high it causes the FDC to be cleared and a low value to be output for the INTCLK signal.

```

library TTL;
library IEEE;

use TTL.std74;
use TTL.std14;
use TTL.std240;
use IEEE.std_logic_1164.all;

entity ClockSelectTTL is
  PORT (
    INTCLK : OUT    std_logic;
    s32MHz : IN     std_logic;
    MP8    : IN     std_logic;
    OSCCTL : IN     std_logic;
    s16MHz : OUT    std_logic;
    EXTCLK : IN     std_logic);
end ClockSelectTTL;

architecture structural of ClockSelectTTL is
  component std74 is
    PORT (Q, QNeg : OUT std_logic; D, CLK, PRENeg, CLRNeg : IN std_logic);
  end component;
  signal QNeg : std_logic;
  signal D : std_logic;

  component std14 is
    PORT (A : IN std_logic; YNeg : OUT std_logic);
  end component;
  signal OSCCTLNeg : std_logic;

  component std240 is
    PORT (YNeg : OUT std_logic; A, OENeg : IN std_logic);
  end component;
  signal INTCLK2 : std_logic;

begin
  U2: std74 PORT MAP (INTCLK2, QNeg, D, s32MHz, '1', MP8);
  INTCLK <= INTCLK2;
  D <= QNeg;
  U34: std14 PORT MAP (OSCCTL, OSCCTLNeg);
  U3 : std240 PORT MAP (s16MHz, EXTCLK, OSCCTL);
  U3_2 : std240 PORT MAP (s16MHz, INTCLK2, OSCCTLNeg);
end structural;

```

FIGURE 7.18: VHDL of TTL Version of the Clock Select System

The buffered EXTCLK and OSCCTL signals are sent to the INVERTING\_BUFFER portion of the gate-level design together with the INTCLK signal from the DIVIDE\_BY\_2 portion. The signals are used in the INVERTING\_BUFFER portion to select the inverted value of either EXTCLK or INTCLK to be output as s16MHz. The top AND gate combines the inverted EXTCLK signal with an inverted OSCCTL signal. The bottom AND gate combines the inverted INTCLK signal with the OSCCTL signal. This ensures that only one gate can output a high signal at any time, because an inverted OSCCTL signal is used in

one AND gate and the original OSCCTL signal in the other AND gate. The result of the two AND gates is combined with an OR gate that outputs the s16MHz signal.

## 7.5 Discussion

The construction of domain-specific BTs, as described in this chapter, requires detailed knowledge of both BTs and the domain being captured. Therefore, although the resulting domain-specific BTs may be understood by a wider audience, there is also an increase in the expertise required for their construction.

The chapter also only provides one case study of domain-specialisation using BE. Future

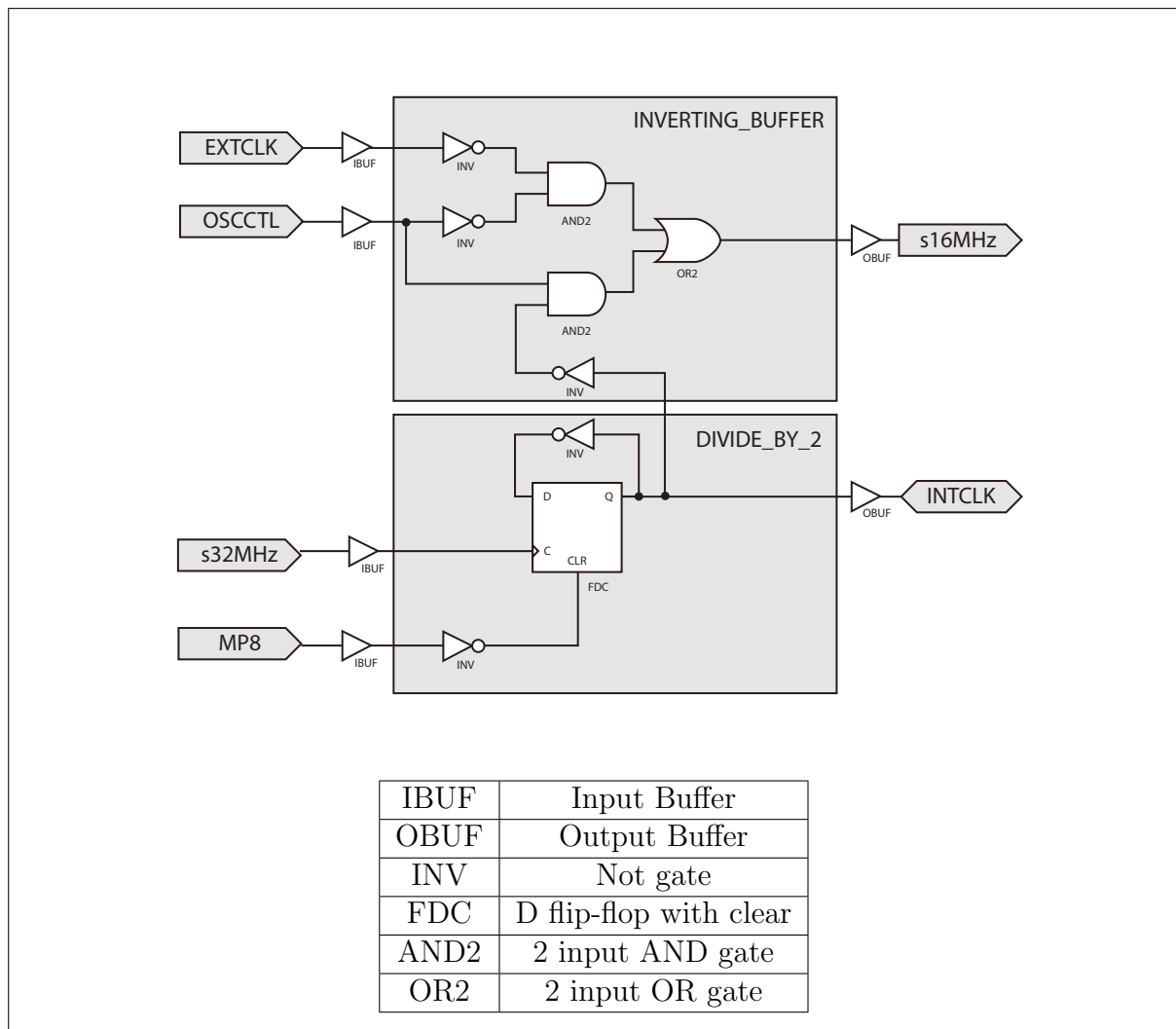


FIGURE 7.19: Synthesised Gate-Level design of Clock Select System

work could focus on providing several more case studies to show the adaptability of the approach. Some case studies should be selected to demonstrate the extension of the BT expression syntax using domain-specific components.

In the case study itself, there is a gap between the domain-specific representation of the circuit schematic and the domain-specific BTs. This is handled in the case study using a customised traceability link and parameterised message passing. In future work, this case study could be revisited and improved by first using structure trees to capture the structure of the circuit schematic. The resulting structure tree could then be linked to the domain-specific BTs which describe the behavior of the circuit.

Finally, the VHDL code generated from the BT is probably not the most efficient solution and relies on synthesis to improve the efficiency at the gate-level design. Because the TTL library that was used for comparison can be used for synthesis it is assumed that the VHDL generated from the BT is more efficient than a TTL solution would be. If this is not the case, then the advantage of this approach is to document the system in an easy to comprehend representation that provides access to an existing toolset used for general-purpose BTs.

## 7.6 Conclusion

A migratory approach that scaleably replicates outdated TTL ICs on an FPGA minimises the risk of upgrading this type of legacy system. The need for such an approach is justified by the real-world case study presented in this chapter. The migratory approach presented in this chapter can also be applied to other industries where legacy systems that use TTL ICs can still be found, such as in medical equipment, aviation, space exploration, and defence. The approach uses BTs as an intermediate representation by creating a domain-specific extension for capturing digital circuits. This approach allows concepts of digital circuits that are not directly mappable to the existing BT expression syntax to be captured. By making a mapping from this domain-specific extension to the existing BT expression syntax, the BT intermediary can be applied to other applications of BE. This is demonstrated by using the BT intermediary to perform model-checking and FMEA.

The contributions of this chapter are summarised below:

1. A migratory approach is described that replicates legacy systems composed of TTL ICs on an FPGA
  - (a) The approach uses BTs as an intermediate representation to capture digital circuits using a domain-specific extension. This approach has two benefits:
    - i. BTs provide documentation of the functionality of the system in a format that is accessible to a wider audience.
    - ii. By mapping the domain-specific extension to the existing BT expression syntax, the intermediary can be used for other applications of BE.
2. The use of a domain-specific extension to BTs was demonstrated.
  - (a) The domain-specific extension models concepts of digital circuits that are not directly mappable into the existing BT syntax.
  - (b) Transformations map the domain-specific extensions to the existing BT syntax to perform model-checking and FMEA.



# Part V

## Discussion





# 8

## Discussion

Mainstream software and systems engineering (Sw&SE) approaches are struggling to build the next generation of large-scale software-intensive systems, resulting in the widespread failure of large-scale projects. Building large-scale systems requires the issue of scalability to be properly addressed by managing the complexity inherent in the task. Mainstream Sw&SE approaches utilise two techniques to manage scalability-induced complexity: the usage of abstraction to hide complexity; and iteration and intuitive miraculous leaps to transition between gaps in a process.

Abstraction is used to hide complexity which simplifies the tasks of comprehending and satisfying the system requirements. The usage of abstraction, however, merely delays the re-emergence of scalability-induced complexity to the point at which the approach is applied to systems of a larger scale. Mainstream approaches also rely on one or more intuitive

miraculous leaps when transitioning from the initial requirements to a specification, a design or a deployed system. These miraculous leaps require the intuition of the developer to build the next stage of the system based upon their current comprehension of what is required. Deficiencies in both the developers current comprehension of the system and their intuition of what is required are accommodated by performing several iterations of the miraculous leap. Each iteration gradually approaches satisfaction of the requirements, assisted by an improved understanding of the system derived from the previous iteration.

This dissertation advocates a different approach which uses a scaleable methodology to build large-scale software-intensive systems. A scaleable methodology can be applied to systems of ever-increasing scale by ensuring that the local problem space maintains a minimal and constant size regardless of the size of the global problem space. The need for intuitive miraculous leaps and excessive iteration is removed by combining a scaleable methodology with an end-to-end approach. This combination ensures that a modeler is only ever required to deal with a minimally sized local problem space, all the way from the initial requirements through to a design and a deployed system.

A Sw&SE approach that uses a scaleable methodology can dramatically improve our ability to manage the complexity of large-scale systems. By managing this complexity, a scaleable methodology reduces the risk of project failure and enables systems to be built in shorter timeframes, for less cost, and to be made at a higher quality.

## 8.1 A Scaleable Methodology for System Design

This dissertation presents a scaleable methodology for systems design by extending the Behavior Engineering (BE) approach to support design in a Model Driven Engineering (MDE) framework. BE is a good candidate for this objective because it already uses a scaleable methodology to perform requirements analysis of large-scale systems. The scaleability of BE for requirements analysis has been proven in industry, where it has had significant success when applied to large specifications of over 1000 requirements [Pow07, Bos08].

Current research involving BE primarily focuses on using BE models as a formal

specification [WD04, WD06, GLYW05, GWY08, GCW07, ZD05a, ZD05b, ZWCD06], which can then be further analysed using techniques such as model-checking. This dissertation adds to this research by extending the BE approach to support design and deployment. The new design stage for BE is developed within an MDE framework by considering BE's graphical representation as a Domain Specific Language (DSL) for capturing the modeling dimension of system behavior. By extending BE to support design and deployment in an MDE framework, the extended BE approach benefits from the productivity improvements associated with MDE.

The resulting combination of a scaleable methodology and an end-to-end approach is demonstrated using three case studies intended to show a wide cross section of applications. The case studies capture each of Bezivin's three types of MDE applications [BBJ07]: forward modeling, interactive modeling, and reverse modeling.

## 8.2 Comparison with Related Work

In Chapter 2, we reviewed the state of the art in MDE. The work presented in this dissertation contrasts with related work in three key areas: (1) MDE approaches ; (2) visual languages used for design of system behavior; and (3) underlying representations.

The MDE approaches we reviewed are three different realisations of MDA: Meta-modeling, the UML Platform-Independent Model (UML PIM), and Executable and Translatable UML (xtUML). All of these approaches focus on using abstraction to hide complexity rather than focusing on using a scaleable methodology for building models. Meta-modeling approaches focus on capturing abstract syntax which is suitable for static languages which focus on syntax but does not suit the dynamic, semantically-rich languages that capture system behavior.

Using the UML as a PIM has numerous issues that need to be overcome, including imprecise and incomplete semantics, a lack of formal use, an overly complex and large language, and an easily misused and underspecified extension mechanism. Another complication results from a central aim of UML which is to allow UML to be used in combination with any methodology. This aim makes combining UML with a scaleable

methodology problematic, because a scaleable methodology requires close links to a representation.

xtUML addresses some of the issues associated with UML by making a subset of UML with executable semantics and an action language. The resulting language is oriented towards design and deployment, however, making it difficult to capture system information at earlier stages. This is overcome in xtUML by using numerous cycles of iteration and verification to cross the informal/formal barrier.

Our approach is closest to a combination of UML PIM and xtUML. The extended BE approach provides an extension mechanism for creating a family of languages similar to UML profiles. The extended BE approach also supports executable models with a well defined formal semantics which is similar to xtUML. The key difference that distinguishes our approach is that it provides a scaleable methodology for building models with an end-to-end approach that is applicable from the initial requirements to a design and then a deployed system.

The second area in which our approach can be compared with current practices involves other visual languages used for the design of system behavior. We elaborate on Sowa's [Sow00] comparison of flowcharts, finite state machines, and Petri nets to discuss the key advantages of BTs. Sowa showed Petri nets can be considered as a combination of flowcharts and finite state machines. BTs are similar to Petri nets with three key distinctions. Firstly, BTs remove the cycles present in a PN by using a reversion operator that forms an implicit link with an earlier part of the tree. This gives BTs a tree-like structure which avoids the potential for obfuscation of information present in a network structure. Secondly, BTs modularise behavior differently to a PN. Whereas a PN groups similar behavior into modules, a BT individually tags each node with an associated component. This has the benefit of allowing the tree-like structure to be maintained which is not possible if similar behaviors are graphically segmented into individual modules. Finally, BTs visually define transitions using a small subset of formalised transition types that are tagged to each node. This negates the need for transitions to be defined in a separate modeling language. It also moves the definition of transitions to nodes, allowing edges to be used solely to define the flow of control.

The final area of comparison with related work is the underlying representation on which both the visual languages and MDE approaches are based. The majority of visual languages and MDE approaches used for capturing system behavior have an underlying object-oriented (OO) representation. Some researchers assert that OO concepts are not suited to MDE [Fra04b]. It is also acknowledged by researchers that objects shift complexity from classifying and designing types of objects to describing the collaboration between objects ([Tri06], p3). This shift of complexity makes OO ill suited for comprehending large-scale systems. BE uses a different underlying representation which we have named behavior-based (BB). BB approaches have a historical root in robotics, but have more recently been applied to artificial intelligence and component-based software engineering. Whereas OO approaches encapsulate interactions inside components together with data and computation, BB approaches separate the task of integration from the definition of computation. This is achieved by first focusing on defining interactions between entities which then naturally decomposes the overall system behavior into several smaller units of self-contained behavior. This in turn has benefits for development, testing, re-use and the maintenance of components. It also enables the scalable methodology that is central to the BE approach.

## 8.3 Contribution

Figure 8.1 provides a summary of the contribution of this dissertation to the field of software engineering. The figure is annotated with both the sections and case studies that are relevant to each contribution. This dissertation provides contributions to both BE and MDE, because it is primarily an extension of BE to support design and deployment in an MDE framework. In the following discussion, each contribution is linked to the relevant section and case study in this dissertation using the format, (section / case study).

The contribution to BE involves the extension of the approach to design and deployment. BE now supports design with the addition of a component model (4.2.2/A,B), an extension mechanism (4.3/C); and a new design stage of the BMP (4.4/A,B,C). Deployment is supported by a toolset consisting of an editor (5.1/A), a virtual machine; and M2M and M2T transformations (5.2,5.3/A,C). The virtual machine consists of two versions: the BRE

(5.4,6.2.1/B); and the eBRE (5.4/A) which is an embedded version of the BRE for small footprint embedded systems.

The contribution to MDE involves positioning BE as a domain-specific language in an MDE framework. The contribution of incorporating the extended BE approach in the MDE framework to create a BE DSL is: (1) it provides an end-to-end scaleable methodology (3,4/A,B,C); (2) it simplifies comparison with other visual DSLs for capturing system behavior (2.3/A,B); and (3) the BE DSL is widely applicable (5,6,7/A,B,C).

The BE DSL provides a scaleable methodology for capturing system behavior that is applicable end-to-end from requirements all the way to a completed design and deployed system. Extending BE to design and deployment in an MDE framework also simplifies comparison of BE with other visual DSLs for capturing system behavior. This allows the benefits of BE to be demonstrated by contrasting the results with similar case studies performed in other visual languages. Finally, to gain the benefits of the scaleable methodology of BE and its underlying representation, the BE DSL has been demonstrated

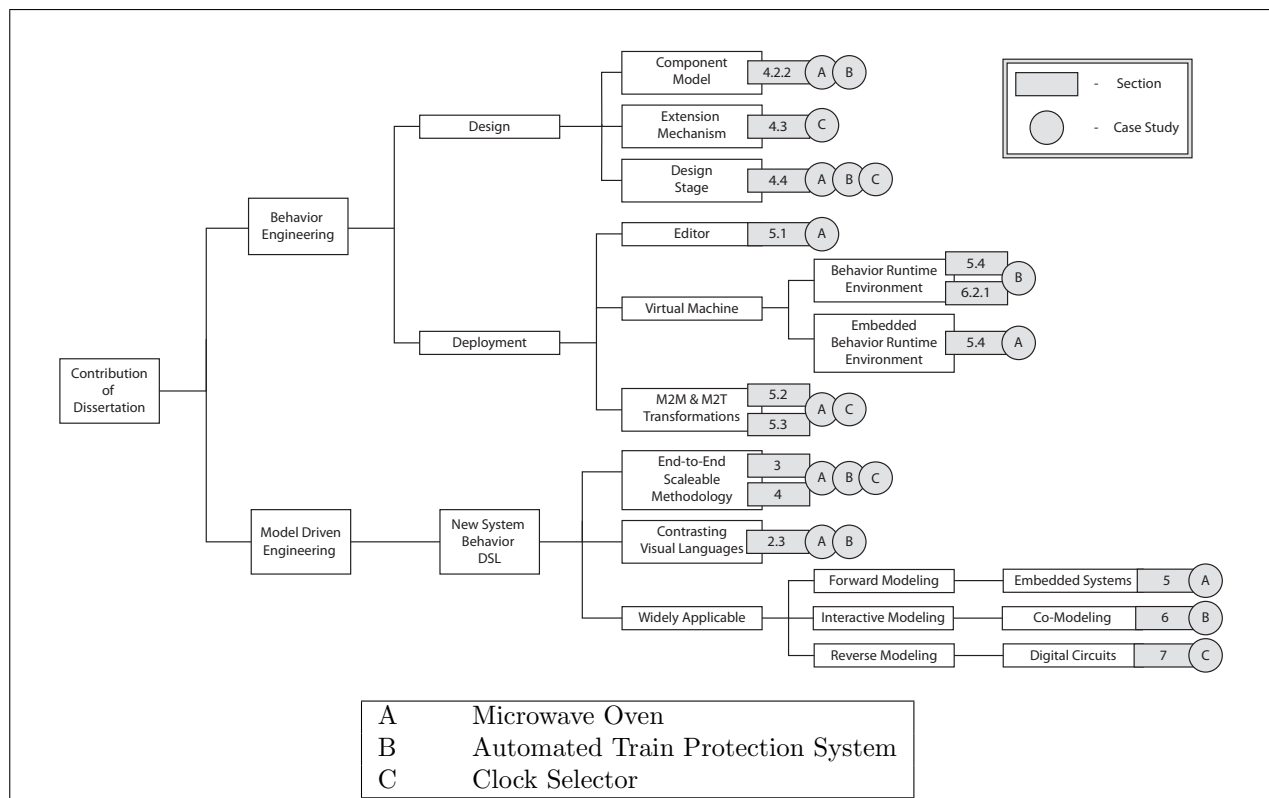


FIGURE 8.1: Summary of Contribution

to be applicable to a wide range of MDE applications. This involved demonstrating Bézivin's [BBJ07] three types of MDE applications consisting of forward modeling, interactive modeling, and reverse modeling. Forward modeling with BE is demonstrated by modeling a case study from the initial requirements all the way to deployment as an embedded system (5/A). The BE approach is also demonstrated by performing interactive modeling using a new approach to integrated software/hardware design called co-modeling (6/B). The BE approach is further demonstrated by performing reverse modeling by using BE as an intermediary representation together with the domain-specific extension mechanism to model legacy digital circuits (7/C).

## 8.4 Future Work

The broad scope of this dissertation allows for several possible avenues for future work. Our suggestions for future work are in three key areas: (1) the design stage of the BMP; (2) tool support for BE design and deployment; and (3) an extension of our co-modeling approach.

### 1. The Design Stage of the Behavior Modeling Process

- (a) **Decompose Large Specifications into system of systems (SoS):** The design stage of the BMP can be improved by incorporating the concept of systems of systems. This would allow large specifications to be decomposed into several smaller integrated designs. One means of achieving this may be to exploit co-modeling to investigate appropriate system partitions and means of communication between the systems. A SoS extension can be made compatible with the existing component model by integrating systems using the system-environment and system-component boundaries.

### (b) Extend the BE Extension Mechanism

- i. **Graphically Defining Domain-Specific Constraints:** The addition of structure trees (STs) to the BML could be used to capture the generic structure of the components that form a system and their allowable set of relationships. A visual DSL for defining instances of the generic system

structure could easily be created by mapping graphics to each component and relation in the structure tree. This visual DSL would allow users to easily define syntactically correct system structures which could be linked to a BT to define the behavior of the components and their relationships. The resulting development framework would be similar to meta-modeling with the exception that domain-specific constraints are defined in a visual language as opposed to being defined in a textual language such as the Object Constraint Language (OCL) which is used together with UML.

- ii. **Fuzzy Control Domain-Specific Extension of Behavior Trees:** The BT expression syntax currently uses a set of discrete states to describe the range of behavior exhibited by a component. Discrete states can be restrictive, particularly when the behavior of a component involves measurements, for example, of distance, temperature, or speed. Fuzzy logic uses linguistic variables which allow qualitative states to be used together with a degree of truth. For example, qualitative states such as cold, moderate, and hot can be used to describe various temperature ranges. This allows a continuous range to be mapped to a discrete set of states. For example, a temperature of 50 °C could be 40% moderate and 60% hot. Fuzzy control [Jan07] links input linguistic variables to output linguistic variables using a set of rules. Fuzzy control has been applied to numerous applications including automating subway systems, traffic light control, washing machines and handwriting recognition. Defining a fuzzy control domain-specific extension for BTs could have two benefits. Firstly, it could simplify the expression of fuzzy control rules. Secondly, it could allow the fuzzy controller and the surrounding software system to be integrated and defined in the same language.

## 2. Tool Support for BE Design and Deployment

### (a) **Extend the Behavior Runtime Environment**

- i. **Multiple Component Instances:** The BRE currently only supports



BTs consisting of a predefined set of components that are known at the start of the execution of the system. BTs also have a means of defining multiple component instances based upon a single component definition. To support multiple component instances, the BRE will need to be extended to dynamically create and destroy component instances.

- ii. **Relational Behavior:** BTs have recently been extended to support an executable version of relational behavior [WCD09]. The BRE can be extended to support executing relational behavior by extending the DynCT and the expression parser.
- iii. **Algorithm Architecture Adequation (AAA):** AAA [GLS99] is a methodology for implementing algorithms onto a hardware platform of networked microcontrollers and specialised hardware. We propose to investigate AAA with the goal of creating a program that analyses BE models to determine the most efficient solution to deploy the BE components on a distributed architecture.

(b) **Improve the BE Eclipse Editor**

- i. **TextBT Plug-in:** In some cases, it may be quicker to define a BT textually rather than graphically. BTs defined using a textual editor could be visualised in real-time using the existing graph layout package, Graphviz. Visualisations made with Graphviz can easily be saved in many formats e.g PDF, PS, PNG, and SVG. The textual format could also provide an interchangeable format which would promote compatibility amongst BT tools.
- ii. **DynBT Plug-in:** The SVG visual output of the textBT plug-in could also be leveraged to visualise the dynamic behavior of a BT. Visualising the flow of control through a BT using a provided trace could be applied to several areas. One area is to provide animations that demonstrate the BT semantics. Another area is to visualise counter-examples resulting from model-checking.
- iii. **BE component integration environment (BECIE):** The concept of BECIE was originally envisaged by Larry Wen to investigate systems of

systems using BE. We propose to extend this work to create a framework for students and academics to investigate the design and deployment of BE models. This will involve BECIE visualising BTs that are executed by the BRE. BECIE will also manage a component repository that allows users to supply BE components to be executed by the BRE.

### 3. Extension of Our Co-modeling Approach

- (a) **Behavior Tree Modelica Library:** This work involves creating a BT library for Modelica similar to the StateGraph library [OÅD05]. This would be beneficial for co-modeling as BTs can then be executed natively in Modelica, rather than in C++ by using external C functions. It would also have the benefit of opening design and deployment of BE models to a wide Modelica userbase.
- (b) **Three Tier Co-modeling Approach:** The current co-modeling approach is hampered by the need to continuously revise the Modelica model each time the co-model is changed or a new co-modeling scenario is investigated. To mitigate this issue, we propose augmenting the current co-modeling approach by integrating the testing language, Testing and Test Control Notation version 3 (TTCN-3) [WDT<sup>+</sup>05]. The resulting three-tier co-modeling approach would utilise TTCN-3 to define the parameters of different co-models and co-modeling scenarios. TTCN-3 would then perform the required simulations using the Modelica and BE models and collate the results.

## Part VI

## Appendices





## Contrasting BTs with other visual languages

In this appendix, we illustrate the differences between Behavior Trees and other visual languages for capturing system behavior using three small case studies. The three case studies are used to contrast Behavior Trees with the visual languages of State Machines, StateCharts, and Exogenous Connectors. In each case study, the language being contrasted with BTs is discussed based on the three key issues presented in Chapter 2:

- **Transitions:** Behavior Trees use a small subset of formalised transitions which are defined graphically in the same diagram. When contrasting the languages with Behavior Trees, we are interested in differences in how transitions are defined. We explore whether the language uses formalised transitions and if it does whether the formal transitions are defined visually in the same diagram.
- **Segmentation:** Behavior Trees segment similar behavior into components, but define

the integration of multiple components visually by interleaving their behavior thereby allowing a tree structure to be used. When contrasting the languages with Behavior Trees, we are interested in differences in how similar behavior is managed. We investigate whether the language avoids segmenting the system by grouping similar behaviors together and whether a network or a tree structure is used to describe the integration of the segments.

- **Methodology:** Behavior Trees, as part of Behavior Engineering, use a rigorous approach to translate the requirements and overcome the informal-formal barrier. In each case study we consider how each language deals with informal knowledge and whether there is an intuitive miraculous leap from the requirements to a specification or a design.

## A.1 Case Study I: State Machines

This case study investigates Wagner’s Microwave Oven model made using StateWorks Studio ([WSWW06], Appendix B, pp. 267-274). This case study is similar to the Microwave Oven used in Chapters 3,4 and 5. The requirements are as follows ([WSWW06], pp. 267-268):

“The oven has a Run push button to start (apply the power) and a Timer that determines the cooking length. Cooking can be interrupted at any time by opening the oven Door. After closing the Door the cooking is continued. Cooking is terminated when the Timer elapses. When the Door is open a Lamp inside the oven is switched on, when the Door is closed the Lamp is off.

The control system has the following inputs:

**Run** push button - when activated starts cooking,

**Timer** - while this runs keep on cooking,

**Door** sensor - can be true (door closed) or false (door open).

And the following outputs:

**Power** - can be true (power on) or false (power off),

**Lamp** - can be true (lamp on) or false (lamp off).

The knobs to set the power and timeout values are irrelevant for the control state machine. The behavior of the microwave oven control is determined by the Run push button, Timer and Door sensor.”

The resulting work product of the approach is a finite state machine which is represented in both graphical and tabular forms. The graphical form is known as a state transition diagram and the tabular form is known as a state transition table. Each state in the state transition diagram is associated with an individual state transition table which shows the entry actions, exit actions and input actions of the state. It should be noted that the state transition table is specific to StateWorks version of state machines. According to Wagner [Wag04], “Actually, there is no standard notation for complete specification of state machine behavior. For instance StateWORKS uses a special transition table for this purpose.”.

Figure A.1 shows the state transition diagram resulting from the requirements of the Microwave Oven. Normally each state in the state transition diagram is associated with an individual state transition table but for brevity’s sake we have collated and summarised the information into a single state transition table. The summary of the individual state transition tables is shown in Table A.1.

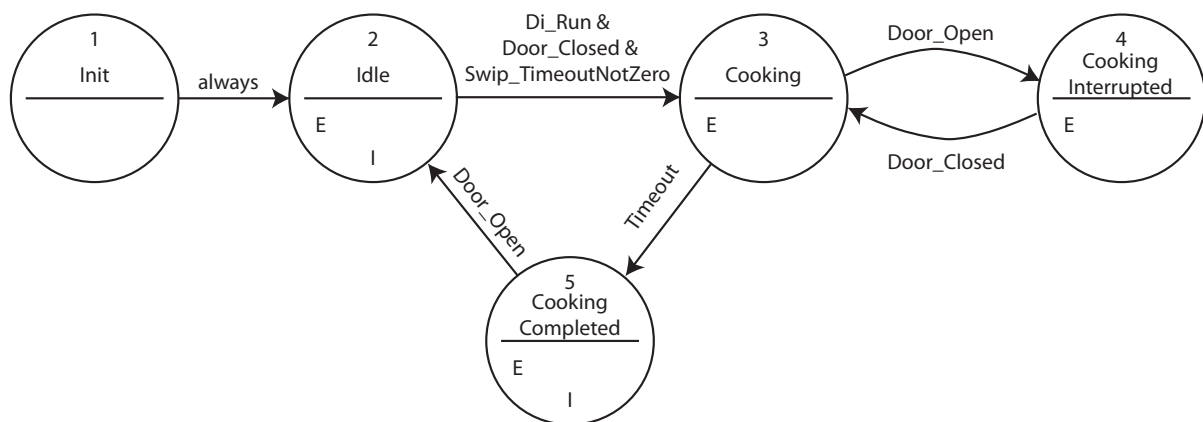


FIGURE A.1: State Transition Diagram

State	Condition	Output
Init	-	-
Idle	Entry Action	Swip_Timeout_On
	Door_Closed	LampOff
	Door_Open	LampOn
Cooking	Entry Action	LampOn, PowerOn, Timer_Start
CookingInterrupted	Entry Action	PowerOff, Timer_Stop
CookingCompleted	Entry Action	LampOff, PowerOff, Timer_Reset
	Door_Open	LampOn

TABLE A.1: State Transition Table

### A.1.1 Transitions

The state transition diagram and state transition table shows two different types of transitions. Transitions that result in a change of state are displayed graphically in the state transition diagram. Transitions that occur within a state (without a resulting state change) are known as actions and are shown in the state transition table.

Each state can have an entry action, an exit action and several input actions. Entry actions (E) occur on entry to a state. Exit actions (X) occur on exit from a state. Input actions (I) occur when input is received while in a state. These actions are not defined in the state transition diagram. The presence of actions associated with each state of the state transition diagram, is instead indicated using the abbreviations E, X, and I.

Transitions are expressions that are composed of virtual inputs that can be linked with logical AND and OR operators. There are several types of virtual inputs allowing transitions to be based upon digital input, analogue input using switchpoints, counters, timers, and the state of other state machines. Actions link virtual inputs to virtual outputs such as digital output, numerical output and alarms.

The types of virtual inputs and outputs is stored in a dictionary associated with the state machine. Without the aid of this dictionary, the type of a virtual input in the state transition diagram or state transition table can only be determined by naming conventions. In Figure A.1, Di\_Run indicates a digital input and Swip\_TimeoutNotZero indicates a switchpoint. Based upon the name Door\_Closed, however, it is not clear that the virtual input is a digital input value of low.

Even though state machines use formalised transitions, the comprehension of state



machine transitions are hindered by two issues. The first issue is that not all transitions are described graphically, so to fully comprehend the model several state transition tables must also be examined. The second issue is that the type of virtual input being used in the transition expression is only informally referenced using naming conventions and must be determined using the dictionary.

### A.1.2 Segmentation

The primary means of segmentation in state machines is to organise similar behavior into states which can then contain several actions. This approach has the shortcoming that it is susceptible to state space explosion. State space explosion occurs because increasing the number of states in a state machine results in a much larger increase in the number of potential interdependencies between the states. This problem can be handled by using several state machines linked together, commonly in a hierarchy. This raises the issue of a suitable methodology for determining which states should be grouped together to form separate state machines and also for determining how these state machines can be integrated together.

### A.1.3 Methodology

The documentation of the case study on the Stateworks website states [Sta06],

“... the requirements as specified in the beginning [of this exercise] were not very detailed. [This] is often the situation with which we are confronted in a real project. Not till we see the first solution [do] we ‘discover’ that it is not what we have expected ... To start a project with an incomplete specification is not desirable, but it is a common practice and often unavoidable.”

This advocates a construct-by-correction approach with unnecessary iteration. A construct-by-correction approach is necessary because a scaleable methodology is not used to translate the requirements into a specification.

Consider the requirement, when the Door is open a Lamp inside the oven is switched on, when the Door is closed the Lamp is off. In the case study, Door is captured as a digital

input object and Lamp is captured as a digital output object. Door.Closed and Door.Open represent the two states of the Door digital input and Lamp.On and Lamp.Off represent the two states of the Lamp digital output.

Door.Open and Door.Closed are described both in the state transition diagram and the state transition table which complicates determining the coverage of this requirement. When the oven is in the idle state, the requirement is satisfied in the state transition table. When the oven is in the cooking and cooking interrupted states, however, the lamp remains on regardless of the state of the door, violating the requirement. It is not until the cooking completed state that the requirement is satisfied again.

This mismatch between the requirement, the state transition diagram, and the state transition table clearly indicates that even for this small example the diagram does not relate to the requirements. The violation of the requirement may indicate a missing requirement that the lamp should be on while the oven is cooking. While this is possible to determine with this small case study, this mismatch is unlikely to be located as easily in a larger system.

The methodology for avoiding space explosion by using hierarchical state machines also has issues. According to Wagner ([WSWW06], p114), "... building of complex control systems does not mean that we start with a complex state machine and then try to partition it. We rather specify several state machines for specific tasks and then link them together." The issue here is that this bottom-up composition requires a top-down decomposition which is by its nature iterative and informal ([WW04], ), "It is usually best to implement such [hierarchical] systems from the lower levels upwards, after an initial top-down planning phase which gives a rough idea of the anticipated structure." Issues with timing and synchronisation are also possible if the means of communication between state machines is unrestricted. While these issues are able to be avoided by a practised modeler, they are not intrinsically avoided by the modeling technique itself.

## A.2 Case Study II: StateCharts

This case study investigates two models: an Aircraft statechart and an Elevator xtUML model. The aircraft statechart by Hull et al ([HJD05], p.51) that is used for contrasting

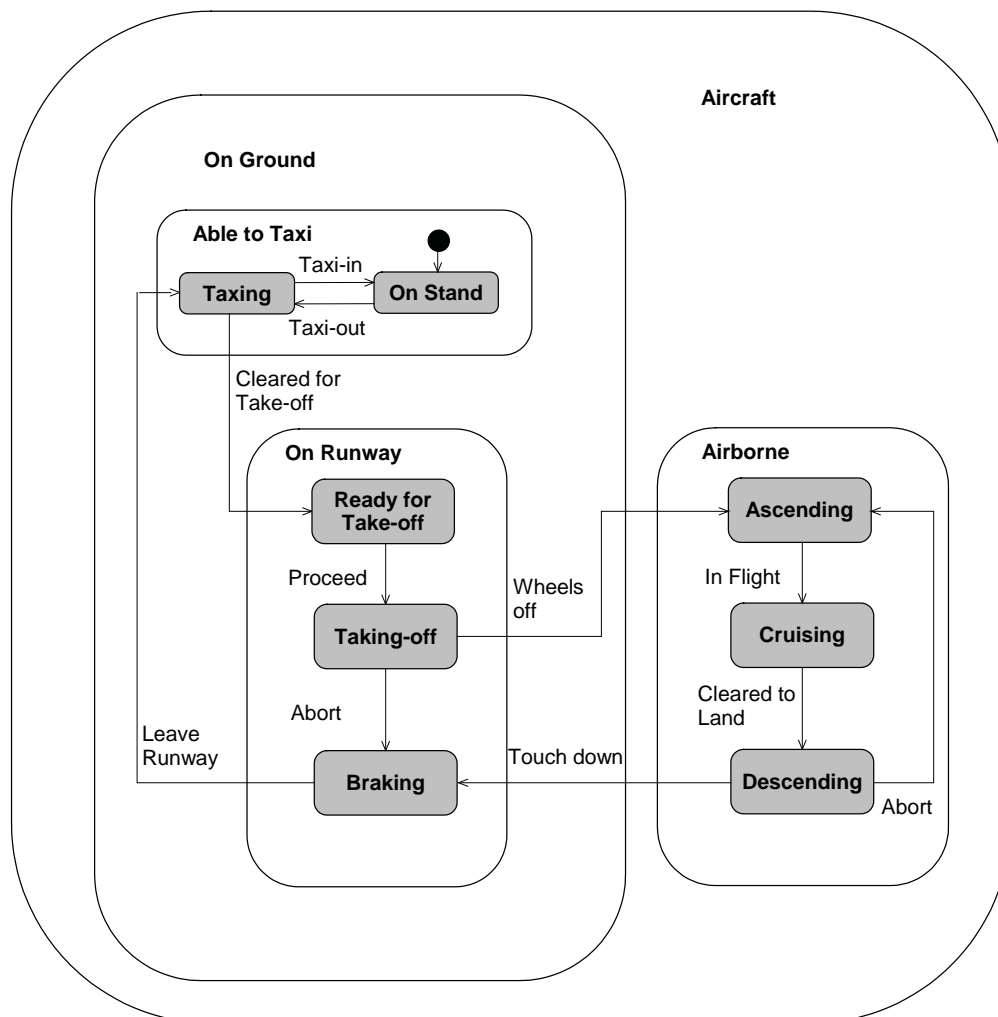


FIGURE A.2: StateChart of an Aircraft

transitions and segmentation with BTs is shown in Figure A.2.

Contrasting the methodology used to build statecharts is difficult, because statecharts are a diagram in the UML which is designed to be compatible with many different methodologies. We avoid this issue by using an xtUML model of an Elevator by Starr [Sta01] to discuss the methodology of statecharts. xtUML uses a restricted form of statecharts together with an action language that can be used to make executable models.

### A.2.1 Transitions

Statecharts can use formalised transitions but these are based on action languages which are not standardised [CD08]. The transitions used in Figure A.2, however, are informal, which has several implications. Firstly, it is not clear what is responsible for triggering the transition. For example, the transition *Cleared for Take-off* is likely triggered by a flight controller, but this cannot be indicated in the diagram. Secondly, it is not clear how the transition is triggered. The *Cleared for Take-off* transition could be triggered by another statechart, a message received across some communication middleware, or input from a graphical user interface.

Because this statechart does not include how transitions are triggered, it would appear to be much simpler than a comparative BT. However, including the additional information required to describe the triggering of transitions by other statecharts significantly complicates the statechart as shown in Figure A.3.

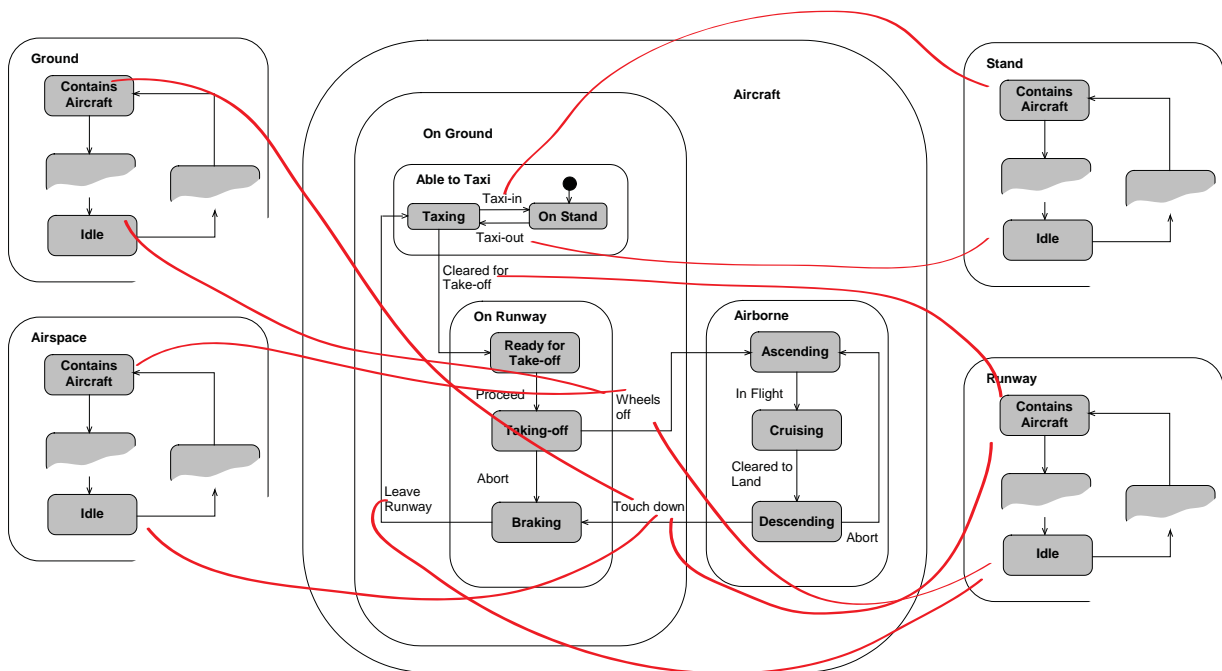


FIGURE A.3: Aircraft StateChart modified to show interdependencies

### A.2.2 Segmentation

The aircraft statechart uses a hierarchy to organise several segmented statecharts, similarly to state machines. Statecharts' use of hierarchies is quite liberal, with transitions being able to occur at multiple points inside a segment of a statechart. Allowing transitions to occur at multiple points inside a segmented statechart removes the abstraction benefits that can be provided by encapsulation. Based on Figure A.2, the information that is grouped into a segment also appears quite liberal. For example, segments are created for a component (Aircraft), a location (On Ground / On Runway), a condition (Able to Taxi) and a state (Airborne).

### A.2.3 Methodology

Figures A.4-A.10 show an xtUML model of an elevator consisting of a partial class collaboration diagram, a partial class diagram, and several statecharts. The nature of these diagrams makes it unlikely an xtUML model could be created by individually translating the requirements and instead would require a miraculous leap of intuition. This can be demonstrated by showing the coverage of a single requirement across the diagrams of the xtUML model. Consider a fictitious requirement that describes some of the behavior of the elevator:

If the Building experiences an earthquake, all Elevator Banks are deactivated, and each shaft associated with the bank is taken out of service. If the elevator cabin is moving when taken out of service, it stops at the nearest floor. Once the elevator cabin is stopped at a floor, the cabin doors are opened and remain held open.

The partial class collaboration diagram shown in Figure A.4 partially describes the requirement by showing the interactions between the classes but the causality is not shown. That is, the cause of the interaction between the two classes and the result of the interaction between the two classes is not visible in the class collaboration diagram.

The partial class diagram shown in Figure A.5 presents a large problem for creating a scaleable methodology. It is very hard to create a Class Diagram without an understanding

of the system as a whole, which makes the task of translating the information from the requirements very difficult.

Figures A.6-A.10 show the statecharts related to the requirement. In each of these statecharts we can see the interaction received from other diagrams and the behavioral effect of the interaction. The behavioral cause of the interaction is only visible by tracing the interaction to the originating statechart.

The example requirement we described spans across seven diagrams. This has several implications. There is a risk of inconsistencies because diagrams may not conform to each other, particularly in the midst of development. It is also hard to know when modeling is finished because the diagrams used make it unlikely each requirement is individually translated. xtUML helps manage these issues to a degree by executing the model and testing the result against the requirements. However, this construct-by-correction approach has the disadvantage of requiring extra effort to locate and resolve issues that are created due to poorly managed complexity.

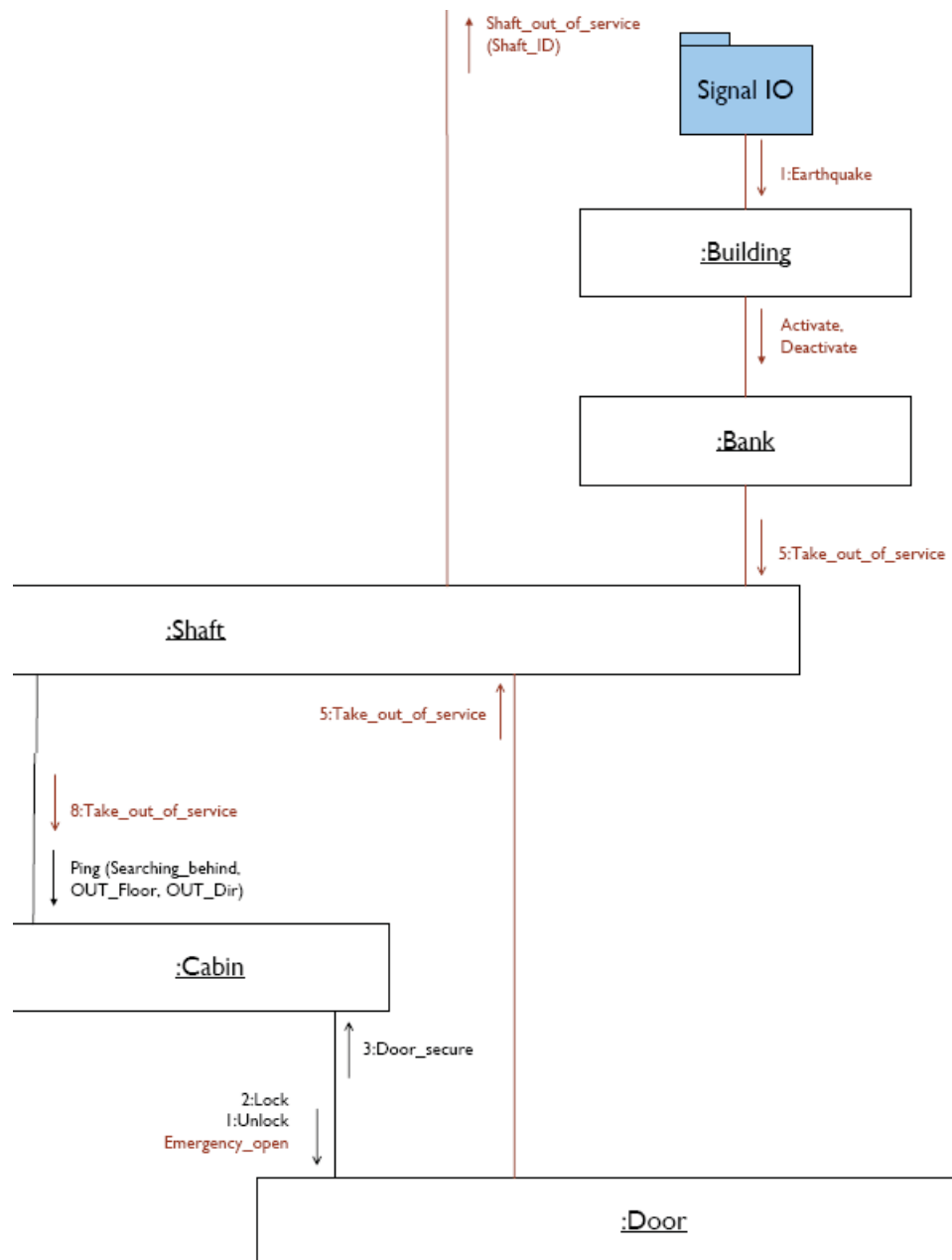


FIGURE A.4: Partial Class Collaboration Diagram

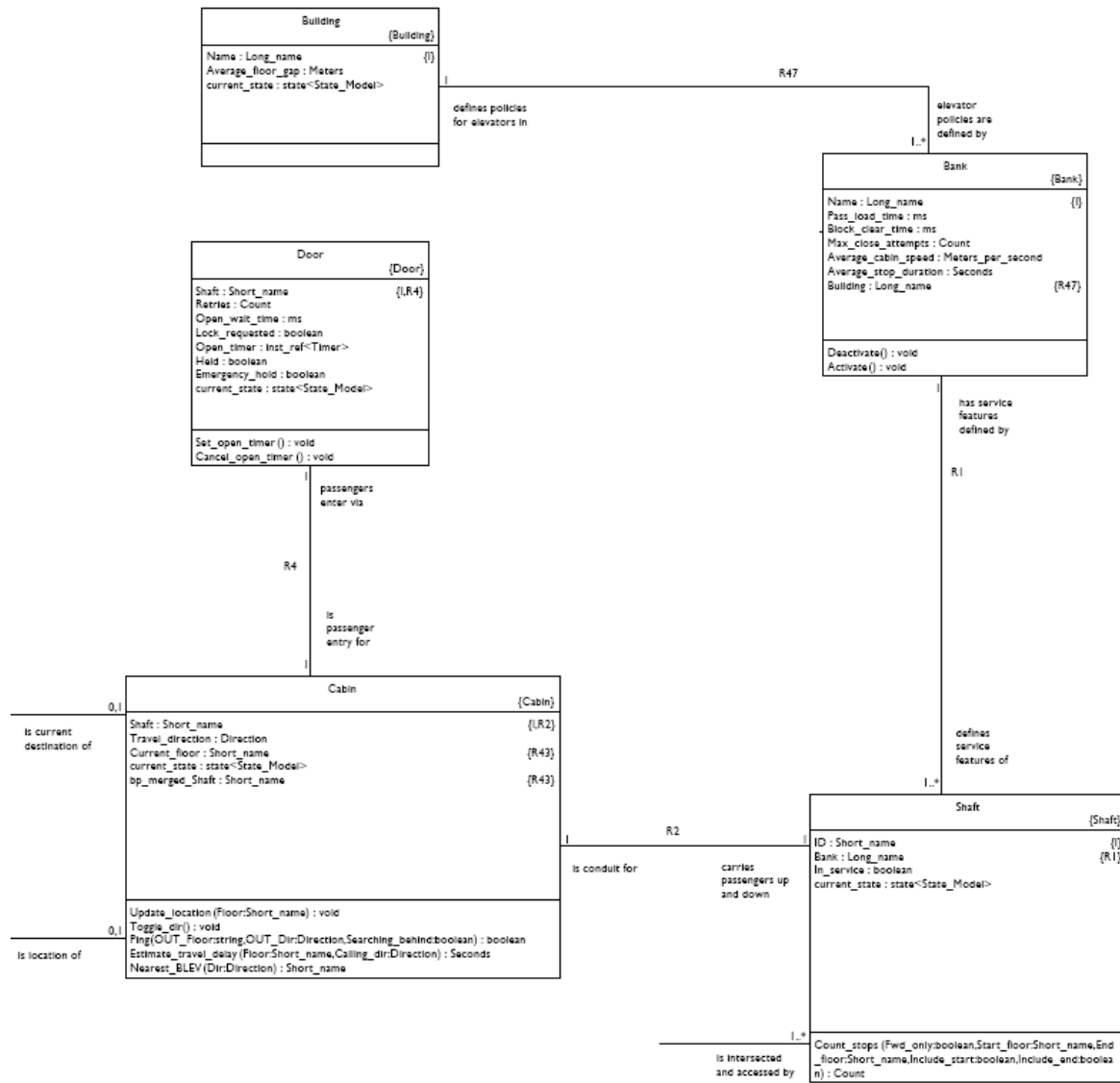


FIGURE A.5: Partial Class Diagram



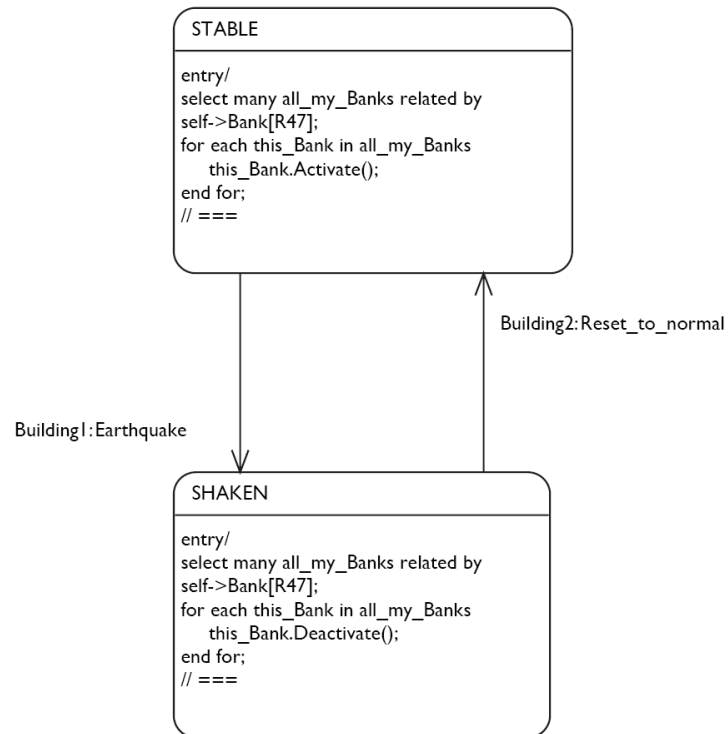


FIGURE A.6: Partial Building StateChart

```

// Bank::Deactivate ()
//
// Take all of my Shafts out of service
//
select many my_inservice_Shafts related by self->Shaft[R1]
  where(selected.In_service);
for each this_Shaft in my_inservice_Shafts
  this_Shaft.In_service = false;
generate Shaft5:Take_out_of_service () to this_Shaft;
end for;
// ===
  
```

FIGURE A.7: Partial Bank Operations in Action Language

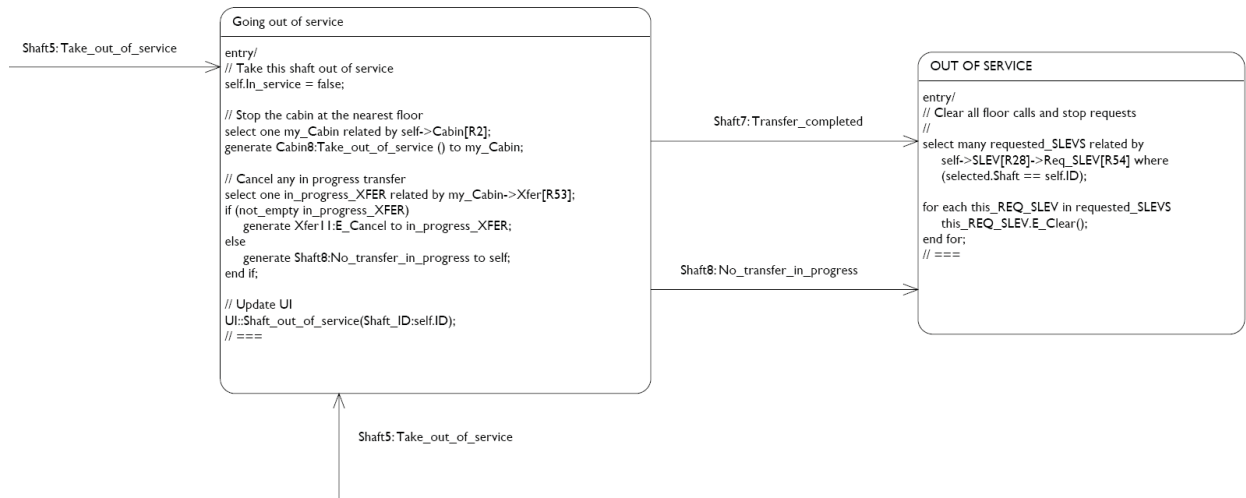


FIGURE A.8: Partial Shaft StateChart

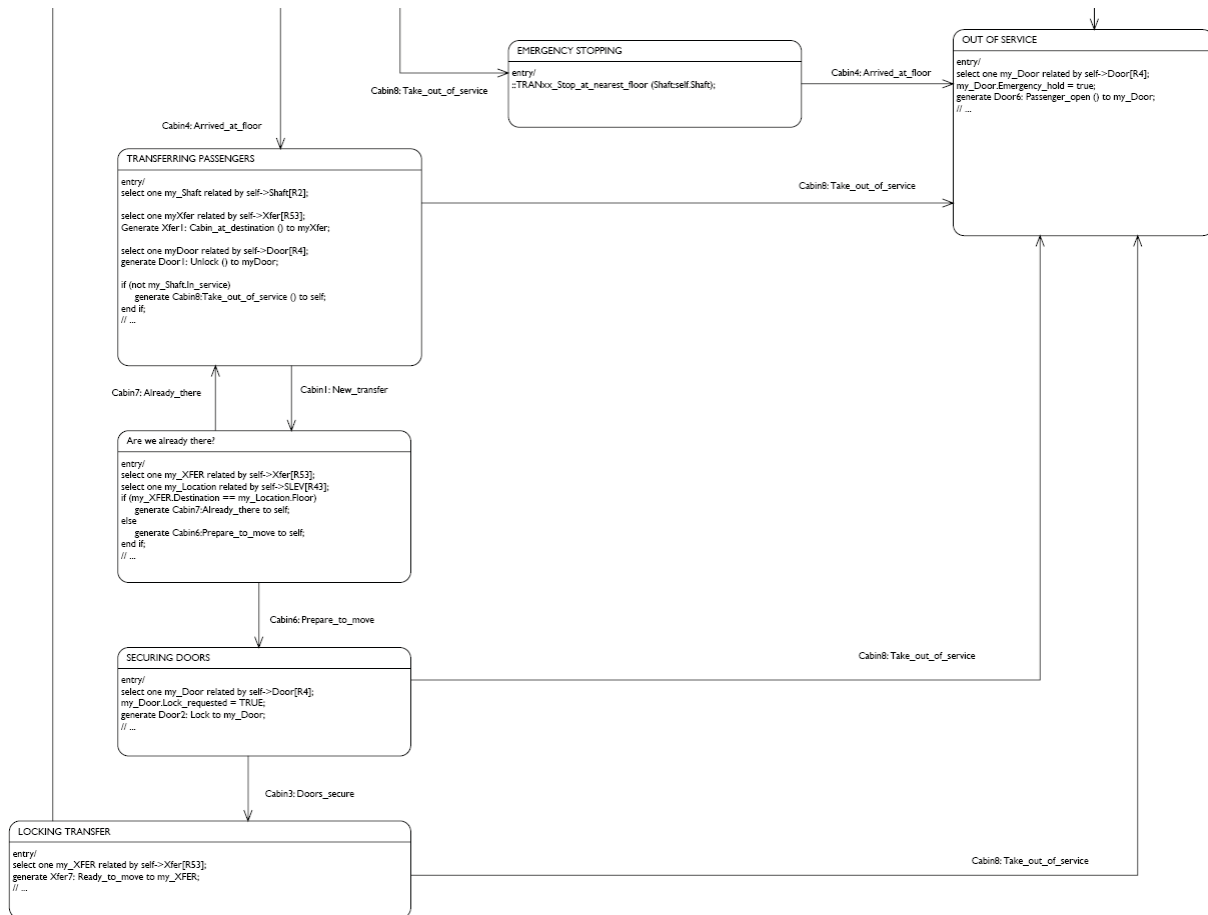


FIGURE A.9: Partial Cab StateChart

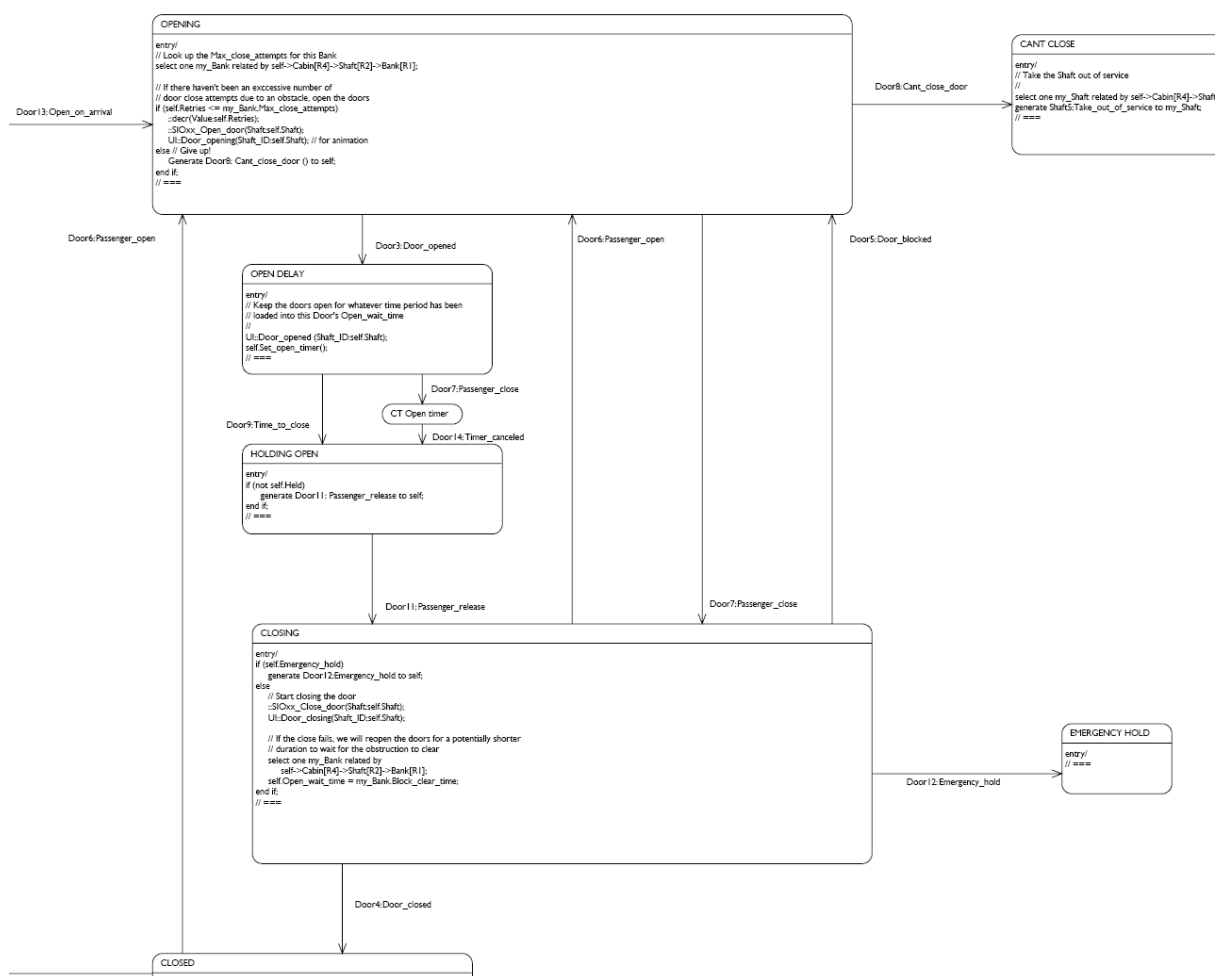


FIGURE A.10: Partial Door StateChart

### A.3 Case Study III: Exogenous Connectors

This case study investigates Exogenous Connectors (EC) [LEW05] using the case study of an automated train protection (ATP) system that is also used in Chapter 6. The EC model of the ATP system [LLW06] is used to demonstrate an approach to systems development using composite components. The requirements of the ATP system are repeated in Table A.2.

Requirement	Description
R1	The ATP system is located on board the train. It involves a central controller and five boundary subsystems that manage the sensors, speedometer, brakes, alarm and a reset mechanism.
R2	The sensors are attached to the side of the train and detect information on the approach to track-side signals, i.e. they detect what the signal is displaying to the train driver.
R3	In order to reduce the effects of component failure three sensors are used. Each sensor generates a value in the range 0 to 3, where 0, 1 and 2 denote the danger, caution, and proceed signals respectively. The fourth sensor value, i.e. 3, is generated if an undefined signal is detected, e.g. may correspond to noise between the signal and the sensor.
R4	The sensor value returned to the ATP controller is calculated as the majority of the three sensor readings. If there does not exist a majority, then an undefined value is returned to the ATP controller.
R5	If a proceed signal is returned to the ATP controller, then no action is taken with respect to the train's brakes.
R6	If a caution signal is returned to the ATP controller, then the alarm is enabled within the driver's cab. Furthermore, once the alarm has been enabled, if the speed of the train is not observed to be decreasing, then the ATP controller activates the train's braking system.
R7	In the case of a danger signal being returned to the ATP controller, the braking system is immediately activated and the alarm is enabled. Once enabled, the alarm is disabled if a proceed signal is subsequently returned to the ATP controller.
R8	Note that if the braking system is activated, then the ATP controller ignores all sensor input until the system has been reset.
R9	If enabled, the reset mechanism deactivates the train's brakes and disables the alarm.

TABLE A.2: Requirements of the ATP system

### A.3.1 Transitions

Figure A.11 shows the EC model of the ALRM\_BRKS\_Control composite component and Figure A.12 shows its XML interface definition. Based upon figure A.11, transitions in the exogenous connectors diagram appear to be informally defined graphically and formally defined in XML. The connector type is only discernable in the graphical diagram from a naming convention where *Inv* refers to an invocation connector, *Sel* refers to a selection connector, *Cond* refers to a condition connector and *Seq* refers to a sequential connector. In the graphical diagram, the parameters of each connector are placed along the path between two connectors. These parameters, however, do not all match the interface definition in XML shown in Figure A.12.

It is also interesting to note that in the EC model, enabling the alarm and checking that the alarm is enabled requires two separate methods. In a BT this is managed by using different formalised types, allowing the same name to be used in both cases. Enabled with

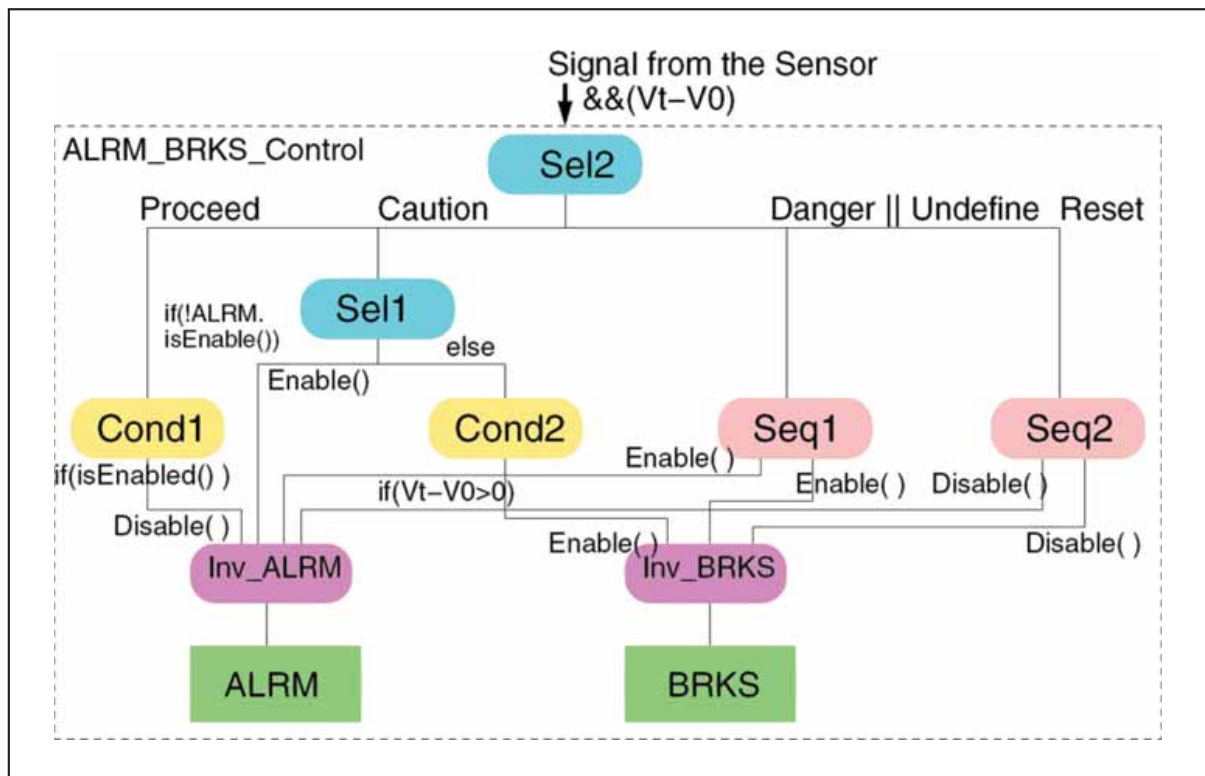


FIGURE A.11: Exogenous Connectors Composite Component Alarm\_Brakes\_Control

a state realisation behavior type corresponds to Enabled() and Enabled with a selection behavior type corresponds to isEnabled().

Another issue is that ALRM.isEnabled() does not appear to be using the invocation connector Inv\_ALRM. This may be because it is unclear how the condition connector would show that as part of its parameter it must invoke the ALRM invocation connector and then the result may either invoke the ALRM invocation connector or the BRKS invocation connector.

```

< Component >
  < Name > Alarm_Brakes_Control < /Name >
  < Operation_Specification >
    Object execute (Stringcondition, StringMethodName, VectorParas)
    < Condition Case = "Proceed" >
      < If Case = "Inv_ALRM.IsEnabled()" > Inv_ALRM.Disable() < /If >
    < /Condition >
    < Condition Case = "Caution" >
      < Condition Case = "!ALRM.IsEnabled()" >
        Inv_ALRM.Enable()
      < /Condition >
      < Condition Case = "ALRM.IsEnabled()" >
        < If Case = "(Vt - V0) > 0" > Inv_BRKS.Enable() < /If >
      < /Condition >
    < /Condition >
    < Condition Case = "Danger" or "Undefine" >
      < Seq > Inv_ALRM.Enable() < /Seq >
      < Seq > Inv_BRKS.Enable() < /Seq >
    < /Condition >
    < Condition Case = "Reset" >
      < Seq > Inv_ALRM.Disable() < /Seq >
      < Seq > Inv_BRKS.Disable() < /Seq >
    < /Condition >
  < /Operation_Specification >
< /Component >

```

FIGURE A.12: Interface Definition of ALRM\_BRKS\_Control Composite Component

### A.3.2 Segmentation

Exogenous connectors use the same principle as Behavior Trees for segmentation, that is they use connectors to integrate the encapsulated computation of components. The approach taken by ECs to do this, however, does have some deficiencies. Composite components in ECs have only one entry point using an execute method. An example of the issues this can cause is where the *Sel2* connector in Figure A.11 receives both the signal from the sensor and the speed change ( $V_t - V_0$ ). This introduces some redundancy because the speed change value is only required by the *Cond2* connector if the signal is caution.

Another issue with ECs is that the final level is always invocation connectors, which are used to execute methods inside components. Because the invocation connectors are only placed on the final level, it becomes hard to follow paths from the previous level of connectors to their invocations. In contrast, Behavior Trees' state realisations (which are semantically similar to invocation connectors) can be used throughout the diagram.

ECs also can perform course-grain segmentation using composite components constructed from several components. However, this requires a methodology to determine the best order in which components should be combined to form composite components.

### A.3.3 Methodology

After the ALRM\_BRKS\_Controller (ABC) composite component the remaining EC model of the ATP system is built as follows. Figure A.13 shows the composition of the ABC composite component with the RSET component to form the RSET\_ALRM\_BRKS\_Controller composite component. This is then combined with the SNRS and SPDO components to create the ECs model of the ATP system shown in Figure A.14.

The reason for determining that ALRM and BRKS should form the first composite component according to Lau et al is [LLW06]: "Considering in the ATP system, the alarm and brakes react to all the signals, we firstly construct a composite component that consists of ALRM and BRKS.". While it is reasonable for this small case study to quickly gain an understanding of the suitable components to begin with, this is unlikely to be the case in a much larger case study. The order of construction of the composite components also seems

to be important, as poor ordering could result in more data being passed through the execute methods of higher-level composite components to components encapsulated at lower-levels.

Another issue with the EC methodology is crossing the informal-formal barrier. The resolution of issues such as handling the undefined signal as a danger signal is visible in the model, but it is not evident that there was an issue in the first place. This is demonstrated by how the EC model handles conflicts in requirements R7-R9 of the ATP system. The EC model satisfies R7 by disabling the alarm when it receives a proceed signal. It also satisfies R9 by disabling the alarm and brakes when a reset signal is received. How this is done, however, allows a reset to be received at any time making it possible for the ATP system to attempt to disable the alarm and brakes again when they are already disabled. Depending on how rigorously the ALRM and BRKS components are built, it is possible that this could introduce faults. The EC model also does not resolve missing behavior in the requirements with the result that there is no means of disabling the reset mechanism. Finally, the EC model violates R8 which states that sensor input should be ignored until the system is reset. The structure of the ABC composite component would actually make satisfying this requirement difficult because the reset signal is sent by the same means as a sensor input. This also brings into question how reusable the composite components are, because the ABC composite component is already implicitly designed to be integrated with the RSET component due to including a reset signal.



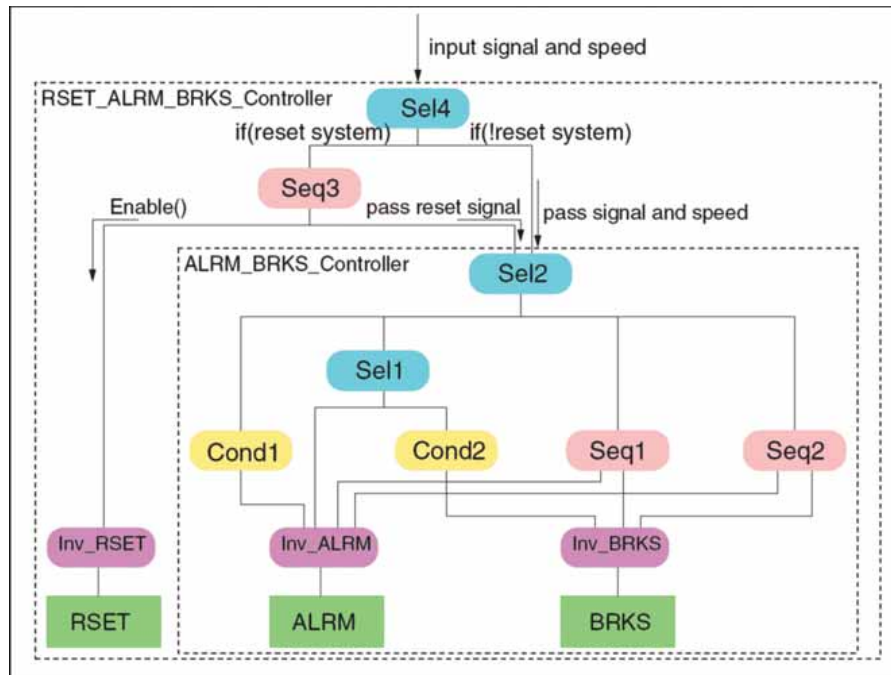


FIGURE A.13: Exogenous Connectors Composite Component Reset\_Alarm\_Brakes\_Control

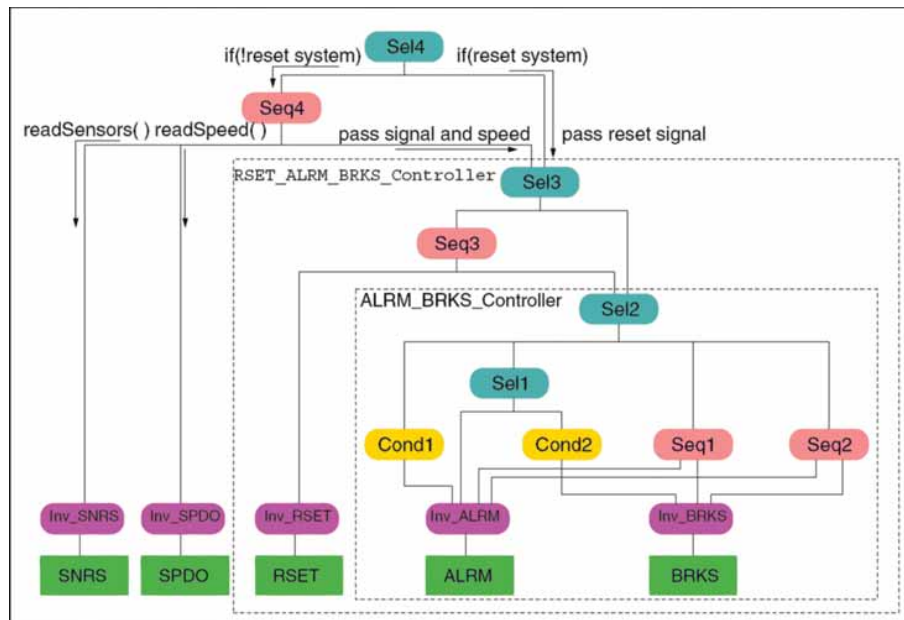


FIGURE A.14: Exogenous Connectors model of the Automated Train Protection System



# B

## Behavior Tree Language

This appendix is intended to be used as a reference for the Behavior Tree language. It consists of the naming conventions for variables, nodes, relations, and trees that are used throughout this dissertation. It also describes the notation and syntax for Behavior Tree nodes, tags, branching, and operators.

## B.1 Naming Conventions

### B.1.1 Variable Naming Conventions

- Component names should be capitalised e.g OVEN
- The first letter of a behavior should be capitalised e.g Open
- The first letter of an attribute should be lowercase e.g. timer

Variable	Description
$N, N_i$	Behavior Tree Nodes
$T, T_i$	Behavior Trees
$C, C_i$	Components
$C\#$	A Component Instance
$s$	A State of a Component
$e$	An Event
$a$	An Attribute of a Component
$b$	A Branching Condition of a Component

TABLE B.1: Variable Naming Conventions

### B.1.2 Node Naming Conventions

Label	Name	Description
A	Component Name	Specifies a component
B	Behavior	Specifies the behavior associated with the component
C	Operator	Describes threaded behavior that is linked to the matching node in the tree
D	Label	An optional label for disambiguation (in case a node appears elsewhere with the same component and behavior)
E	Behavior Type	Delimiters on the behavior indicating the type of behavior involved
F	Traceability Link	A reference to the requirements document
G	Traceability Status	Indicates how the node relates to the traceability link
H	Tag	The box on the left-hand side of the node (may be omitted in different contexts)
I	Behavior Tree Node	A node consisting of all or some of the information above

TABLE B.2: Elements of a Behavior Tree Node

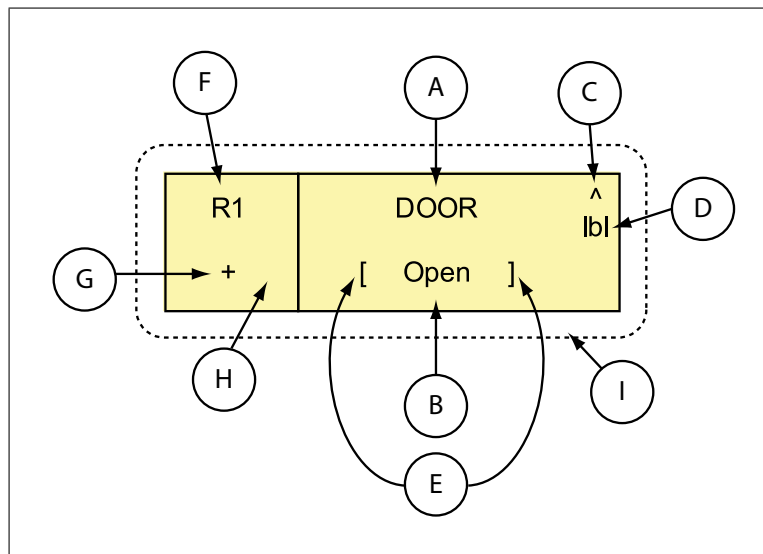


FIGURE B.1: Behavior Tree Node Naming Conventions

### B.1.3 Relation Naming Conventions

Label	Name	Description
A	Primary Component & Behavior	The component and behavior that form the relation
B	Related Component	Component (and optional behavior) related to the primary component and behavior
C	Qualifier	Specifies the type of the relation. Must be one of What, Where, When, Why, Who or How
D	Preposition	Further qualifies the relation to remove potential ambiguity
E	Secondary Relation	The related component is linked to the primary component using a forward slash (/). Multi-level relations can be formed by using multiple forward slashes

TABLE B.3: Elements of a Behavior Tree Relation

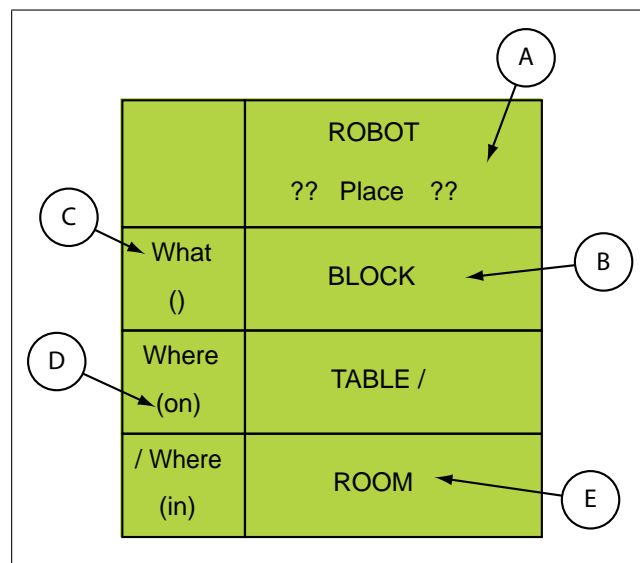


FIGURE B.2: Behavior Tree Relation Naming Conventions

### B.1.4 Tree Naming Conventions

Label	Name	Description
A	Ancestor Node	Any node which appears in a direct line between the node of interest and the root node of the tree
B	Parent Node	An immediate ancestor
C	Sibling Node	A node which shares the same parent
D	Sibling Branch	A subtree with a sibling node as its root
E	Child Node	A node immediately below the node of interest
F	Descendant	Any node appearing below the node of interest

TABLE B.4: Nodes of a Behavior Tree

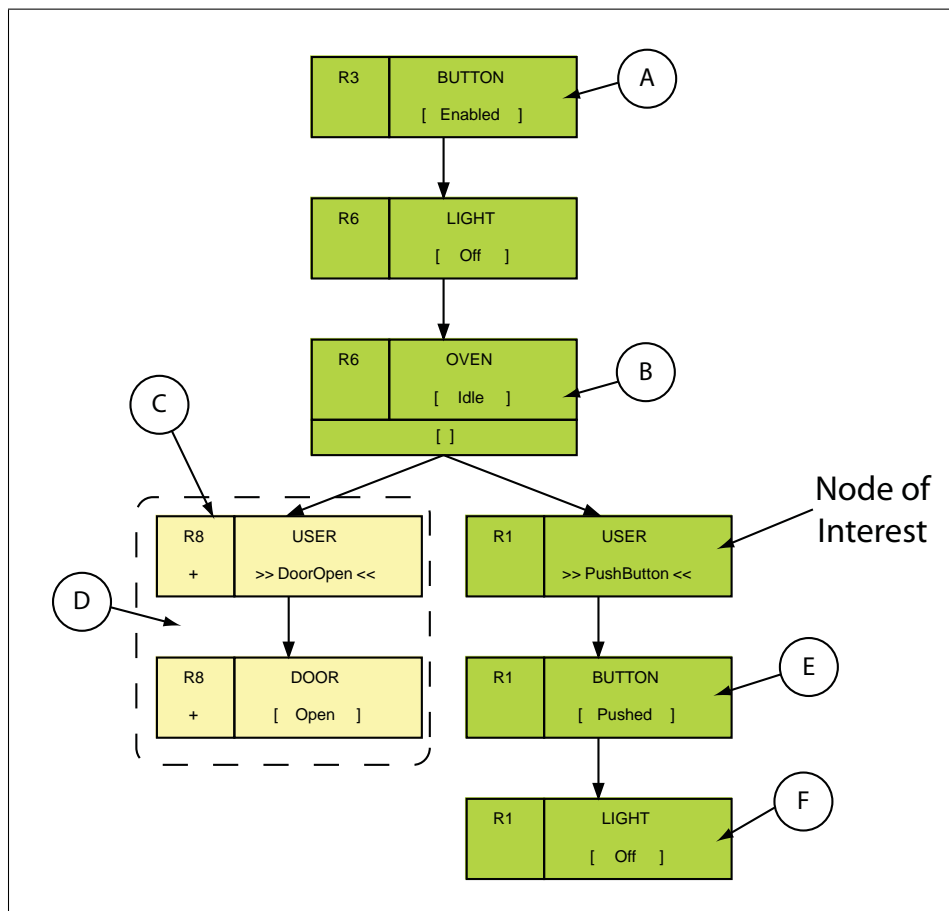


FIGURE B.3: Behavior Tree Tree Naming Conventions

### B.1.5 Tree Branch Naming Convention

Label	Name	Description
A	Root Node	The first node in a tree (does not have a parent)
B	Edge	A connection between two nodes
C	Leaf Node	A node with no children
D	Branch	A subtree of the node of interest

TABLE B.5: Branches of a Behavior Tree

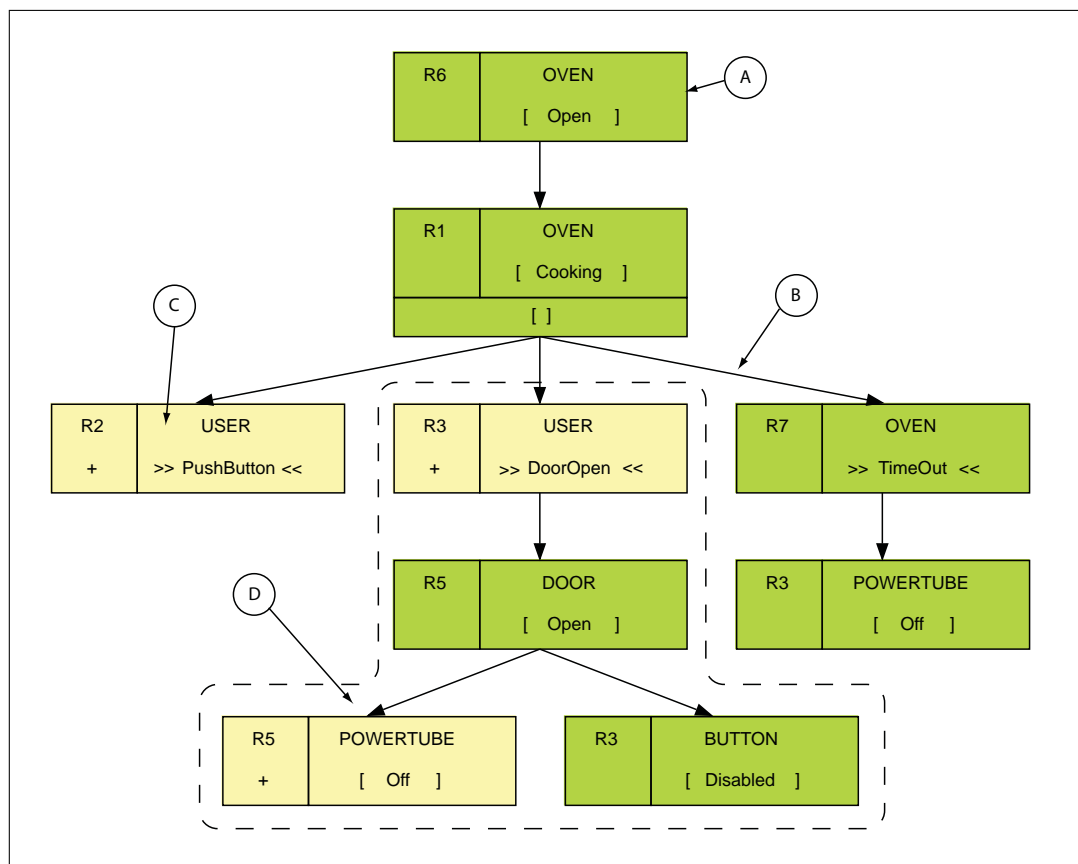
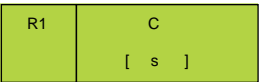
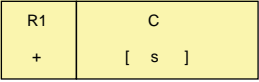
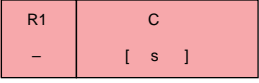
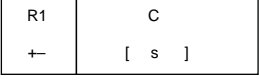

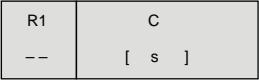


FIGURE B.4: Tree Branch Naming Convention


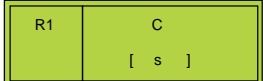
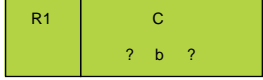
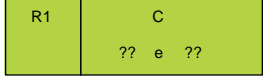
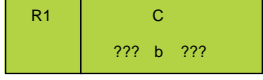
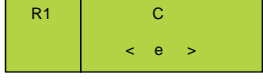
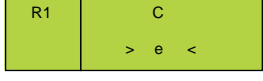
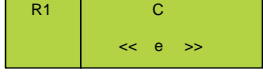
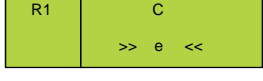



## B.2 Behavior Tree Notation & Syntax

### B.2.1 Node Tags

Type	Graphical Notation	Description
Original		No traceability status indicates that the behavior is stated in the original requirements. The color “green” is used for original requirements.
Implied		The “+” traceability status indicates that the behavior is not explicitly stated in the original requirements but is implied by the requirement. The color “yellow” is used for implied behavior.
Missing		The “-” traceability status indicates that the behavior is missing from the original requirements and is needed for completeness. The color “red” is used for missing behavior.
Design		The “+” traceability status indicates that the behavior is a refinement of the original requirements, indicating that the behavior is implied but the detail to describe it is missing.
Updated		The “++” traceability status indicates that the behavior has been added in the post-development or maintenance phase. The color “blue” is used for updated behavior. Where there are different series of changes / upgrades we use ++v1.0, ++v2.0, etc to indicate the particular upgrade series.
Deleted		The “-” traceability status indicates that the behavior has been deleted from the behavior tree. The color “grey” is used for deleted behavior, but the nodes may also be hidden optionally by using tool support.

## B.2.2 Basic Nodes









Type	Graphical Notation	Description
State Realisation		Component <i>C</i> realises state <i>s</i> .
System State Realisation		This is a state realisation decorated with a double box to indicate the component is a system component in the current context. There can only be one system component in each context.
Selection		If condition <i>b</i> evaluates to true, then pass control to child nodes otherwise terminate.
Event		Wait until event <i>e</i> is received.
Guard		Wait until condition <i>b</i> evaluates to true, then pass control to child nodes.
Internal Output		Generate input <i>e</i> and send to the system.
Internal Input		Wait for input <i>e</i> from the system.
External Output		Generate output <i>e</i> and send to the environment.
External Input		Wait for input <i>e</i> to be received from the environment.
Empty Node		Empty Nodes can be used together with labels to be origins or destinations of node operators. Empty Nodes are also useful for grouping child nodes into multiple branch types.

### B.2.3 Behavior Tree Composition

Type	Graphical Notation	Description
Sequential Composition		Execute $N$ , passing control to tree $T$ . The behavior of concurrent BTs may be interleaved between $N$ and $T$ .
Atomic Composition		Execute $N_1$ immediately followed by $N_2$ , passing control to tree $T$ . The behavior of concurrent BTs may not be interleaved between $N_1$ and $N_2$ .
Parallel Branching		Execute $N$ , passing control to both $T_1$ and $T_2$ .
Alternative Branching		A nondeterministic choice is made between $T_1$ and $T_2$ , depending on which is ready to execute (not blocked)

## B.2.4 Node Operators

- Operators on source nodes match against the Component, Behavior, Behavior Type and Label (if present) of the destination node.
- An operator may be prefixed by a label and a fullstop to refer to a destination node with a label e.g. *lbl.^* indicates to revert to destination node with label *lbl*.

Type	Graphical Notation	Description
Reference		Behave as the destination node. The destination node must appear in an alternative branch to the origin.
Reversion		Behave as the destination node. The destination node must be an ancestor. All sibling behaviour is terminated.
Branch Kill		Terminate all behavior associated with destination tree.
Synchronisation		Wait for destination node (or nodes).
May		The node may execute normally, or may have no effect.
Conjunction		The operators &,   and XOR correspond to logical conjunction, disjunction and exclusive or respectively.
Disjunction		
Exclusive OR		



## Investigating the Hardware Component Model

In this appendix we use a small case study to investigate the characteristics of the hardware component model that has allowed hardware development to achieve scalability and reuseability. The case study is developed using a group of digital logic integrated circuits (ICs) called Transistor-Transistor Logic (TTL) [Lan74]. TTL ICs were invented in the 1960's and remained popular until the 1990's when they began to be replaced by programmable logic integrated circuits. TTL ICs were created for a wide variety of functions including gates, memory/counting devices, shift registers, decoders, data selectors and an arithmetic logic unit. The size of TTL ICs are small by today's standards, varying between 20-200 transistors depending on the complexity of the chip's functionality.

This case study involves building a simple 12-hour digital clock using the one hertz signal (one cycle a second) provided. The digital clock must utilise the one hertz signal to display time in hours, minutes and seconds. Figure C.1 shows the decomposition of this task into

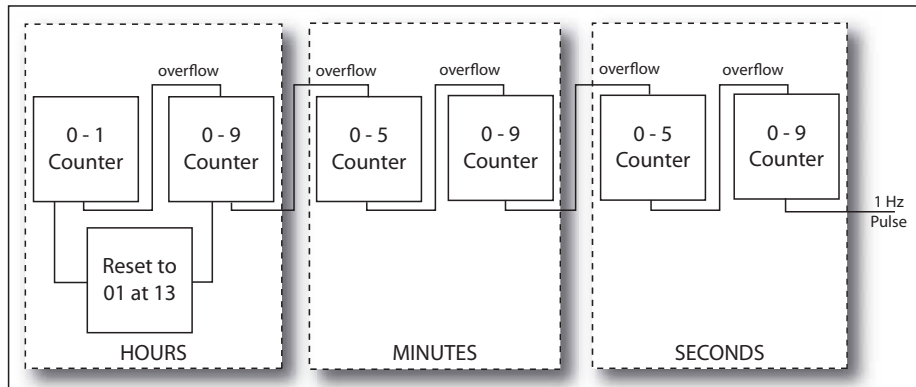


FIGURE C.1: Overview of a simple Digital Clock

modules for the hours, minutes and seconds. Both the seconds module and the minutes module require a counter from zero to nine. When this counter overflows, a counter from zero to five is triggered. The hours module requires more complex logic to count from one to twelve. The three modules are connected by using the overflow of the seconds module to trigger the minutes module and an overflow of the minutes module to trigger the hours module.

The central functionality required for this case study is a counter. In electronics, counters are composed of several flip-flops, each storing one-bit of data. The left of figure C.2 shows an internal gate-level view of the 7493 TTL IC, a four-bit binary counter <sup>1</sup>. The four bit binary counter counts to 15 and then overflows and resets to zero.

Our digital clock requires a decade counter that only counts from zero to nine. This is achieved in the hardware component model by modifying the functionality of the 7493 to create another IC, the 7490 Decade counter. The right of Figure C.2 shows the resulting 7490 IC. The 7490 includes functionality to reset the counter when it reaches 9 (0b1001) based on when the second and third flip-flops are low and the fourth flip-flop is high. It also includes functionality to allow the counter to be reset to 9 based on an external signal.

This internal functionality is not directly accessible when using a hardware component,

<sup>1</sup>For those not familiar with the notation of hardware schematics a brief overview follows. A filled circle between two or more intersecting paths indicates the paths are connected. An unfilled circle (or bubble) indicates the signal along the path is inverted. The block accepting two signal paths with a straight back and rounded curved front is an AND gate. The block accepting two signal paths with a curved back and a pointed front is an OR gate. The master-slave flip-flops are rectangular with the J, K, CLOCK, Q (and /Q) inputs and outputs

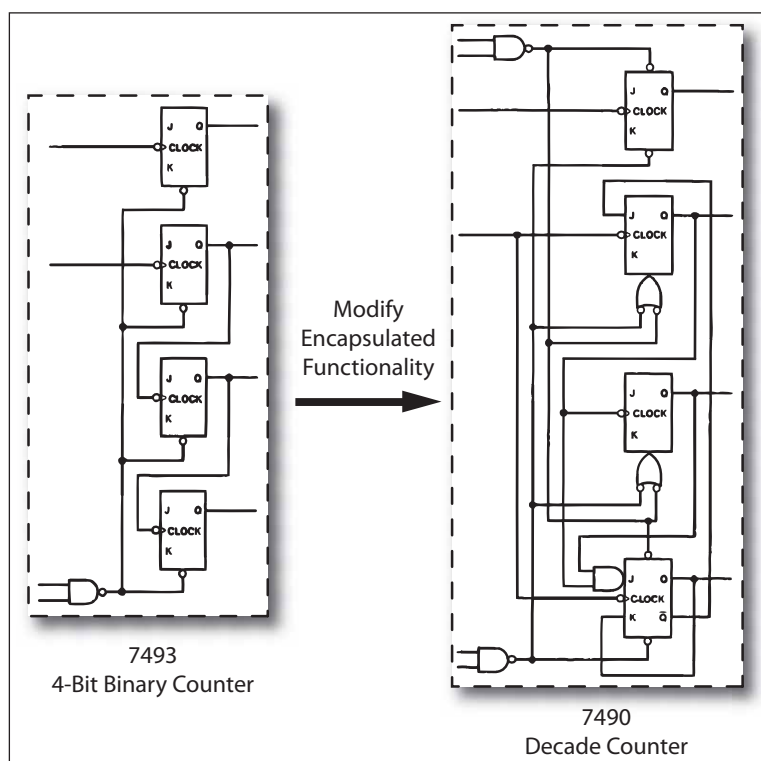


FIGURE C.2: Modifying the Encapsulated Functionality of a Hardware Component

but is encapsulated behind a clearly defined interface of several pins. Figure C.3 shows both the internal and external views of the 7490 IC. The functionality exposed through the pins of the IC are designed so that the component can be integrated without requiring an understanding of the internal workings of the IC itself. This allows the external view of the IC to be focused solely on describing how the chip is controlled by creating connections to the input and output of the IC. The configuration of these connections to the pins of the IC determines the behavior exhibited by the component. For example in Figure C.4, if the  $Q_A$  pin is connected to the  $INPUT_B$  pin it causes the 7490 IC to be a decade counter in binary coded decimal (BCD). Other possible configurations of the 7490 IC can set it to count in bi-quinary code decimal or to operate as a divide by ten counter.

Figure C.4 shows the integration of the 7490 IC with two other components to create the units of the seconds module of the digital clock. The two components are a 7447 IC and a 7-segment display. The 7447 chip is a Binary Coded Decimal (BCD) to 7-segment Decoder/Driver. A 7-segment display is a special panel of Light Emitting Diodes (LED's)

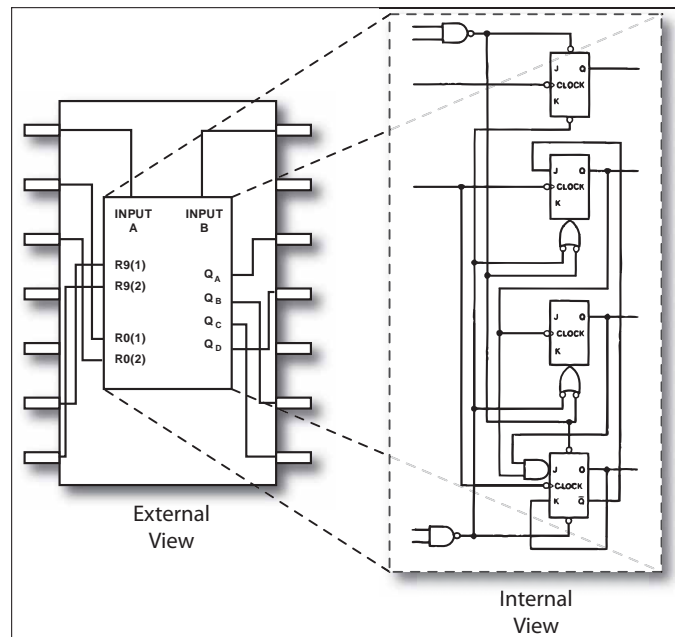


FIGURE C.3: Internal and External Views of the 7490 IC

used to display the numbers 0-9.

Even this small part of the design highlights the clear separation of computation and control used by the hardware component model. Computation is encapsulated inside the three hardware components and is only accessible through a clearly defined interface of pins external to the IC. Control of the components is determined by how the pins of the

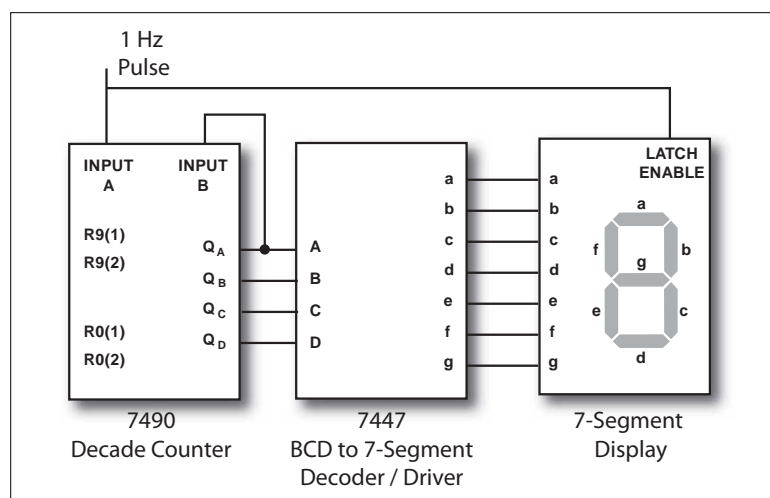


FIGURE C.4: Units of the Seconds Section



hardware components are wired together. The wiring that connects the pins determines the functionality that each component exhibits individually and also how this functionality is integrated with other components.

This part of the design also has another interesting point. The 7490 chip and the 7-segment display are incompatible, and cannot be directly wired together. In the hardware model, rather than modifying both components to be compatible, a third hardware component (the 7447 IC) is introduced to interface between the two components. This concept of a component operating as an interface increases reuse as components do not need to be modified to be compatible, and the interface component may be reused with other similar combinations of incompatible components.

The seconds tens portion of the digital clock requires a counter from 0-5. This counter can be made by reusing the seconds units and modifying the 7490 chip to count from 0-5 by triggering a reset to zero on the binary value of six (0110b). This modification is achieved entirely by changing the wiring to the IC, as shown in Figure C.5. The wiring modifies the configuration of the component, so that when the counter reaches six, it is reset to zero using the  $R_0$  pins. The seconds module is completed by integrating the seconds tens counter with the seconds units by connecting the ten's clock pulse to the  $Q_D$  of the unit counter which pulses once every 10 seconds. The same module can be reused for the minutes module of the digital clock because it requires exactly the same functionality as the seconds module.

The hours portion differs from the minutes and the seconds in that it must count from 1-12. The hours units must count from (01-09) and then it must be able to be reset at three back to one in the second cycle (10-12). It is not possible to achieve this functionality with the 7490 IC used previously, as it is unable to reset the count to one. This requires a new component, the 74196 IC, which is a presetable decade counter. As noted previously, the 74196 IC is a new component made by modifying the encapsulated functionality of the existing 7490 IC. The 74196 chip can now be used to count from one using a set of inputs and a *LOAD* control pin.

The hours tens are required to count from 0-1, increasing when the units reach nine, and resetting to zero when the units reach three in the second cycle. As this does not require all the functionality of the 7490, a simpler component can be used. The 7476 chip is a JK

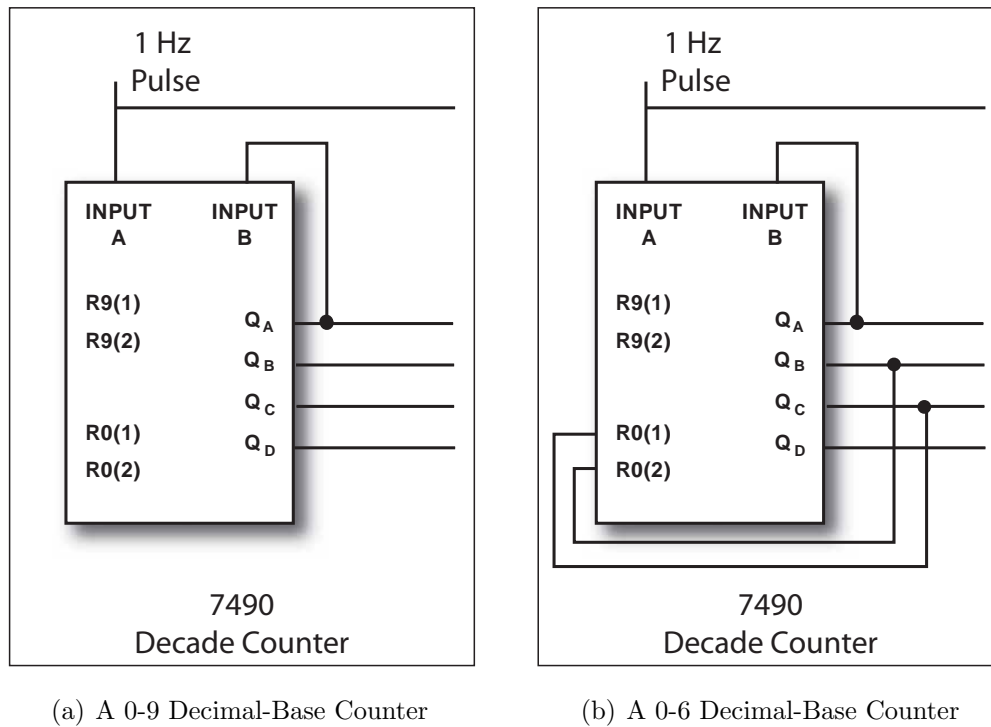


FIGURE C.5: Reconfiguring a component using integration

Flip-Flop with Preset and Clear. As the 7476 chip only has one output, it can be connected to the b and c inputs of the 7-segment display. This partial use of an interface will make the 7-segment display show a one when the count is one, and nothing when the count is zero.

Though the hours units and hours tens are now complete, it is not possible to integrate them directly to achieve the required functionality. In order to do this, more logic must be added which resets the count on 13 back to one. This is achieved with a 7410 chip, which contains a 3-input NAND gate. The completed hours module is shown in Figure C.6.

Figure C.7 shows the reuse of components in the simple digital clock case study. Reuse takes the form of either modification of an existing component to create a new component or reuse of an integrated group of components. New components can be created from existing components by reusing the existing encapsulated functionality and extending this functionality to a new task. This is shown with the 7490 IC which extends the encapsulated functionality of the 7493 IC and the 74196 IC which similarly extends the 7490 IC. An integrated group of components can also be reused by treating a number of components and their integration as a single entity. This entity can then be reused in the same

configuration, as with the seconds module being reused as the minutes module. Alternatively, the integration of the entity can be modified to create a new configuration as with the seconds tens created by modifying the integration of 7490 IC. This modification may include integration with additional components. This is shown with the hours module which reuses the seconds module with changes to components and addition of a 7410 component.

The case study also allows the characteristics of the hardware component model to be observed. Firstly, the functionality of a hardware component is fully encapsulated and only accessible through a clearly defined interface of pins external to the component. This allows the separation of the computation (encapsulated functionality) from the integration of the components by wiring together the pins of components. The integration of components in this manner promotes loose coupling between components. It can also be used to give components multiple configurations, with the active configuration chosen by integration.

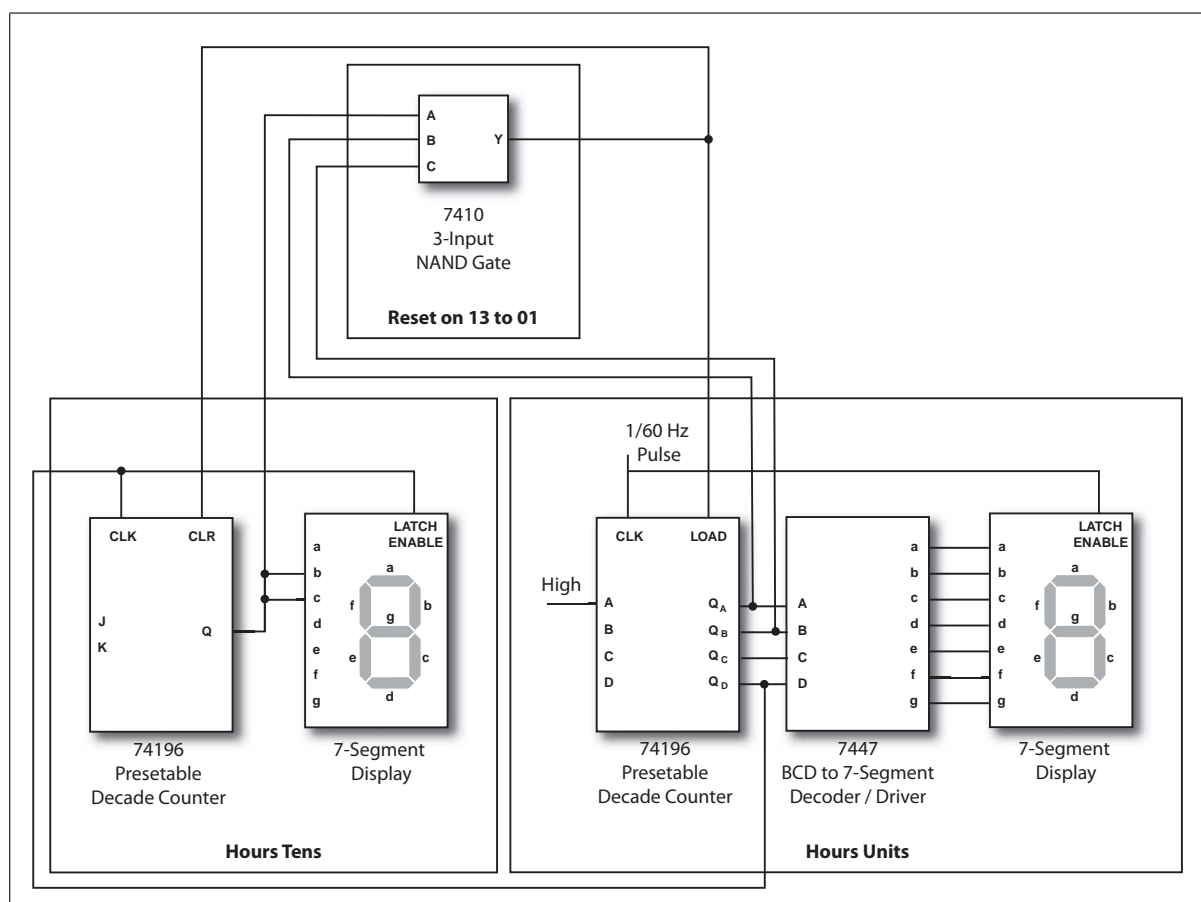


FIGURE C.6: The Hours Module of the Simple Digital Clock

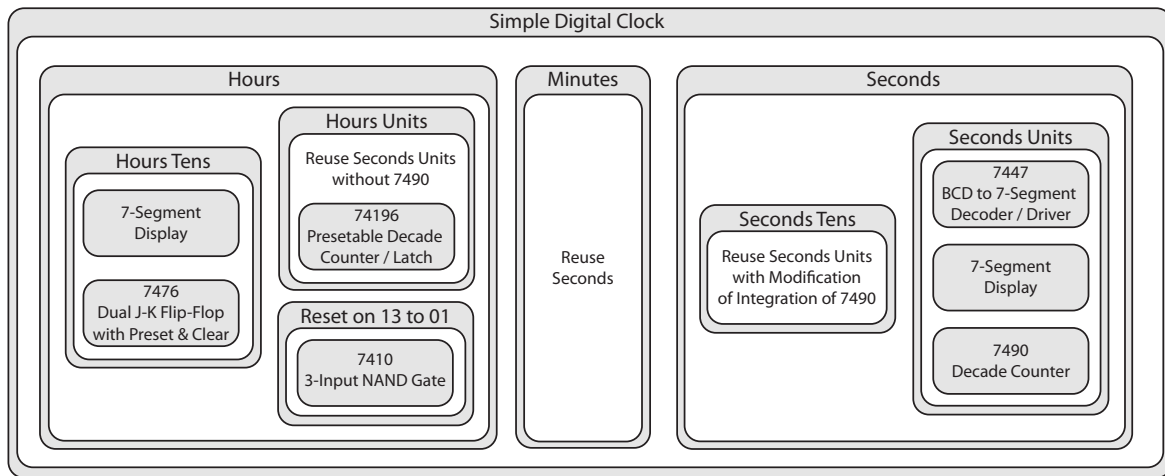


FIGURE C.7: Reuse in Digital Clock Case Study

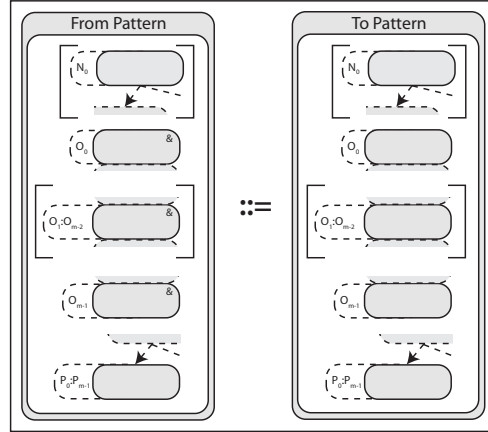
This was shown with the 7490 IC which was used to count to nine in one configuration and was reused to count to six with a different configuration of the same component. Finally, the approach to integration in the hardware component model encourages the use of components as interfaces to connect two incompatible components. This was shown with the 7447 IC which converts from the BCD format to a format suitable for a 7-segment display.

# D

## Behavior Tree Model-to-Model Transformations

This appendix contains the model-to-model (M2M) transformations used for the embedded Behavior Run-time Environment (eBRE) and standard Behavior Run-time Environment (BRE). The M2M transformations are described in the BT M2M transformation language discussed in Section 5.2. The conjunction, disjunction, exclusive or, reversion, reference, branch-kill and synchronisation transformation rules follow.

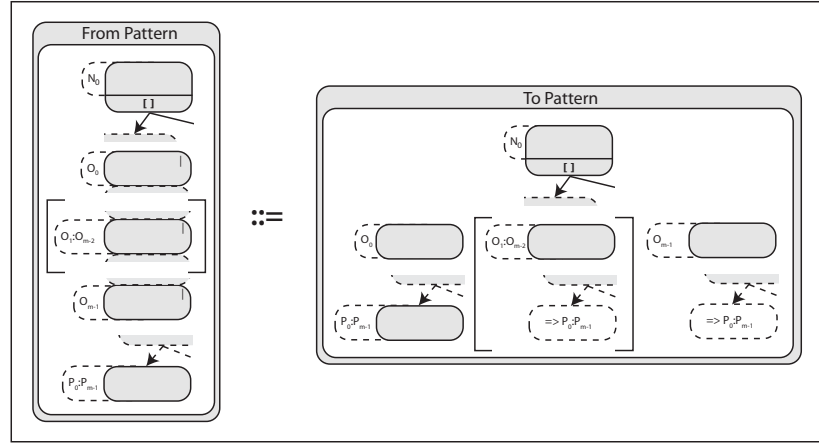
## D.1 Conjunction Transformation Rule



**From Pattern:**  $\{O_0:O_{m-1}\}.op = \&$   
**To Pattern:**  $\{O_0:O_{m-1}\}.op = ''$

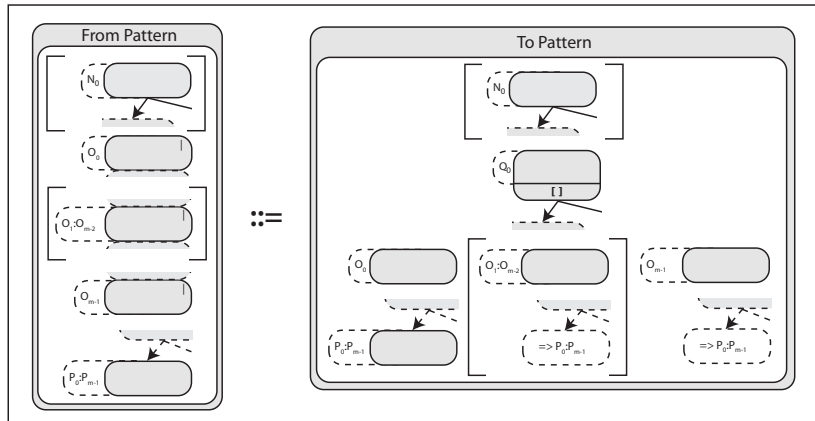
FIGURE D.1: Conjunction Transformation Rule

## D.2 Disjunction Transformation Rule



**From Pattern:**  $\{O_0:O_{m-1}\}.op = '|'$   
**To Pattern:**  $\{O_0:O_{m-1}\}.op = ''$

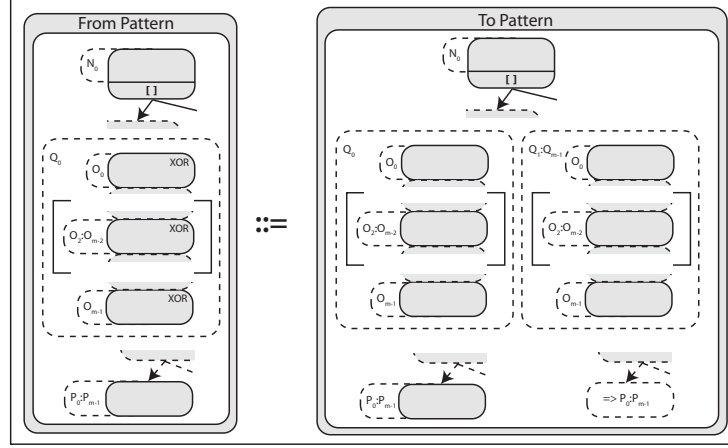
FIGURE D.2: Disjunction Transformation Rule (Alternate Branching)



**From Pattern:**  $\{O_0:O_{m-1}\}.op = '|'$   
**To Pattern:**  $\{O_0:O_{m-1}\}.op = ''$

FIGURE D.3: Disjunction Transformation Rule (Parallel Branching)

### D.3 Exclusive OR Transformation Rule



**From Pattern:**

$$Q_0.\{O_0:O_{m-1}\}.op = 'XOR'$$

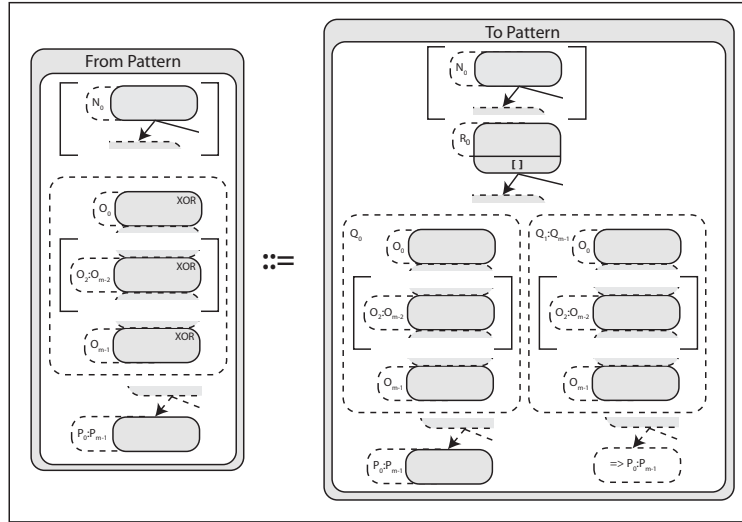
**To Pattern:**

$$Q.m = O.m$$

$$Q_0.\{O_0:O_{m-1}\}.op = ''$$

$$\forall i \in 0:Q.m, \forall j \in 0:O.m, \text{ if } i \neq j \text{ then } Q_i.O_j.b = 'NOT (' + Q_i.O_j.b + ')'$$

FIGURE D.4: Exclusive OR Transformation Rule (Alternate Branching)



**From Pattern:**

$$Q_0.\{O_0:O_{m-1}\}.op = 'XOR'$$

**To Pattern:**

$$Q.m = O.m$$

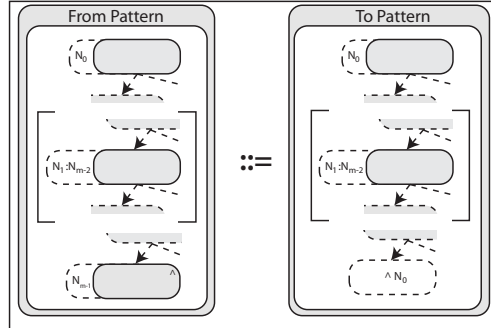
$$Q_0.\{O_0:O_{m-1}\}.op = ''$$

$$\forall i \in 0:Q.m, \forall j \in 0:O.m, \text{ if } i \neq j \text{ then } Q_i.O_j.b = 'NOT (' + Q_i.O_j.b + ')'$$

FIGURE D.5: Exclusive OR Transformation Rule (Parallel Branching)



## D.4 Reversion Transformation Rule

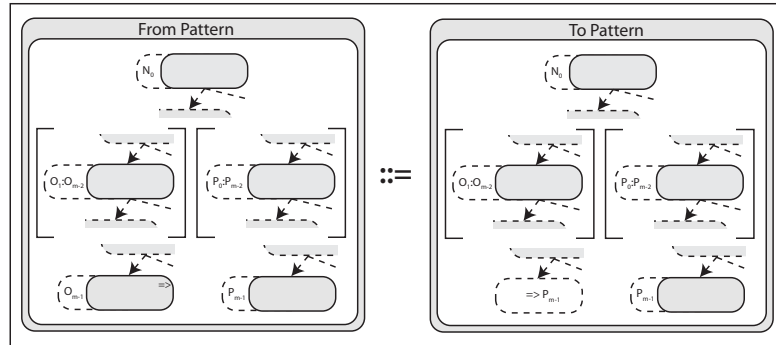


**From Pattern:**  $N_{m-1}.op = '\wedge'$   
 $N_0.C = N_{m-1}.C$   
 $N_0.b = N_{m-1}.b$   
 $N_0.bt = N_{m-1}.bt$   
 $N_0.l = N_{m-1}.l$

**To Pattern:**

FIGURE D.6: Reversion Transformation Rule

## D.5 Reference Transformation Rule

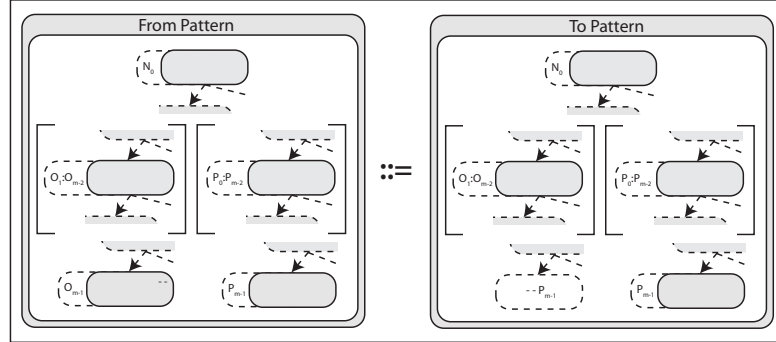


**From Pattern:**  $O_{m-1}.op = '=>'$   
 $O_{m-1}.C = P_{m-1}.C$   
 $O_{m-1}.b = P_{m-1}.b$   
 $O_{m-1}.bt = P_{m-1}.bt$   
 $O_{m-1}.l = P_{m-1}.l$

**To Pattern:**

FIGURE D.7: Reference Transformation Rule

## D.6 Branch-Kill Transformation Rule

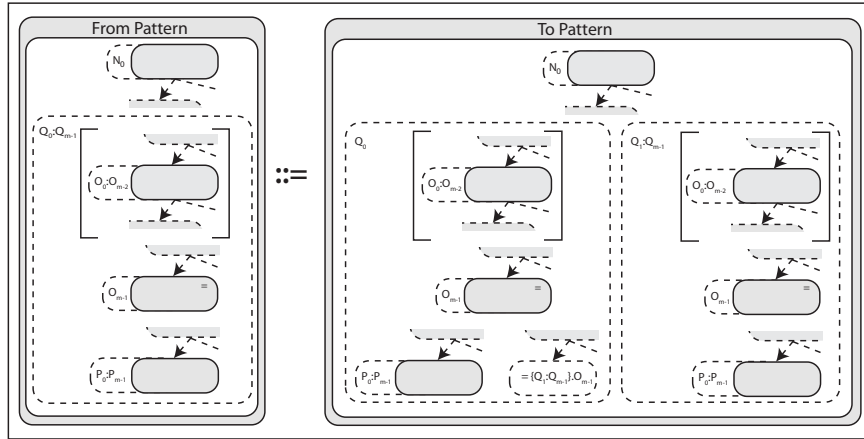


**From Pattern:**  $O_{m-1}.op = ' '$   
 $O_{m-1}.C = P_{m-1}.C$   
 $O_{m-1}.b = P_{m-1}.b$   
 $O_{m-1}.bt = P_{m-1}.bt$   
 $O_{m-1}.l = P_{m-1}.l$

**To Pattern:**

FIGURE D.8: Branch-Kill Transformation Rule

## D.7 Synchronisation Transformation Rule



**From Pattern:**  $\{Q_0 : Q_{m-1}\}.O_{m-1}.op = ' '$   
 $Q_0.O_{m-1}.C = \{Q_1 : Q_{m-1}\}.O_{m-1}.C$   
 $Q_0.O_{m-1}.b = \{Q_1 : Q_{m-1}\}.O_{m-1}.b$   
 $Q_0.O_{m-1}.bt = \{Q_1 : Q_{m-1}\}.O_{m-1}.bt$   
 $Q_0.O_{m-1}.l = \{Q_1 : Q_{m-1}\}.O_{m-1}.l$

**To Pattern:**

FIGURE D.9: Synchronisation Transformation Rule

# References

- [Ale64] Alexander, C. W. (1964), *Notes on the Synthesis of Form (Harvard Paperbacks)*, Harvard University Press.
- [Amb07] Ambler, S. (2007), ‘Agile Model Driven Development (AMDD): The Key to Scaling Agile Software Development’, Viewed 28 December 2009, <http://www.agilemodeling.com/essays/amdd.htm>.
- [Arb98] Arbab, F. (1998), ‘What Do You Mean, Coordination?’, in *Bulletin of the Dutch Association for Theoretical Computer Science (NVTI)*, pp. 11–22.
- [Arb04] Arbab, F. (2004), ‘Reo: a channel-based coordination model for component composition’, *Mathematical Structures in Computer Science*, vol. 14(3), pp. 329–366.
- [Arm06] Armstrong, D. J. (2006), ‘The Quarks of Object-Oriented Development’, *Communications of the ACM*, vol. 49(2), pp. 123–128.
- [AS04] Angelov, C. and Sierszecki, K. (2004), ‘A Software Framework for Component-Based Embedded Applications’, in *Proceedings of the 11th Asia-Pacific Software Engineering Conference (APSEC)*, pp. 655–662.
- [BBI<sup>+</sup>04] Booch, G., Brown, A., Iyengar, S., Rumbaugh, J., and Selic, B. (2004), *MDA Journal: Straight from the Masters*, chap. An MDA Manifesto, pp. 133–143, Meghan-Kiffer Press.

- [BBJ07] Bézivin, J., Barbero, M., and Jouault, F. (2007), ‘On the Applicability Scope of Model Driven Engineering’, in *Proceedings of the Fourth International Workshop on Model-Based Methodologies for Pervasive and Embedded Software (MOMPES '07)*, pp. 3–7, IEEE Computer Society, Washington, DC, USA.
- [BCvH<sup>+</sup>03] Billington, J., Christensen, S., van Hee, K., Kindler, E., Kummer, O., Petrucci, L., Post, R., Stehno, C., and Weber, M. (2003), ‘The Petri Net Markup Language: Concepts, Technology, and Tools’, in *Proceedings of the 24th International Conference on Applications and Theory of Petri Nets (ICATPN 2003)*, pp. 1023–1024, Eindhoven, The Netherlands.
- [Beh09] Behavior Engineering (2009), ‘Behavior Engineering Website’, Viewed 28 December 2009, [www.behaviorengineering.org](http://www.behaviorengineering.org).
- [Bel04] Bell, E., Alex (2004), ‘Death by UML Fever’, *Queue*, pp. 72–80.
- [Ben97] Bennett, D. W. (1997), *Designing Hard Software: The Essential Tasks*, Prentice Hall.
- [Bet04] Bettin, J. (2004), *MDA Journal: Straight from the Masters*, chap. Model-Driven Software Development, Meghan-Kiffer Press.
- [Béz05] Bézivin, J. (2005), ‘On the Unification Power of Models’, *Software and System Modeling*, vol. 4(2), pp. 171–188.
- [Béz06] Bézivin, J. (2006), ‘Introduction to Model Engineering: A gentle introduction to a new way of considering the construction and maintenance of information systems’, Viewed 28 December 2009, <http://www.eclipse.org/gmt/omcw/resources/chapter02/downloads/IntroductionToModelEngineering.INRIA.ppt>.
- [BJGB07] Barbero, M., Jouault, F., Gray, J., and Bézivin, J. (2007), ‘A Practical Approach to Model Extension’, *Model Driven Architecture- Foundations and Applications*, pp. 32–42.

- [BLWG99] Bisbal, J., Lawless, D., Wu, B., and Grimson, J. (1999), ‘Legacy Information Systems: Issues and Directions’, *IEEE Software*, vol. 16(5), pp. 103–111.
- [Bos08] Boston, J. (2008), ‘Behavior Trees - How they improve Engineering Behaviour?’, in *6th Annual Software and Systems Engineering Process Group Conference (SEPG 2008)*, (Raytheon Australia).
- [Bro86] Brooks, R. A. (1986), ‘A Robust Layered Control System for a Mobile Robot’, *IEEE Journal of Robotics and Automation*, vol. RA-2(1), pp. 14–23.
- [Bro87] Brooks, F. P. J. (1987), ‘No Silver Bullet: Essence and Accidents of Software Engineering’, *Computer*, vol. 20(4), pp. 10–19.
- [Bro90] Brooks, R. A. (1990), ‘Elephants Don’t Play Chess’, *Robotics and Autonomous Systems*, vol. 6(1&2), pp. 3–15.
- [CD08] Crane, M. L. and Dingel, J. (2008), ‘Towards a Formal Account of a Foundational Subset for Executable UML Models’, in *Proceedings of the 11th International Conference on Model Driven Engineering Languages and Systems (MoDELS ’08)*, pp. 675–689, Springer-Verlag, Berlin, Heidelberg.
- [CGW08] Colvin, R., Grunske, L., and Winter, K. (2008), ‘Timed Behavior Trees for Failure Mode and Effects Analysis of time-critical systems’, *Journal of Systems and Software*, vol. 81(12), pp. 2163–2182.
- [CH07] Colvin, R. and Hayes, I. (2007), ‘A Semantics for Behavior Trees’, ACCS Technical Report ACCS-TR-07-01, ARC Centre for Complex Systems.
- [CH09] Colvin, R. and Hayes, I. J. (2009), ‘CSP with Hierarchical State’, in *Proceedings of the 7th International Conference on Integrated Formal Methods (IFM 2009)*, pp. 118–135, Düsseldorf, Germany.
- [Cha05] Charette, R. N. (2005), ‘Why Software Fails’, *IEEE Spectrum*, vol. 42(9), pp. 42–49.

- [Coo04a] Cook, S. (2004), ‘Eventful times at OOPSLA’, Viewed 28 December 2009, <http://blogs.msdn.com/stevecook/archive/2004/10/27/248322.aspx>.
- [Coo04b] Cook, S. (2004), *MDA Journal: Straight from the Masters*, chap. Microsoft’s Approach to Modelling is Customer Driven, pp. 104–106, Meghan-Kiffer Press.
- [Coo08] Cook, S. (2008), ‘Rejoining the OMG: UML and beyond’, Viewed 28 December 2009, <http://blogs.msdn.com/stevecook/archive/2008/09/29/rejoining-the-omg-uml-and-beyond.aspx>.
- [Cox86] Cox, B. J. (1986), *Object-Oriented Programming: An Evolutionary Approach*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.
- [CR06] Clayberg, E. and Rubel, D. (2006), *Eclipse: Building Commercial-Quality Plugins*, Addison-Wesley Professional, 2nd ed.
- [CSW08] Clark, T., Sammut, P., and Willans, J. (2008), *Applied Metamodeling: A Foundation for Language Driven Development*, Viewed 28 December 2009, <http://itcentre.tvu.ac.uk/~clark/docs/Applied%20Metamodelling%20%28Second%20Edition%29.pdf>, 2nd ed.
- [Dav02] Davis, J. (2002), ‘Model Integrated Computing: A Framework for Creating Domain Specific Design Environments’, in *Proceedings of the 6th World Multiconference on Systems, Cybernetics, and Informatics (SCI)*, Orlando, FL.
- [Dep87] Department of Defense (1987), ‘MIL-STD-1553B: Digital Time Division Command/Response Multiplex Data Bus’, Viewed 28 December 2009, <http://snebulos.mit.edu/projects/reference/MIL-STD/MIL-STD-1553B.pdf>.
- [DHJ<sup>+</sup>01] Dahlström, A., Heintz, F., Jacobsson, M., Thapper, J., and Öberg, M. (2001), ‘The NOAI Team Description’, in *RoboCup 2000: Robot Soccer World Cup IV*, pp. 413–416, Springer-Verlag, London, UK.

- [Dmi04] Dmitriev, S. (2004), ‘Language Oriented Programming: The Next Programming Paradigm’, Viewed 28 December 2009, <http://www.onboard.jetbrains.com/is1/articles/04/10/lop/mps.pdf>.
- [dMOR<sup>+</sup>04] de Moura, L., Owre, S., Rueß, H., Rushby, J., Shankar, N., Sorea, M., and Tiwari, A. (2004), ‘SAL 2’, in R. Alur and D. Peled, eds., *Computer-Aided Verification (CAV 2004)*, vol. 3114 of *Lecture Notes in Computer Science*, pp. 496–500, Springer-Verlag, Boston, MA.
- [Dro01] Dromey, R. G. (2001), ‘Genetic Software Engineering - Simplifying Design Using Requirements Integration’, in *IEEE Working Conference on Complex and Dynamic Systems Architecture*, Brisbane, Australia.
- [Dro02] Dromey, R. G. (2002), ‘From Requirements to Design - Without Miracles’, Viewed 28 December 2009, <http://www.sqi.gu.edu.au/docs/sqi/gse/Dromey-ICSE-2003.pdf>.
- [Dro03] Dromey, R. G. (2003), ‘From Requirements to Design: Formalising the Key Steps’, in *IEEE International Conference on Software Engineering and Formal Methods (SEFM’03)*, pp. 2–11, Brisbane, Australia, (Invited Keynote Address).
- [Dro06a] Dromey, R. (2006), ‘Scaleable formalization of imperfect knowledge’, in *1st International Workshop - Asian Working Conference on Verified Software (AWCVS06)*, pp. 21–33.
- [Dro06b] Dromey, R. G. (2006), ‘Climbing over the “No Silver Bullet” Brick Wall’, *IEEE Software*, vol. 23(2), pp. 118–120.
- [Dro06c] Dromey, R. G. (2006), ‘Formalizing the Transition from Requirements to Design’, in J. He and Z. Liu, eds., *Mathematical Frameworks for Component Software - Models for Analysis and Synthesis*, World Scientific Series on Component-Based Development, pp. 156–187, World Scientific Publishing Co., Inc., River Edge, NJ, USA, (Invited Chapter).

- [Ecl09] Eclipse Foundation (2009), ‘Eclipse Modeling Framework Project (EMF)’, Viewed 28 December 2009, <http://www.eclipse.org/modeling/emf/>.
- [Est09] Esterel Technologies (2009), ‘SCADE Suite - The Standard for the Development of Safety-Critical Embedded Software in Aerospace & Defense, Rail Transportation, Energy and Heavy Equipment Industries’, Viewed 28 December 2009, <http://www.esterel-technologies.com/products/scade-suite/>.
- [Fow05] Fowler, M. (2005), ‘Language Workbenches: The Killer-App for Domain Specific Languages?’, Viewed 28 December 2009, <http://www.martinfowler.com/articles/languageWorkbench.html>.
- [Fra02] Frankel, D. (2002), *Model Driven Architecture: Applying MDA to Enterprise Computing*, John Wiley & Sons, Inc., New York, NY, USA.
- [Fra04a] Frankel, D. S. (2004), *MDA Journal: Straight from the Masters*, chap. The MDA Marketing Message and the MDA Reality, pp. 112–118, Meghan-Kiffer Press.
- [Fra04b] Frankel, D. S. (2004), *MDA Journal: Straight from the Masters*, chap. MDA and the Object Technology Barrier, Meghan-Kiffer Press.
- [Fre09] Free Model Foundry (2009), ‘FMF Standard (7400 series) with Bushold Models Directory’, Viewed 28 December 2009, <http://freemodelfoundry.com/stdnh.php>.
- [Fri04] Fritzson, P. (2004), *Principles of Object-Oriented Modeling and Simulation with Modelica 2.1*, Wiley-IEEE Press.
- [FS06] France, R. B. and Solberg, A. (2006), ‘Model-Driven Development Using UML 2.0: Promises and Pitfalls’, *Computer*, vol. 39(2), pp. 59–66.
- [Fug93] Fuggetta, A. (1993), ‘A Classification of CASE Technology’, *Computer*, vol. 26(12), pp. 25–38.



- [GCW07] Grunske, L., Colvin, R., and Winter, K. (2007), ‘Probabilistic Model-Checking Support for FMEA’, in *Proceedings of the 4th International Conference on Quantitative Evaluation of Systems (QEST’07)*, pp. 119–128.
- [Git05] Gitzel, R. (2005), *Model-Driven Software Development Using a Metamodel-Based Extension Mechanism for UML*, Ph.D. thesis, University of Mannheim.
- [Giu07] Giudice, D. L. (2007), ‘The State of Model-Driven Development’, Tech. rep., Forrester.
- [GLS99] Grandpierre, T., Lavarenne, C., and Sorel, Y. (1999), ‘Optimized Rapid Prototyping For Real-Time Embedded Heterogeneous multiprocessors’, in *Proceedings of 7th International Workshop on Hardware/Software Co-Design (CODES’99)*, Rome, Italy.
- [GLYW05] Grunske, L., Lindsay, P. A., Yatapanage, N., and Winter, K. (2005), ‘An Automated Failure Mode and Effect Analysis Based on High-Level Design Specification with Behavior Trees’, in J. Romijn, G. Smith, and J. van de Pol, eds., *Proceedings of the 5th International Conference on Integrated Formal Methods (IFM 2005)*, vol. 3771 of *Lecture Notes in Computer Science*, pp. 129–149, Springer, Eindhoven, The Netherlands.
- [GPHSD05] Gonzalez-Perez, C., Henderson-Sellers, B., and Dromey, R. G. (2005), ‘A Metamodel for the Behavior Trees Modelling Technique’, in *Proceedings of the Third International Conference on Information Technology and Applications (ICITA’05)*, vol. 2, pp. 35–39, IEEE Computer Society, Washington, DC, USA.
- [GSC<sup>+</sup>04] Greenfield, J., Short, K., Cook, S., Kent, S., and Crupi, J. (2004), *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools*, John Wiley & Sons.
- [Gut04a] Guttman, M. (2004), *MDA Journal: Straight from the Masters*, chap. MDA and Microsoft, pp. 107–111, Meghan-Kiffer Press.

- [Gut04b] Guttman, M. (2004), *MDA Journal: Straight from the Masters*, chap. Microsoft Should Not Compete with MDA, pp. 95–103, Meghan-Kiffer Press.
- [GvN47] Goldstine, H. H. and von Neumann, J. (1947), *Planning and Coding of Problems for an Electronic Computing Instrument*, Institute for Advanced Study, Princeton, NJ.
- [GWY08] Grunske, L., Winter, K., and Yatapanage, N. (2008), ‘Defining the abstract syntax of visual languages with advanced graph grammars - A case study based on behavior trees’, *Journal of Visual Languages and Computing*, vol. 19(3), pp. 343–379.
- [Haa02] Haas, P. (2002), *Stochastic Petri Nets: Modelling, Stability, Simulation*, Springer-Verlag, New York.
- [Har92] Harel, D. (1992), ‘Biting the Silver Bullet: Toward a Brighter Future for System Development’, *Computer*, vol. 25(1), pp. 8–20.
- [Hec09] Hecker, C. (2009), ‘My Liner Notes for Spore’, Viewed 28 December 2009, [http://chrishecker.com/My\\_Liner\\_Notes\\_for\\_Spore](http://chrishecker.com/My_Liner_Notes_for_Spore).
- [HJD05] Hull, E., Jackson, K., and Dick, J. (2005), *Requirements Engineering*, Springer, 2nd ed.
- [HKPDT06] Hillah, L., Kordon, F., Petrucci-Dauchy, L., and Trèves, N. (2006), ‘PN Standardisation: A Survey’, in E. Najm, J.-F. Pradat-Peyre, and V. Donzeau-Gouge, eds., *Formal Techniques for Networked and Distributed Systems (FORTE’06)*, vol. 4229 of *Lecture Notes in Computer Science*, pp. 307–322, Springer, Paris, France.
- [HM07] Heering, J. and Mernik, M. (2007), ‘Domain-specific languages in perspective’, Tech. rep., CWI, sEN-E0702.
- [HT06] Hailpern, B. and Tarr, P. (2006), ‘Model-driven development: The good, the bad, and the ugly’, *IBM Systems Journal*, vol. 45(3), pp. 451–461.

- [IAR09] IAR Systems (2009), ‘IAR visualSTATE’, Viewed 28 December 2009, <http://www.iar.com/website1/1.0.1.0/371/1/>.
- [IBM09] IBM (2009), ‘Rational Statemate’, Viewed 28 December 2009, <http://www-01.ibm.com/software/awdtools/statemate/>.
- [IEE04] IEEE Computer Society (2004), ‘1076.6 - IEEE Standard for VHDL Register Transfer Level (RTL) Synthesis’, Viewed 28 December 2009, <http://ieeexplore.ieee.org/servlet/opac?punumber=9308>.
- [Ins08] Institute for Software Integrated Systems (2008), ‘GME: Generic Modeling Environment’, Viewed 28 December 2009, <http://www.isis.vanderbilt.edu/projects/gme/>.
- [Int09a] Intel (2009), ‘60 Years of the Transistor: 1947–2007’, Viewed 28 December 2009, <http://www.intel.com/technology/timeline.pdf>.
- [Int09b] IntelliWizard (2009), ‘UML StateWizard’, Viewed 28 December 2009, <http://www.intelliwizard.com/>.
- [Int09c] Intentional Software (2009), ‘Public Intentional Demo’, Viewed 28 December 2009, <http://blog.intentsoft.com/>.
- [Ire08] Ireland, A. (2008), ‘High Integrity Computing (F23PS3): An Exercise in High Integrity Software Development’, Viewed 28 December 2009, <http://www.macs.hw.ac.uk/~air/hic-hisd/assignments/spark-atp-assign-bsc.pdf>.
- [Isl05] Isla, D. (2005), ‘Managing Complexity in the Halo 2 AI System’, in *Proceedings of the Game Developers Conference (GDC’05)*, San Francisco, CA.
- [Jan07] Jantzen, J. (2007), *Foundations of Fuzzy Control*, John Wiley & Sons.
- [Jet09] JetBrains (2009), ‘Meta Programming System’, Viewed 28 December 2009, <http://www.jetbrains.com/mps/index.html>.

- [JKW07] Jensen, K., Kristensen, L., and Wells, L. (2007), ‘Coloured Petri Nets and CPN Tools for modelling and validation of concurrent systems’, *International Journal on Software Tools for Technology Transfer (STTT)*, vol. 9(3), pp. 213–254.
- [Jow53] Jowett, M. A. (1953), *The Dialogues of Plato: Translated into English with Analyses and Introductions*, Oxford University Press, Ely House, London, 4th ed.
- [Kra03] Krasner, J. (2003), ‘Embedded Software Development Issues and Challenges: Failure Is NOT An Option - It Comes Bundled With The Software’, Viewed 28 December 2009, [http://embeddedforecast.com/emf\\_esdi&c.pdf](http://embeddedforecast.com/emf_esdi&c.pdf).
- [KWB03] Kleppe, A., Warmer, J., and Bast, W. (2003), *MDA Explained: The Model Driven Architecture: Practice and Promise*, Addison-Wesley Professional.
- [Lan74] Lancaster, D. E. (1974), *TTL Cookbook*, Sams, Indianapolis, IN, USA.
- [Lan05] Lano, K. (2005), *Advanced Systems Design with Java, UML and MDA*, Butterworth-Heinemann, Newton, MA, USA.
- [LCM06] Lange, C. F., Chaudron, M. R., and Muskens, J. (2006), ‘In Practice: UML Software Architecture and Design Description’, *IEEE Software*, vol. 23(2), pp. 40–46.
- [LEW05] Lau, K.-K., Elizondo, P. V., and Wang, Z. (2005), ‘Exogenous Connectors for Software Components’, in G. T. Heineman, I. Crnkovic, H. W. Schmidt, J. A. Stafford, C. A. Szyperski, and K. C. Wallnau, eds., *Proceedings of the 8th International Symposium on Component-Based Software Engineering (CBSE 2005)*, vol. 3489 of *Lecture Notes in Computer Science*, pp. 90–106, Springer Verlag, St. Louis, MO, USA.
- [LLUE07] Lau, K.-K., Ling, L., Ukis, V., and Elizondo, P. V. (2007), ‘Composite Connectors for Composing Software Components’, in M. Lumpe and W. Vanderperren, eds., *6th International Symposium on Software Composition*

- (SC 2007), vol. 4829 of *Lecture Notes in Computer Science*, pp. 266–280, Springer, Braga, Portugal.
- [LLW06] Lau, K.-K., Ling, L., and Wang, Z. (2006), ‘Composing Components in Design Phase using Exogenous Connectors’, in *Proceedings of the 32nd EUROMICRO Conference on Software Engineering and Advanced Applications (EUROMICRO-SEAA 2006)*, pp. 12–19, IEEE, Cavtat/Dubrovnik, Croatia.
- [LW07] Lau, K.-K. and Wang, Z. (2007), ‘Software Component Models’, *IEEE Transactions on Software Engineering*, vol. 33(10), pp. 709–724.
- [Mat09] Mathworks (2009), ‘Simulink - Simulation and Model-Based Design’, Viewed 28 December 2009, <http://www.mathworks.com/products/simulink/>.
- [MB02] Mellor, S. J. and Balcer, M. (2002), *Executable UML: A Foundation for Model-Driven Architectures*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.
- [Mci68] Mcilroy, M. D. (1968), ‘Mass produced software components’, in P. Naur and B. Randell, eds., *Proceedings of the NATO Conference on Software Engineering*, pp. 138–155, NATO Science Committee, Garmish, Germany.
- [MD02] Milosevic, Z. and Dromey, R. G. (2002), ‘On Expressing and Monitoring Behaviour in Contracts’, in *Proceedings of the Sixth International Conference on Enterprise Distributed Object Computing (EDOC’02)*, IEEE Computer Society, Los Alamitos, CA, USA.
- [Mel04] Mellor, S. J. (2004), *MDA Journal: Straight from the Masters*, chap. Agile MDA, pp. 145–159, Meghan-Kiffer Press.
- [Mel06] Mellor, S. J. (2006), ‘Demystifying UML’, *Embedded Systems Design*, vol. 19(3).
- [Mel07a] Mellor, S. J. (2007), ‘Embedded Systems in UML’, Viewed 28 December 2009, [http://www.omg.org/news/whitepapers/050307\\_Embedded\\_Systems\\_in\\_UML\\_by\\_S\\_Mellor.pdf](http://www.omg.org/news/whitepapers/050307_Embedded_Systems_in_UML_by_S_Mellor.pdf).

- [Mel07b] Mellor, S. J. (2007), ‘We Can Generate Systems Today’, Viewed 28 December 2009, [http://www.omg.org/news/whitepapers/050307\\_Embedded\\_Systems\\_in\\_UML\\_by\\_S\\_Mellor.pdf](http://www.omg.org/news/whitepapers/050307_Embedded_Systems_in_UML_by_S_Mellor.pdf).
- [Men09] Mentor Graphics (2009), ‘BridgePoint’, Viewed 28 December 2009, [http://www.mentor.com/products/sm/model\\_development/bridgepoint/](http://www.mentor.com/products/sm/model_development/bridgepoint/).
- [Mey97] Meyer, B. (1997), ‘UML: The Positive Spin’, *American Programmer*, vol. 10(3).
- [MFD08] Myers, T., Fritzson, P., and Dromey, R. G. (2008), ‘Seamlessly Integrating Software & Hardware Modelling for Large-Scale Systems’, in *Proceedings of the 2nd International Workshop on Equation-Based Object-Oriented Languages and Tools (EOOLT’08)*, pp. 5–15, Paphos, Cyprus.
- [MFD09] Myers, T., Fritzson, P., and Dromey, R. G. (2009), ‘Co-Modeling: From Requirements to an Integrated Software/Hardware Model’, *Computer (submitted)*.
- [MHS05] Mernik, M., Heering, J., and Sloane, A. M. (2005), ‘When and how to develop domain-specific languages’, *ACM Computing Surveys (CSUR)*, vol. 37(4), pp. 316–344.
- [MKUW04] Mellor, S. J., Kendall, S., Uhl, A., and Weise, D. (2004), *MDA Distilled*, Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA.
- [Mod09] Modelica Association (2009), ‘Modelica Standard Library’, Viewed 28 December 2009, <http://www.modelica.org/libraries/Modelica>.
- [Moo65] Moore, G. E. (1965), ‘Cramming more components onto integrated circuits’, *Electronics*, vol. 38(8).
- [Nas08] Nass, R. (2008), ‘An insider’s view of the 2008 Embedded Market Study’, Viewed 28 December 2009, <http://www.embedded.com/design/210200580>.

- [Nat09] National Instruments (2009), ‘NI LabVIEW 2009 for Designing Embedded Systems’, Viewed 28 December 2009, <http://www.ni.com/labview/whatsnew/embedded.htm>.
- [OÅD05] Otter, M., Årzén, K.-E., and Dressler, I. (2005), ‘StateGraph—A Modelica Library for Hierarchical State Machines’, in G. Schmitz, ed., *Proceedings of the 4th International Modelica Conference (Modelica 2005)*, pp. 569–578, Hamburg, Germany.
- [Obj03] Object Modeling Group (2003), ‘MDA Guide Version 1.0.1’, Viewed 15 August 2010, <http://www.omg.org/cgi-bin/doc?omg/03-06-01.pdf>.
- [Obj05] Object Modeling Group (2005), ‘Unified Modeling Language: Superstructure, Version 2.1.1’, Viewed 28 December 2009, <http://www.omg.org/cgi-bin/apps/doc?@formal/07-02-05.pdf>.
- [Obj06] Object Modeling Group (2006), ‘Object Constraint Language: OMG Available Specification, Version 2.0’, Viewed 28 December 2009, <http://www.omg.org/spec/OCL/2.0/>.
- [Obj07] Object Modeling Group (2007), ‘MOF 2.0/XMI Mapping, Version 2.1.1’, Viewed 28 December 2009, <http://www.omg.org/spec/XMI/2.1.1/>.
- [Obj08a] Object Modeling Group (2008), ‘Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification: Version 1.0’, Viewed 28 December 2009, <http://www.omg.org/spec/QVT/1.0/>.
- [Obj08b] Object Modeling Group (2008), ‘Semantics of a Foundational Subset for Executable UML Models (FUML), Version 1.0 - Beta 1’, Viewed 28 December 2009, <http://www.omg.org/spec/FUML/1.0/Beta1/>.
- [Obj09a] Object Modeling Group (2009), ‘Catalog Of UML Profile Specifications’, Viewed 28 December 2009, [http://www.omg.org/technology/documents/profile\\_catalog.htm](http://www.omg.org/technology/documents/profile_catalog.htm).

- [Obj09b] Object Modeling Group (2009), ‘Introduction To OMG’s Unified Modeling Language (UML)’, Viewed 28 December 2009, [http://www.omg.org/gettingstarted/what\\_is\\_uml.htm](http://www.omg.org/gettingstarted/what_is_uml.htm).
- [Obj09c] Object Modeling Group (2009), ‘Unified Modeling Language: Infrastructure, Version 2.2’, Viewed 28 December 2009, <http://www.omg.org/spec/UML/2.2/Infrastructure/PDF/>.
- [Obj09d] Object Modeling Group (2009), ‘Unified Modeling Language (UML), Version 2.2’, Viewed 28 December 2009, <http://www.omg.org/technology/documents/formal/uml.htm>.
- [Par00] Parr, S. M. (2000), *VICON - A Framework for Component-Based Design : Connecting the Pieces*, Master’s thesis, School of Computing and Information Technology. Griffith University.
- [Pet95] Petre, M. (1995), ‘Why looking isn’t always seeing: readership skills and graphical programming’, *Communications of the ACM*, vol. 38(6), pp. 33–44.
- [Pow07] Powell, D. (2007), ‘Requirements Evaluation Using Behavior Trees: Findings from Industry’, in *Australian Conference on Software Engineering (ASWEC’07)*, (Industry Paper).
- [Pre09] Precise UML Group (2009), ‘Main Details’, Viewed 28 December 2009, <http://www.cs.york.ac.uk/puml/maindetails.html>.
- [Rat05] Rational Software (2005), ‘Rational Unified Process: Best Practices for software development teams’, Tech. rep., IBM, tP026B, Rev 11/01.
- [RFW04] Raistrick, C., Francis, P., and Wright, J. (2004), *Model Driven Architecture with Executable UML(TM)*, Cambridge University Press, New York, NY, USA.
- [RM10] Rosenberg, D. and Mancarella, S. (2010), ‘Embedded Systems Development using SysML: An Illustrated Example using Enterprise Architect’, Viewed 18



- September 2010, [http://www.sparxsystems.com/downloads/ebooks/Embedded\\_Systems\\_Development\\_using\\_SysML.pdf](http://www.sparxsystems.com/downloads/ebooks/Embedded_Systems_Development_using_SysML.pdf).
- [Rot89] Rothenberg, J. (1989), *Artificial Intelligence, Simulation, and Modeling*, chap. The Nature of Modeling, pp. 75–92, John Wiley and Sons, Inc.
- [Sch06] Schmidt, D. C. (2006), ‘Guest Editor’s Introduction: Model-Driven Engineering’, *Computer*, vol. 39, pp. 25–31.
- [Sco04] Scott, K. (2004), *Fast Track UML 2.0*, APress.
- [Sel06] Selic, B. (2006), ‘UML 2: A model-driven development tool’, *IBM Systems Journal*, vol. 45(3), pp. 607–620.
- [Sel07] Selic, B. (2007), ‘A Systematic Approach to Domain-Specific Language Design Using UML’, in *Proceedings of the 10th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC’07)*, pp. 2–9, IEEE Computer Society, Washington, DC, USA.
- [SFP08] Suess, J. G., Fritzson, P., and Pop, A. (2008), ‘The Impreciseness of UML and Implications for ModelicaML’, in *Proceedings of the 2nd International Workshop on Equation-Based Object-Oriented Languages and Tools (EOOLT’08)*, pp. 17–27, Paphos, Cyprus.
- [Sie02] Siegel, J. (2002), ‘Making the case: OMG’s Model Driven Architecture’, *Software Development Times*, Viewed 28 December 2009, <http://www.sdtimes.com/link/26807>.
- [Sim79] Simon, H. A. (1979), *Models of Thought*, Yale University Press, London, England.
- [Sim88] Simon, H. A. (1988), *The Sciences of the Artificial*, The MIT Press, London, England, 2nd ed.
- [SM92] Shlaer, S. and Mellor, S. (1992), *Object Lifecycles*, Yourdon Press, New Jersey.

- [Sou09] Sourceforge (2009), ‘Home of SMC - The State Machine Compiler’, Viewed 28 December 2009, <http://smc.sourceforge.net/>.
- [Sow00] Sowa, J. F. (2000), *Knowledge representation: logical, philosophical and computational foundations*, Brooks/Cole Publishing Co., Pacific Grove, CA, USA.
- [Sta01] Starr, L. (2001), *Executable Uml: How to Build Class Models*, Prentice Hall PTR, Upper Saddle River, NJ, USA.
- [Sta06] StateWorks (2006), ‘Microwave Oven Control - Case Study’, Viewed 28 December 2009, [www.stateworks.com/active/download/MW0ven.pdf](http://www.stateworks.com/active/download/MW0ven.pdf).
- [Sto04] Storrie, H. (2004), ‘Semantics of Control-Flow in UML 2.0 Activities’, in *Proceedings of the 2004 IEEE Symposium on Visual Languages & Human Centric Computing (VLHCC '04)*, pp. 235–242, IEEE Computer Society, Washington, DC, USA.
- [SVC06] Stahl, T., Voelter, M., and Czarnecki, K. (2006), *Model-Driven Software Development: Technology, Engineering, Management*, John Wiley & Sons.
- [Tho03] Thomas, D. (2003), ‘UML - Unified or Universal Modeling Language?: UML2, OCL, MOF, EDOC - The Emperor Has Too Many Clothes’, *Journal of Object Technology*, vol. 2(1), pp. 7–12.
- [Tho04] Thomas, D. (2004), ‘MDA: Revenge of the Modelers or UML Utopia?’, *IEEE Software*, vol. 21(3), pp. 15–17.
- [Tri06] Triskel Project Team (2006), ‘Model-Driven Engineering for Component Based Software - 2006 Activity Report’, Tech. rep., INRIA.
- [Uni06] United Business Media (2006), ‘2006 Embedded Systems Design State of Embedded Market Survey’, Viewed 28 December 2009, [ftp://ftp.embedded.com/pub/ESD%20SubscribSurvey/2006%20ESD%20Market%20Study.pdf](http://ftp.embedded.com/pub/ESD%20SubscribSurvey/2006%20ESD%20Market%20Study.pdf).

- [Wag04] Wagner, T. (2004), ‘VFSM-ML 1.0 - Virtual Finite State Machine Mark-Up Language’, Viewed 28 December 2009, <http://www.stateworks.com/active/download/XML-VFSM-ML.pdf>.
- [Wal61] Walter, W. G. (1961), *The Living Brain*, Penguin Books Ltd., Harmondsworth, Middlesex.
- [Wan98] Wang, J. (1998), *Timed Petri Nets, Theory and Application*, Kluwer Academic Publishers.
- [War94] Ward, M. P. (1994), ‘Language-Oriented Programming’, *Software - Concepts and Tools*, vol. 15(4), pp. 147–161.
- [WCD09] Winter, K., Colvin, R., and Dromey, R. G. (2009), ‘Dynamic Relational Behaviour for Large-Scale Systems’, in *Proceedings of the 2009 Australian Software Engineering Conference (ASWEC’09)*, pp. 173–182, IEEE Computer Society, Los Alamitos, CA, USA.
- [WD04] Wen, L. and Dromey, R. (2004), ‘From Requirements Change to Design Change: A Formal Path’, in *IEEE International Conference on Software Engineering and Formal Methods (SEFM 2004)*, pp. 104–113.
- [WD06] Wen, L. and Dromey, R. (2006), ‘Architecture Normalization for Component-based Systems’, *Electronic Notes in Theoretical Computer Science*, vol. 160, pp. 335–348.
- [WDT<sup>+</sup>05] Willcock, C., Deiss, T., Tobies, S., Keil, S., Engler, F., and Schulz, S. (2005), *An Introduction to TTCN-3*, John Wiley & Sons, West Sussex, England.
- [Wik09] Wikipedia (2009), ‘Moore’s law: History’, Viewed 28 December 2009, [http://en.wikipedia.org/wiki/Moore's\\_law](http://en.wikipedia.org/wiki/Moore's_law).
- [Wol03] Wolf, W. (2003), ‘A Decade of Hardware/Software Codesign’, *Computer*, vol. 36(4), pp. 38–43.

- [WSWW06] Wagner, F., Schmuki, R., Wagner, T., and Wolstenholme, P. (2006), *Modeling Software with Finite State Machines: A Practical Approach*, Auerbach Publications.
- [WW03] Wagner, F. and Wolstenholme, P. (2003), ‘Modeling and Building Reliable, Re-Useable Software’, in *Proceedings of the 10th IEEE International Conference on Engineering of Computer-Based Systems (ECBS 2003)*, pp. 277–286, Huntsville, AL, USA.
- [WW04] Wagner, T. and Wolstenholme, P. (2004), ‘Misunderstandings about state machines’, Viewed 28 December 2009, <http://www.stateworks.com/active/download/wagf04-2-state-machine-misunderstandings.pdf>.
- [Yin09] Yin, R. K. (2009), *Case Study Research: Design and Methods*, Sage Publications.
- [Zac87] Zachmann, J. A. (1987), ‘A Framework for Information Systems Architecture’, *IBM Systems Journal*, vol. 26(3).
- [Zaf09] Zafar, S. (2009), *Integration of Access Control Requirements into System Specifications*, Ph.D. thesis, Griffith University.
- [ZD05a] Zafar, S. and Dromey, R. G. (2005), ‘Integrating Safety and Security Requirements into Design of an Embedded System’, in *Proceedings of the 12th Asia-Pacific Software Engineering Conference (APSEC’05)*, pp. 629–636, IEEE Computer Society Press, Taipei, Taiwan.
- [ZD05b] Zafar, S. and Dromey, R. G. (2005), ‘Managing Complexity in Modelling Embedded Systems’, in *Proceedings of Systems Engineering/Test and Evaluation Conference (SETE 05)*, Brisbane, Australia.
- [ZWCD06] Zafar, S., Winter, K., Colvin, R., and Dromey, R. G. (2006), ‘Verification of an Integrated Role-Based Access Control Model’, in *Proceedings of the 1st Asian Working Conference on Verified Software (AWCVS’06)*, pp. 21–33, Macao.

# Glossary

**algorithm architecture adequation** a methodology for implementing algorithms onto a hardware platform of networked microcontrollers and specialised hardware.

**behavior based** a software development approach that defines the behavior of a system by separating individual behaviors from their means of integration.

**behavior directed acyclical graphs** a diagram consisting of nodes wich describe behaviors and terminals which describe skills.

**behavior engineering** an integrated approach to systems development that supports the engineering of large-scale dependable software intensive systems and both the systems and software engineering level.

**behavior engineering expression syntax** a concise group of expressions that can be parsed and executed by the behavior run-time environment.

**behavior engineering extension mechanism** a mechanism used to define a mapping for a new expression in terms of the existing expression syntax.

**behavior modeling language** the representation used by behavior engineering that is composed of three integrated views where information is represented in a behavior tree, composition tree and a structure tree.

**behavior modeling process** the process used to create behavior engineering models. It is made of the four distinct stages of formalisation, a fitness-for-purpose test, specification

and design.

**behavior run-time environment** a virtual run-time environment for executing BE models.

**behavior tree** a formal, tree-like graphical form that represents the behavior of individual or networks of entities which realize and change states, create and break relations, make decisions, respond to and cause events, and interact by exchanging information and passing control.

**behavior tree process algebra** a low level language used to define the formal semantics of behavior trees.

**BT M2M language** a graphical language designed to document new transformation rules for the purposes of adding new expressions to the BT expression syntax.

**co-design** the mapping of functionality of an embedded system onto a hardware platform that is composed of a combination of software operating on general purpose microcontrollers and specialised hardware integrated circuits.

**co-modeling** the application of the principles of co-design at a higher level of abstraction and an earlier stage of system development.

**component behavior tree** a projection of each of the behavior tree nodes involving a single component whilst preserving the structure of the behavior tree.

**component interaction network** consists of each component represented only once, and interactions between two components are represented by a single line.

**component interface diagram** provides the preconditions and postconditions of a transition of the state of the component.

**composition tree** is one of the integrated views of the behavior modeling language. It has the form of a tree showing the hierarchy of components that the system annotated with the complete vocabulary of the system.

**computer aided software engineering** the development of applications to support and partially automate software development.

**domain specific language** a language that utilises the notation and abstractions common to a domain.

**eclipse** an open development platform which provides extensible frameworks, tools and runtimes for software development.

**executable & translatable UML** a subset of the UML with executable semantics and an action language that is used to create executable models.

**exogenous connectors** a behavior based component based software engineering approach that encapsulates and manages all control and data flows between components.

**finite state machine** a popular form of state transition diagrams that represents states as circles and transitions between the states as directed arrows.

**forward modeling** the creation of a system from a model.

**general purpose language** a language that uses generalised constructs to cut across multiple modeling dimensions.

**interactive modeling** a system and a model co-existing with each other, with any changes in one influencing the other.

**language exploitation** creation of a domain specific language by extending an existing general purpose language.

**language invention** creation of a domain specific language from scratch.

**language oriented programming** the development of domain-oriented and formally specified very high level languages which are suited to a particular programming problem.

**language workbench** implement the concepts of language oriented programming to simplify the design and integration of domain specific languages.

**metamodel** a model that describes the concepts of other models.

**model driven architecture** a prevalent view of model driven engineering.

**model driven engineering** a software engineering approach that advocates a shift from code-centric development to model-centric development, in which models are the primary artifacts of all stages of development.

**modelica** a mathematical modeling language.

**object constraint language** a formal textual language for describing constraints and queries on models that is compatible with the MOF.

**object orientation** a software development approach that uses objects to group related data and methods.

**petri net** a directed biparte graph consisting of places and transitions. Extensions include colored petri nets, hierarchical petri nets, timed petri nets and stochastic petri nets..

**place transition net** the simplest form of Petri nets that consist of places and transitions which are connected together by directed arcs.

**platform independent model** a model of the subject matter of concern described independently of any implementation details.

**platform specific model** a platform independent model with the addition of technical details making it suitable to be implemented on a specific platform.

**process control model** a five-state process control model that is commonly used in operating systems to interleave processes and has been modified to execute behavior trees.

**query view transformation** a standard allowing information between different metamodels to be shared.

**reverse modeling** the creation of a model from an existing system.



**structure tree** the third integrated view of the behavior modeling language. Still in the early stages of being formalised and integrated into the behavior modeling process.

**subsumption architecture** an architecture consisting of a number of layers of behavior that are directly coupled between sensors and actuators. Fixed priority arbitration is used to resolve competition between different layers, allowing a higher level behavior to subsume a lower level behavior and inhibit its output to actuators.

**system-component boundary** the system interacts with the components across this boundary.

**system-environment boundary** the system interacts with the environment across this boundary.

**UML diagram interchange** describes a means for communicating UML models for achieving tool interoperability.

**UML profiles** light-weight extension mechanism for creating domain specific languages by language exploitation within UML2.

**unified modeling language** an object-oriented modeling notation that is widely regarded as an industry standard language for communicating requirements, architectures and designs.

**visually integrated component system** a behavior based component based software engineering approach that uses an integration server to link together several black-box components.



# Acronyms

**.NET** microsoft .NET framework.

**AAA** algorithm architecture adequation.

**ATL** ATLAS transformation language.

**ATP** automated train protection.

**BB** behavior based.

**BE** behavior engineering.

**BECIE** BE component integration environment.

**behavior DAG** behavior directed acyclical graph.

**BML** behavior modeling language.

**BMP** behavior modeling process.

**BRE** behavior run-time environment.

**BT** behavior tree.

**BTPA** behavior tree process algebra.

**CASE** computer aided software engineering.

**CBSE** component based software engineering.

**CBT** component behavior tree.

**CID** component interface diagram.

**CIM** computational independent model.

**CIN** component interaction network.

**CORBA** common object request broker architecture.

**CPNML** coloured Petri net modeling language.

**CT** composition tree.

**CWM** common warehouse metamodel.

**DBT** design behavior tree.

**DpyCT** deployment composition tree.

**DSL** domain specific language.

**DynCT** dynamic composition tree.

**eBRE** embedded behavior run-time environment.

**EC** exogenous connector.

**EJB** enterprise javabeans.

**FMEA** failure modes and effects analysis.

**FPGA** field programmable gate array.

**FSM** finite state machine.

**fUML** foundational subset for executable UML models.

**GPL** general purpose language.

**GUI** graphical user interface.

**HDL** hardware definition language.

**IBT** integrated behavior tree.

**ICs** integrated circuits.

**ICT** integrated composition tree.

**J2EE** java platform enterprise edition.

**JET** java emitter templates.

**LDD** language driven development.

**LOP** language oriented programming.

**M2M** model-to-model.

**M2T** model-to-text.

**MARTE** modeling and analysis of real-time and embedded systems.

**MBT** model behavior tree.

**MCT** model composition tree.

**MDA** model driven architecture.

**MDE** model driven engineering.

**MOF** meta object facility.

**OCL** object constraint language.

**OMG** object modeling group.

**OO** object orientation.

**OS** operating systems.

**PIM** platform independent model.

**PN** Petri net.

**PNML** petri net markup language.

**PSM** platform specific model.

**PT** place transition.

**QVT** query view transformation.

**RBT** requirement behavior tree.

**RCT** requirement composition tree.

**SoS** system of systems.

**SPEM** software process engineering metamodel.

**ST** structure tree.

**Sw&SE** software and systems engineering.

**SysML** systems modeling language.

**TTCN-3** Testing and Test Control Notation version 3.

**TTL** transistor-transistor logic.

**UML** unified modeling language.

**VHDL** VHSIC hardware description language.

**VICON** visually integrated component.

**XMI** XML metadata interchange.

**XML** extensible markup language.

**xtUML** Executable UML.

# Index

- BE component model, 100–103
- BE EMF editor, 125–128
- BE expression syntax, 101–105
- BE extension mechanism, 103–105
- behavior based approaches, 40
- behavior engineering, 7–11, 57–112
- behavior modeling language, 58–67
  - deployment composition tree, 105
  - design behavior tree, 105
  - dynamic composition tree, 105
  - integrated behavior tree, 71
  - integrated composition tree, 71
  - model behavior tree, 73
  - model composition tree, 73
  - requirement behavior tree, 68
  - requirement composition tree, 68
- behavior modeling process, 58
  - design, 105–108
  - fitness for purpose test, 71–73
  - formalisation, 68
  - specification, 73–75
- behavior run-time environment, 101
- behavior trees, 59–66
  - behavior tree syntax, 60, 65–66
  - behavior types
    - event, 62
    - guard, 62
    - input, 62
    - output, 62
    - selection, 62
    - state realisation, 62
- composing
  - alternative branching, 65
  - atomic composition, 65
  - parallel branching, 65
  - sequential composition, 65
- operators
  - branch kill, 65
  - conjunction, 65
  - disjunction, 65
  - exclusive or, 65
  - reference, 65
  - reversion, 65
  - synchronisation, 65



- BT M2M transformation language, 130–135
- co-design, 155–157
- co-modeling, 13, 153–186
- component based software engineering, 43
- component behavior tree, 112
- component interaction network, 112
- component interface diagram, 112
- composition trees, 66–67
- computer aided software engineering, 18
- domain specific languages, 9–10, 51–54
- embedded BRE, 124
  - BE component model, 100–103
  - process control model, 140–144
- executable UML, 30–32
- exogenous connectors, 44
- finite state machine, 34
- forward modeling, 11, 121
- general purpose language, 9
- interaction axiom, 71
- interactive modeling, 11, 153
- language exploitation, 9
- language invention, 9
- meta object facility, 23
- metamodel, 19
- metamodeling, 23–25
- model driven architecture, 17
- model driven engineering, 8–11, 17–56
- model-to-model transformations, 128–135
- model-to-text transformations, 137–140
- object oriented, 39
- petri nets, 34
- platform independent model, 22
- platform specific model, 22
- precondition axiom, 71
- reverse modeling, 11
- scaleability, 3–12
- subsumption architecture, 42
- system of systems, 239
- traceability status
  - deleted behavior, 63
  - design refinement, 63
  - implied behavior, 63
  - missing behavior, 63
  - original behavior, 63
  - updated behavior, 63
- UML profiles, 29
- unified modeling language, 10, 26–30
- visually integrated component system, 44