UNIVERSITY OF SOUTHAMPTON

FACULTY OF PHYSICAL AND APPLIED SCIENCES

Electronics and Computer Science

An Approach to Atomicity Decomposition in the Event-B Formal Method

by

Asieh Salehi Fathabadi

Thesis for the degree of Doctor of Philosophy

August 2012

UNIVERSITY OF SOUTHAMPTON

ABSTRACT

FACULTY OF PHYSICAL AND APPLIED SCIENCES
Electronics and Computer Science

Doctor of Philosophy

AN APPROACH TO ATOMICITY DECOMPOSITION IN THE EVENT-B FORMAL METHOD

by Asieh Salehi Fathabadi

Formal methods are mathematically based techniques and tools to model software and hardware systems. Event-B is a formal method that emerged over the last decade as an evolution of classical B. Event-B is supported by an open and extensible Eclipse-based tool-set, called Rodin. Rodin provides an integrated environment supporting the whole process of multi-stage modelling and handling of the associated proofs. Rodin extensibility is exploited by developing a number of plug-ins to extend the main platform capabilities. During recent years, Event-B and Rodin have been used to model some real-world complex systems and prove consistency properties of them. However developing models of large and complex systems is not an easy task, since it can result in complex models and difficult proofs. There are some techniques in Event-B which can help to tackle the difficulties of modelling complex systems; refinement and model decomposition are two examples. Atomicity decomposition was recently introduced as another technique to help with the structuring of refinement-based development of complex systems in Event-B.

In this research, we have investigated how the development process with Event-B can be enriched further by using the atomicity decomposition approach. The atomicity decomposition approach provides a graphical notation to structure refinement and to support the explicit sequencing of events in an Event-B model. In this approach, modelling usually starts with a single atomic event of the system which is split to two or more sub-events in the next refinement level.

We have further developed the atomicity decomposition patterns and features. A formal description of the atomicity decomposition language is presented. The transformation rules from an atomicity decomposition diagram to the Event-B model are defined. The atomicity decomposition diagrams can be transformed to Event-B models using these rules. Exploiting the extensibility of the Rodin platform, a Rodin plug-in tool was developed to provide atomicity decomposition support in Event-B. The modelling and tool extensions developed in this thesis are applied to two complex case studies, the Media Channel System and the BepiColombo System. We present an evaluation of the atomicity decomposition approach using insights gained from these case studies.

Contents

Declaration of Authorship **xix**

Acknowledgements **xxi**

1 Introduction **1**

1.1 Thesis Motivation and Contribution . . . 2

1.2 Outcomes and Thesis Organisation . . . 4

2 Background **5**

2.1 Introduction . . . 5

2.2 Formal Methods . . . 5

2.2.1 Overview of Modelling . . . 5

2.2.2 Definition of Formal Methods . . . 6

2.2.3 Refinement . . . 6

2.3 Overview of Some Formal Modelling Languages . . . 7

2.3.1 VDM . . . 7

2.3.2 Z . . . 8

2.3.3 B-Method . . . 8

2.3.4 CSP . . . 9

2.3.5 Action Systems . . . 9

2.3.6 A Comparison . . . 10

2.4 Event-B . . . 10

2.4.1 The Event-B Definition . . . 10

2.4.2 Event-B Structure and Notation . . . 11

2.4.2.1 Context Structure . . . 11

2.4.2.2 Machine Structure . . . 12

2.4.2.3 Events . . . 12

2.4.3 Refinement in Event-B . . . 13

2.4.4 Proof Obligations . . . 15

2.4.5 Rodin as an Event-B Tool . . . 17

2.4.6 A Comparison Between Event-B and Other Formal Methods . . . 17

2.5 Event-B Model Decomposition . . . 18

2.5.1 Overview . . . 18

2.5.2 Decomposition Styles . . . 19

2.5.2.1 Shared Variable Style . . . 20

2.5.2.2 Shared Event Style . . . 20

3 Atomicity Decomposition Part 1 - Overview and Background **23**

3.1 Introduction . . . 23
3.2 Overview of Atomicity Decomposition Diagram in Event-B . . . 23
3.3 Event-B Refinement and Atomicity Decomposition Diagrams . . . 25
3.4 Examples of Application . . . 26
3.5 Conclusion . . . 29

4 Atomicity Decomposition Part 2 - Patterns and Features 31
4.1 Introduction . . . 31
4.2 Atomicity Decomposition Diagram Patterns . . . 31
4.2.1 Introduction . . . 31
4.2.2 Sequence Pattern . . . 32
4.2.3 Loop Pattern . . . 37
4.2.4 and-constructor Pattern . . . 40
4.2.5 or-constructor Pattern, Multiple Choice . . . 42
4.2.6 xor-constructor Pattern, Exclusive Choice . . . 45
4.2.7 all-replicator Pattern . . . 48
4.2.8 some-replicator Pattern . . . 51
4.2.9 one-replicator Pattern . . . 53
4.3 Additional Features of the Atomicity Decomposition Approach . . . 56
4.3.1 The Most Abstract Level . . . 56
4.3.2 Combined Atomicity Decomposition Diagram . . . 56
4.3.3 Several Atomicity Decompositions for a Single Event . . . 57
4.3.4 Strong Sequencing versus Weak Sequencing . . . 58
4.3.5 Loop Resetting Event . . . 59
4.4 Different Approaches to Model Ordering in Event-B . . . 61
4.5 Related Works and Comparison . . . 65
4.6 Conclusion . . . 68

5 Atomicity Decomposition Part 3 - Language Description and Translation Rules 69
5.1 Introduction . . . 69
5.2 An Example . . . 69
5.3 Atomicity Decomposition Language Specification . . . 70
5.4 Atomicity Decomposition Translation Rules (TRs) to Event-B . . . 73
5.4.1 Introduction . . . 73
5.4.2 Auxiliary Functions Definitions . . . 75
5.4.2.1 Traversing Functions . . . 75
5.4.2.2 Functions to Build Sequencing Invariants/Guards . . . 79
5.4.2.3 Predecessor/Successor Functions . . . 80
5.4.3 Translating a Leaf . . . 83
5.4.3.1 TR_leaf1: mapping a leaf to a variable . . . 83
5.4.3.2 TR_leaf2: mapping a non-replicator leaf to a typing invariant . . . 84
5.4.3.3 TR_leaf3: mapping a replicator leaf to a typing invariant 85
5.4.3.4 TR_leaf4: mapping a leaf to a sequencing invariant . . . 86
5.4.3.5 TR_leaf5: mapping a solid leaf to a gluing invariant . . . 88
5.4.3.6 TR_leaf6: mapping a solid leaf to a refining event . . . 88

5.4.3.7 TR_leaf7: mapping a dashed leaf to a non-refining event 90
5.4.3.8 TR_leaf8: mapping a leaf to a sequencing guard 91
5.4.3.9 TR_leaf9: mapping a non-replicator leaf to a guard . . . 91
5.4.3.10 TR_leaf10: mapping a replicator leaf to a guard 93
5.4.3.11 TR_leaf11: mapping a non-replicator leaf to an action . . 94
5.4.3.12 TR_leaf12: mapping a replicator leaf to an action 95
5.4.4 Translating the xor-constructor 96
5.4.4.1 TR_xor1: mapping a solid xor-constructor to a gluing invariant 96
5.4.4.2 TR_xor2: mapping a xor-constructor to an invariant . . . 97
5.4.4.3 TR_xor3: mapping a xor-constructor to guards 98
5.4.5 Translating the one-replicator 100
5.4.5.1 TR_one1: mapping a solid one-replicator to a gluing invariant 100
5.4.5.2 TR_one2: mapping an one-replicator to (an) invariant(s) 100
5.4.5.3 TR_one3: mapping an one-replicator to (a) guard(s) . . . 102
5.4.6 Translating the Loop Constructor 104
5.4.6.1 TR_loop1: mapping a loop to (a) guard(s) 104
5.4.6.2 TR_loop2: mapping a loop to (a) guard(s) 105
5.4.6.3 TR_loop3: mapping a loop to a resetting event 107
5.4.7 Translating a Weak Sequencing Flow 108
5.4.7.1 TR_weak1: mapping a weak sequencing flow to (a) invariant(s) 108
5.4.7.2 TR_weak2: mapping a weak sequencing flow to (a) guard(s) 109
5.5 Conclusion 110

6 Tool Development: Atomicity Decomposition Plug-in in Rodin platform 113
6.1 Introduction 113
6.2 Architecture and Technologies 114
6.3 User Interface 117
6.4 Conclusion 117

7 Case Studies 121
7.1 Introduction 121
7.2 Media Channel System (Published in FMCO 2009 Conference) 123
7.2.1 Overview of the Media Channel System 123
7.2.1.1 Requirements for Establishing a Media Channel 124
7.2.1.2 Requirements for Modifying an Established Media Channel 124
7.2.1.3 Requirements for Closing an Established Media Channel 125
7.2.2 Abstract Specification 125
7.2.2.1 Static Part of the Specification 125
7.2.2.2 Events and Dynamic Part of the Specification 126
7.2.3 1st Refinement: Breaking the Atomicity of Establish Media Channel 127
7.2.4 2nd Refinement: Breaking the Atomicity of Modify Media Channel 129
7.2.5 3rd Refinement: Breaking the Atomicity of Close Media Channel . 135
7.2.6 4th Refinement: Second Level Breaking the Atomicity of Modify Media Channel 135

7.2.7 5th Refinement: Second Level Breaking the Atomicity of Close Media Channel . . . 137
7.2.8 Evaluation of Manual Event-B Model and Automatic Event-B Model 138
7.2.8.1 Variable Naming Protocol . . . 138
7.2.8.2 Different Approaches to Model Ordering In Event-B Model 138
7.2.8.3 One More Refinement Level in the Manual Model . . . 139
7.2.8.4 Weak Guard versus Strong Guard . . . 140
7.2.8.5 Tool Application: Atomicity Decomposition Model of the Media Channel System . . . 142
7.2.9 Overview of Proof Obligations . . . 142
7.3 BepiColombo Space Craft System (Published in NFM 2011 Symposium) . 144
7.3.1 Overview of the BepiColombo System . . . 144
7.3.2 Modelling Architecture . . . 145
7.3.3 Abstract Specification . . . 146
7.3.3.1 Static Part of the Specification . . . 146
7.3.3.2 Events and Dynamic Part of the Specification . . . 147
7.3.4 1st Refinement: Refining the Validation Phase . . . 150
7.3.5 2nd Refinement: Refining the Semantic Check . . . 152
7.3.6 3rd Level of Refinement . . . 153
7.3.7 Decomposing BepiColombo Model to Core and Device Sub-models 155
7.3.7.1 Combining Atomicity Decomposition and Model Decomposition in Event-B . . . 155
7.3.7.2 Shared Event Model Decomposition . . . 155
7.3.8 4th and 5th Refinements: Core Sub-model Refinements . . . 158
7.3.9 Evaluation of Manual Event-B Model and Automatic Event-B Model 161
7.3.9.1 A Merged Guard versus Separate Guards . . . 161
7.3.9.2 Gluing Invariants . . . 162
7.3.9.3 Model Decomposition . . . 163
7.3.9.4 some-replicator, one-replicator . . . 164
7.3.9.5 Naming Protocol . . . 164
7.3.9.6 Tool Application: Atomicity Decomposition Model of the BepiColombo System . . . 165
7.3.10 Overview of Proof Obligation . . . 165
7.4 Conclusion . . . 167

8 Evaluation of Atomicity Decomposition in Case Study Developments 169
8.1 Introduction . . . 169
8.2 Exploring Alternatives . . . 170
8.3 Preventing of Wrong Event Decomposition . . . 170
8.4 Events Tracking . . . 171
8.5 Requirements Clarification . . . 173
8.6 Combining Atomicity Decomposition and Model Decomposition . . . 174
8.7 New Constructors and Features . . . 177
8.7.1 Introduction . . . 177
8.7.2 New Constructors . . . 177
8.7.3 Additional Features . . . 177
8.8 Conclusion . . . 178

9 Conclusions and Future Works 181
9.1 Conclusion . . . 181
9.2 Future Works . . . 183

A The Event-B Model of the Media Channel System 185
A.1 Abstract Specification . . . 185
A.1.1 Context: *C1* . . . 185
A.1.2 Machine: *M0* . . . 185
A.2 1st Refinement . . . 186
A.2.1 Context: *C2* . . . 186
A.2.2 Machine: M1 . . . 186
A.3 2nd Refinement . . . 189
A.3.1 Machine: M2 . . . 189
A.4 3rd Refinement . . . 195
A.4.1 Machine: M3 . . . 195
A.5 4th Refinement . . . 200
A.5.1 Machine: *M4* . . . 200
A.6 5th Refinement . . . 208
A.6.1 Machine: *M5* . . . 208

B The Event-B Model of the BepiColombo System 217
B.1 Abstract Specification . . . 217
B.1.1 Context: *C0* . . . 217
B.1.2 Machine: *M0* . . . 217
B.2 1st Refinement . . . 219
B.2.1 Machine: M1 . . . 219
B.3 2nd Refinement . . . 221
B.3.1 Context: *C1* . . . 221
B.3.2 Machine: M2 . . . 222
B.4 3rd Refinement . . . 225
B.4.1 Context: *C2* . . . 225
B.4.2 Machine: M3 . . . 225
B.5 Core Sub-model . . . 230
B.5.1 Context: *Context_M3* . . . 230
B.5.2 Machine: M3 . . . 230
B.5.3 1st Refinement . . . 234
B.5.3.1 Context: *Context_M4* . . . 234
B.5.3.2 Machine: M4 . . . 234
B.5.4 2nd Refinement . . . 239
B.5.4.1 Context: *Context_M5* . . . 239
B.5.4.2 Machine: M5 . . . 239
B.6 Device Sub-model . . . 245
B.6.1 Context: *Context_M3* . . . 245
B.6.2 Machine: M3 . . . 245

References 249

List of Figures

2.1 Machine and Context Relationships . . . 11
2.2 Structure of a Context . . . 12
2.3 Structure of a Machine . . . 12
2.4 An Event-B Model (Context Ctx, Abstract Machine $M1$, Concrete Machine $M2$) . . . 15
2.5 Model Decomposition in Event-B . . . 19
2.6 Shared Variable Decomposition . . . 20
2.7 Shared Event Decomposition . . . 21

3.1 Atomicity Decomposition Diagram . . . 24
3.2 Event-B Model of Atomicity Decomposition Diagram in Figure 3.3 . . . 25
3.3 Atomicity Decomposition Diagram of Event-B Model in Figure 3.2 . . . 26
3.4 Abstract Model of an Outputting System . . . 27
3.5 Atomicity Decomposition Diagram of an Outputting System . . . 27
3.6 Event-B Refinement of an Outputting System . . . 28
3.7 Atomicity Decomposition Diagram of File Write . . . 28
3.8 Invariants of File Write Refinement Model . . . 29
3.9 Event-B Model of File Write . . . 29

4.1 The Most Abstract Level Diagrams . . . 56
4.2 Combined Atomicity Decomposition Diagram . . . 57
4.3 Several Atomicity Decomposition for a Single Event, *Event_a* . . . 58
4.4 Strong Sequencing, Weak Sequencing . . . 58
4.5 Loop Resetting Example . . . 60
4.6 Loop Resetting as a Separate Event . . . 60
4.7 Loop Resetting in the Last Event . . . 61
4.8 Loop Resetting in the First Event . . . 62
4.9 Loop Resetting Example . . . 62
4.10 Loop Resetting with Parameter . . . 62
4.11 Subset Sets . . . 63
4.12 Disjoint Sets . . . 63
4.13 State Function . . . 64

5.1 An Example of Atomicity Decomposition Diagram . . . 70
5.2 Syntax of Atomicity Decomposition Language (ADL) . . . 71
5.3 ABNF of the Diagram in Figure 5.1 . . . 73
5.4 Invalid Combination of the Constructors . . . 73
5.5 *list_of_leaves* Function . . . 76

5.6 *disjunction_of_leaves* Function . . . 77
5.7 *conjunction_of_leaves* Function . . . 77
5.8 *union_of_leaves* Function . . . 77
5.9 *build_seq_inv* Function . . . 78
5.10 *build_seq_grd* Function . . . 78
5.11 Sequencing Between Two Leaf Events . . . 79
5.12 *seq_inv* Function . . . 80
5.13 *seq_grd* Function . . . 81
5.14 Predecessor of a Subtree . . . 81
5.15 *predecessor* Function . . . 82
5.16 *successor* Function . . . 82
5.17 TR_leaf1: mapping a leaf to a variable . . . 83
5.18 Application of TR_leaf1 in the Example of Figure 5.1, First Refinement Level . . . 84
5.19 TR_leaf2: mapping a non-replicator leaf to a typing invariant . . . 85
5.20 Application of TR_leaf2 in the Example of Figure 5.1, First Refinement Level . . . 85
5.21 TR_leaf3: mapping a replicator leaf to a typing invariant . . . 86
5.22 Application of TR_leaf3 in the Example of Figure 5.1, First Refinement Level . . . 86
5.23 TR_leaf4: mapping a leaf to a sequencing invariant . . . 87
5.24 Application of TR_leaf4 in the Example of Figure 5.1, Second Refinement Level . . . 87
5.25 TR_leaf5: mapping a solid leaf to a gluing invariant . . . 88
5.26 Application of TR_leaf5 in the Example of Figure 5.1, First Refinement Level . . . 88
5.27 TR_leaf6: mapping a solid leaf to a refining event . . . 89
5.28 Application of TR_leaf6 in the Example of Figure 5.1, First Refinement Level . . . 89
5.29 TR_leaf7: mapping a dashed leaf to a new event . . . 90
5.30 Application of TR_leaf7 in the Example of Figure 5.1, First Refinement Level . . . 90
5.31 TR_leaf8: mapping a leaf to a sequencing guard . . . 91
5.32 Application of TR_leaf8 in the Example of Figure 5.1, Second Refinement Level . . . 92
5.33 TR_leaf9: mapping a non-replicator leaf to a guard . . . 92
5.34 Application of TR_leaf9 in the Example of Figure 5.1, First Refinement Level . . . 93
5.35 TR_leaf10: mapping a replicator leaf to a guard . . . 93
5.36 Application of TR_leaf10 in the Example of Figure 5.1, First Refinement Level . . . 94
5.37 TR_leaf11: mapping a non-replicator leaf to an action . . . 94
5.38 Application of TR_leaf11 in the Example of Figure 5.1, First Refinement Level . . . 95
5.39 TR_leaf12: mapping a replicator leaf to an action . . . 95
5.40 Application of TR_leaf12 in the Example of Figure 5.1, First Refinement Level . . . 96

5.41 TR_xor1: mapping a solid xor-constructor to a gluing invariant 96
5.42 Application of TR_xor1 in the Example of Figure 5.1, First Refinement Level 97
5.43 TR_xor2: mapping a xor-constructor to an invariant 97
5.44 Application of TR_xor2 in the Example of Figure 5.1, Second Refinement Level 98
5.45 TR_xor3: mapping a xor-constructor to guards 99
5.46 Application of TR_xor3 in the Example of Figure 5.1, Second Refinement Level 99
5.47 TR_one1: mapping a solid one-replicator to a gluing invariant 100
5.48 Application of TR_one1 in the Example of Figure 5.1, First Refinement Level 100
5.49 TR_one2: mapping an one-replicator to (an) invariant(s) 101
5.50 Application of TR_one2 in the Example of Figure 5.1, Second Refinement Level 102
5.51 TR_one3: mapping an one-replicator to (a) guard(s) 103
5.52 Application of TR_one3 in the Example of Figure 5.1, Second Refinement Level 103
5.53 TR_loop1: mapping a loop to (a) guard(s) 104
5.54 Application of TR_loop1 in the Example of Figure 5.1, Second Refinement Level 105
5.55 TR_loop2: mapping a loop to a resetting event 106
5.56 Application of TR_loop2 in the Example of Figure 5.1, Second Refinement Level 106
5.57 TR_loop3: mapping a loop to a resetting event 107
5.58 Application of TR_loop3 in the Example of Figure 5.1, Second Refinement Level 108
5.59 TR_weak1: mapping a weak sequencing flow to (a) guard(s) 109
5.60 Application of TR_weak1 in the Example of Figure 5.1, Second Refinement Level 109
5.61 TR_weak2: mapping a weak sequencing flow to (a) guard(s) 110
5.62 Application of TR_weak2 in the Example of Figure 5.1, Second Refinement Level 110

6.1 Atomicity Decomposition Plug-in Architecture 114
6.2 Atomicity Decomposition Language, EMF Meta-model 116
6.3 An ETL rule, Corresponded to TR_xor1 (Section 5.4.4.1) 116
6.4 User Interface 118
6.5 Event-B Model of the Instance of the Atomicity Decomposition Model in Figure 6.4 119

7.1 A Simple Image of the Media Channel between Two Endpoints 123
7.2 Protocol of the Media Channel System 124
7.3 Context C1, Media Channel System 125
7.4 The Atomicity Decomposition Diagram, M0, Media Channel System 126
7.5 Variables and Invariants, M0, Media Channel System 126
7.6 Event-B Model, M0, Media Channel System 127
7.7 Establish Media Channel Scenario 128

7.8 Breaking the Atomicity of Establish a Media Channel, M1 128
7.9 Manual Variables and Invariants, M1, Media Channel System 129
7.10 Event-B Model, M1, Media Channel System 130
7.11 Event-B Model, M1, Media Channel System 130
7.12 Event-B Model, M1, Media Channel System 130
7.13 Modify Set of Codecs of a Media Channel by Selector Scenario 131
7.14 Modify Codec of a Media Channel by Descriptor Scenario 131
7.15 Modify Port of a Media Channel by Descriptor Scenario 131
7.16 Breaking the Atomicity of Modify a Media Channel, M2 133
7.17 Event-B Model, M2, Media Channel System 133
7.18 Event-B Model, M2, Media Channel System 134
7.19 Loop Resetting, M2, Media Channel System 134
7.20 Breaking the Atomicity of Close a Media Channel, M3 135
7.21 Further Breaking the Atomicity of Modify a Media Channel, M4 136
7.22 Event-B Model, M4, Media Channel System 136
7.23 Close a Media Channel Scenarios 137
7.24 Further Breaking the Atomicity of Close a Media Channel, M5 137
7.25 Event-B Model, M5, Media Channel System 138
7.26 Manual Event-B Model, M0, Media Channel System 139
7.27 Manual Event-B Model, M1, Media Channel System 140
7.28 Automatic Event-B Model, M1, Media Channel System 141
7.29 Combined Atomicity Decomposition Diagram of *modify* Event, Media Channel System 142
7.30 Atomicity Decomposition Model of the Media Channel System 143
7.31 Proof Obligation Statistics for the Automatic Media Channel Event-B Model 143
7.32 Proof Obligation Statistics for the Manual Media Channel Event-B Model 144
7.33 High-level Architecture of BepiColombo SIXS/MIXS OBSW 145
7.34 Development Architecture of the BepiColombo Event-B Model 146
7.35 Context C0, BepiColombo System 147
7.36 The Atomicity Decomposition Diagrams, M0, BepiColombo System . . . 148
7.37 Variables and Invariants, M0, BepiColombo System 149
7.38 Event-B Model, M0, BepiColombo System 149
7.39 Event-B Model, M0, BepiColombo System 150
7.40 Event-B Model, M0, BepiColombo System 150
7.41 The Atomicity Decomposition Diagrams, M1, BepiColombo System . . . 151
7.42 Event-B Model, M1, BepiColombo System 152
7.43 Gluing Invariants, M1, BepiColombo System 152
7.44 Context C1, BepiColombo System 152
7.45 The Atomicity Decomposition Diagram, M2, BepiColombo System 153
7.46 Event-B Model, M2, BepiColombo System 153
7.47 The Atomicity Decomposition Diagrams, M3, BepiColombo System . . . 154
7.48 Model Decomposition, Shared Events Style, BepiColombo System 156
7.49 Shared Events, BepiColombo System 156
7.50 An Instance of a Shared Event Before Model Decomposition, BepiColombo System 157

7.51 Instances of a Shared Event After Model Decomposition, BepiColombo System . . . 158
7.52 The Atomicity Decomposition Diagram in the Core Sub-Model, M4 and M5, BepiColombo System . . . 159
7.53 Event-B Model of Weak Sequencing in the Core Sub-Model, M4 and M5, BepiColombo System . . . 159
7.54 The Atomicity Decomposition Diagrams in the Core Sub-Model, M4 and M5, BepiColombo System . . . 160
7.55 one-replicator Invariants, M4, BepiColombo System . . . 160
7.56 one-replicator Invariants, M5, BepiColombo System . . . 160
7.57 Event-B Model of one-replicator and Weak Sequencing in the Core Sub-Model, M4 and M5, BepiColombo System . . . 161
7.58 Manual Event-B Model, M1, BepiColombo System . . . 162
7.59 Combined Atomicity Decomposition Diagram of *TCExecute_Ok* Event, BepiColombo System . . . 163
7.60 some-replicator Event, Manual Event-B Model, BepiColombo System . . . 164
7.61 Atomicity Decomposition Model of the BepiColombo System . . . 165
7.62 Proof Obligation Statistics for the Automatic BepiColombo Event-B Model 167
7.63 Proof Obligation Statistics for the Manual BepiColombo Event-B Model . 167

8.1 Decomposing Atomicity of *modify* Event in Two Levels of Refinement . . 170
8.2 Decomposing Atomicity of *modify* Event in One Level of Refinement . . . 170
8.3 Wrong Atomicity Decomposition . . . 171
8.4 Overall Refinement Structure After Model Decomposition, BepiColombo System . . . 172
8.5 Atomicity Decomposition Diagram Without Considering Data Generation Requirement, BepiColombo System . . . 173
8.6 Atomicity Decomposition Diagram After Clarifying Data Generation Requirement, BepiColombo System . . . 174
8.7 Overall Refinement Structure, Abstract Model and Two Refinement Levels, BepiColombo System . . . 175
8.8 Event Sequencing Before Model Decomposition, BepiColombo System . . 176
8.9 Event Sequencing Preserved After Model Decomposition, BepiColombo System . . . 176

List of Tables

2.1 Event Forms 13
2.2 Action Forms 13
2.3 Proof Obligations in Event-B 16

4.1 Sequence Pattern 33
4.2 Loop Pattern 38
4.3 and-constructor Pattern 42
4.4 or-constructor Pattern 44
4.5 xor-constructor Pattern 48
4.6 all-replicator Pattern 50
4.7 some-replicator Pattern 52
4.8 one-replicator Pattern 55

7.1 Summary of Event-B Refinements, BepiColombo System 147
7.2 Summary of the Automatic BepiColombo Development, Number of Variables and Events 166
7.3 Summary of the Manual BepiColombo Development, Number of Variables and Events 166

Declaration of Authorship

I, Asieh Salehi Fathabadi , declare that the thesis entitled *An Approach to Atomicity Decomposition in the Event-B Formal Method* and the work presented in the thesis are both my own, and have been generated by me as the result of my own original research. I confirm that:

- this work was done wholly or mainly while in candidature for a research degree at this University;
- where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated;
- where I have consulted the published work of others, this is always clearly attributed;
- where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work;
- I have acknowledged all main sources of help;
- where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself;
- parts of this work have been published as: [1] and [2]

Signed:...

Date:..

Acknowledgements

I can not express my gratitude enough to my supervisor Prof. Michael Butler for his inspiration, advice and cordiality throughout the PhD. Without his guidance and support I would not have been able to complete this thesis. I learnt much from him not only in terms of academic research but also life experience and patience.

Special thanks go to Dr. Rezazadeh for his advice and other ESS group members for their help and warm support.

I would like to thank my Iranian friends without whom my experience in Southampton would not be the same. Specially to Amir reza for his patience and understanding whilst I pursue my dream of a PhD.

I am very much indebted to my parents and my sisters for their support and for giving me energy, hope and warmth. Without their encouragement, it would not have been possible to finish, or even begin, this work. My special thank goes to my father; following his career path was one of the significant motivation for me to live away from my hometown.

Chapter 1

Introduction

Formal Methods [3, 4] are mathematically based modelling techniques used to specify and verify hardware and software systems. Z [5, 6], VDM [7, 8] and B-Method (also known as classical B) [9, 10, 11, 12] are among the most recent formal methods.

Event-B [4, 13, 14] is an evolution of the classical B. Event-B uses the concept of Refinement [8] in modelling. Event-B modelling starts with an abstract specification of a system. Details are added during refinement steps in order to arrive at a more detailed model. The mathematical language of Event-B is base on set theory and first order logic. Based on the Event-B language, a set of proofs can be produced and discharged for each Event-B model. Rodin [13, 15, 16, 17] is an open source, extensible and integrated modelling tool supporting Event-B. This tool is not only used as a modelling environment, but also provides an integrated environment for proving properties of models. Formal modelling is not just about constructing descriptions, but proving some properties about the formal models is equally important. Rodin provides an integrated environment for both modelling and proving. Extensibility of Rodin makes it easy for new features to be added to it. During recent years, some Eclipse based plug-ins were developed and added to Rodin. ProB [18] as an animator, UML-B [19] as graphical environment provider, B2Latex [20] as a translator from B to Latex and model decomposition [21] which allows decomposition of a model into sub-models, have been developed and added to Rodin.

Recently Event-B has been applied to developing some notable industrial cases [22, 23]. However building models of large and complex systems results in large and complex models and proofs. Dealing with large and complex models and proofs is a difficult and time consuming task. Some techniques such as Atomicity Decomposition [24] sometimes called Event Decomposition can help to solve this difficulty. Atomicity decomposition augments refinement in Event-B in order to structure the refinement process.

This thesis focusses on atomicity decomposition as an approach for modelling large and complex systems using Event-B. This approach enables developers to structure the refinement process in Event-B. Refinement in Event-B is too general. It does not explicitly show all of the relations between behaviors of the abstract model, called abstract events, and the behaviors of the refinement, called the concrete events. Atomicity decomposition diagrams enables us to explicitly show the relationships between abstract events and concrete events. It also imposes an explicit ordering between events within a single level of refinement.

Based on the atomicity decomposition approach, during each refinement level, abstract events can be decomposed into several sub-events using a provided graphical notation. This approach demonstrates how coarse-grained atomicity is refined to more fine-grained atomicity. This approach is also capable of showing an overall structure of several refinement levels. Therefore it provides an effective way to handle complex development. On the other hand, providing decomposition constructs and patterns, makes the modelling of large systems more manageable. Using constructs and patterns, we can achieve reusability in Event-B development. In the atomicity decomposition approach, patterns refer to common atomicity decomposition styles.

To implement atomicity decomposition, tool support has been developed as a plug-in for the Rodin platform. This tool provides an environment to define atomicity decomposition rules and patterns. The developed atomicity decomposition plug-in can help to ease the burden of the manual work.

1.1 Thesis Motivation and Contribution

The modelling of large and complex systems can result in large and complex models, and proofs [25]. Refinement, generic instantiation and decomposition, are three techniques which can help us to overcome this difficulty [25]. Decomposition [24, 26] in Event-B has two types; Model Decomposition and Atomicity Decomposition. In the case of model decomposition, which will be explained in Section 2.5, when a model becomes too large, we can split it into small sub-models which are much easier to tackle. Through generic instantiation an existing model can be used as generic and instantiated to be used in another development [27]. Our focus in this thesis is the latest, Atomicity Decomposition.

We are aware that refinement, as will be explained in Sections 2.2.3 and 2.4.3, is a useful modelling technique and can be a good solution for those difficulties, but it can not solve all complexity problems since it does not show the relations between refinement levels. Clear relationships between actions of refinement levels in a graphical environment, which is done using the atomicity decomposition approach, make complex models more understandable.

This thesis contributes to the development of systems using the Event-B formal method and Rodin tool-set. Thesis contributions are listed below :

- **Structure Refinement in Event-B**
  One of the important contributions of the atomicity decomposition approach, which is first outlined in [24], is that it shows the relationships between the earlier level of modelling called the abstract model and the corresponding later refinement level called the concrete model. Whereas just by applying refinement technique into the Event-B text we are not able to show the relationships between refinement levels, Section 3.3. Therefore we can say that atomicity decomposition is an approach in Event-B which augments Event-B refinement with additional structure.

  Atomicity decomposition is a representation of the refinement process in Event-B which explicitly presents relationships between actions of different refinement levels. Using atomicity decomposition diagrams through refinement levels make this technique a visual refinement strategy. Also it has the ability to show the explicit ordering between actions of one level of refinement, Section 3.2.

- **Evaluation via Complex Case Studies**
  The atomicity decomposition approach has been applied to the development process of two complex case studies, the Media Channel System and the BepiColombo System (Chapter 7). These developments highlight the benefits of the atomicity decomposition approach, during the development process of a complex system. An evaluation of the atomicity decomposition approach using insights gained from these case studies is outlined in Chapter 8. The application of the atomicity decomposition approach in development of the media channel system is published in "Formal Methods for Components and Objects" (FMCO) 2009 conference [1]. And the BepiColombo development using the atomicity decomposition approach is published in "Nasa Formal Methods" (NFM) 2011 symposium [2].

- **Constructs and Patterns**
  Atomicity decomposition assists us in the development of refinement patterns, and this result can decrease the modelling effort. As a result of developing the case studies some new construct patterns and features have been discovered which are presented in Chapter 4.

- **Language Description and Translation Rules**
  Defining atomicity decomposition patterns helped us to describe the atomicity decomposition language in a formal way. The general and formal description of the language of atomicity decomposition diagrams and rules of translating each element of diagram to Event-B model are presented in Chapter 5.

- **Tool Support and Automatic Generation of Models**
  Developing the atomicity decomposition plug-in in the Rodin platform as tool support for this approach; this results in automatic generation of an Event-B model from an atomicity decomposition diagram, which can decrease the modelling effort in complex systems. The tool development is presented in Chapter 6. We

present the work developed for the atomicity decomposition approach including the theory behind it (Chapter 5), the extension to tool support (Chapter 6) and the application to case studies (Chapter 7).

1.2 Outcomes and Thesis Organisation

This Report is organised as follows. Chapter 2 introduces the necessary background to understand the rest of the document. Some background knowledge about the formal method is outlined in Section 2.2, followed by introducing existing formal methods in Section 2.3. Then Section 2.4 focuses on Event-B and its structure and refinement. Finally Section 2.5 explains the background definition of model decomposition in Event-B which is used later in developing a case study, the BepiColombo system.

The atomicity decomposition approach is first introduced by Butler in [24]. Chapter 3 is a literature review of atomicity decomposition presented in [24]. In this chapter, the benefits of the approach in structuring refinement in Event-B is highlighted and two small examples are presented.

We manually applied the approach presented in [24] to two complex case studies, the media channel system and the BepiColombo system, an on-board instrument controller for a space craft. Some new construct patterns and features were discovered during the case study developments. These new patterns and features together with the existing ones from [24] are presented in Chapter 4.

Later using the pattern definitions, the general and formal description of the atomicity decomposition approach and rules of translating to Event-B model are presented in Chapter 5. The tool supported the atomicity decomposition approach is presented in Chapter 6.

Instead of the manual modelling, using the developed tool support we have developed the model of case studies for the second time in a semi-automatic way. Chapter 7 presents the application of the atomicity decomposition approach in the developments of case studies including manual and semi-automatic models. Then in Chapter 8 a critical evaluation of the atomicity decomposition approach is presented based on experience of the case study developments. Finally, there is a conclusion and explanation of future work.

Chapter 2

Background

2.1 Introduction

This chapter presents relevant background on modelling, formal methods and Event-B. Section 2.2 will give a brief overview of modelling, its difficulties and outlines the significant role of it as an early stage in the software development process. Then formal methods as modelling techniques will be presented. And it is followed by an overview of some formal methods in Section 2.3. Event-B as a formal method for specifying and proving about software and hardware systems, its notation and structure will be described in Section 2.4. This section also outlines the definition of refinement in Event-B and a brief explanation of Rodin, an open Eclipse based toolkit for modelling in Event-B. Finally an overview of model decomposition in Event-B which is later used in development of a case study is described in Section 2.5.

2.2 Formal Methods

2.2.1 Overview of Modelling

There is a big difference between modelling and programming. First the model of a system is not exactly the system; it means the model of a system is not executable like the program of a system. For example, one can not play with the model of a computer game. Moreover as Abrial says [3, 4], a program contains the algorithm whereas a model describes the properties of a program; in other words, the initial model of a program describes the way by which we can finally judge that the final program is correct. For example, the initial model of an array sorting program does not explain how to sort it. It rather explains what the properties of a sorted array are and what the relation is between the initial non-sorted array and the final sorted one.

One of the benefits of using modelling as a step in the development process of a system is minimizing failure risks and cost in the testing phase [3]. A model of a program comes with proofs which related to the program. In proving we make certain that all properties can be proved to be consistent. With using proofs, we reason about our models [25]. More precisely, the model of a program is not just descriptions of it; modelling can be accompanied by proving some consistency properties [3, 4].

2.2.2 Definition of Formal Methods

Formal methods can be defined as mathematically-based techniques which are used to specifying and reasoning about software and hardware systems [3, 4]. Holloway believes that engineers will consider formal methods [28].

The language of most formal methods is a language of classical logic and set theory. Abrial states that it is convenient to communicate models to everyone that has some mathematical background. Also using mathematical language will allow us to do some reasoning in the form of proofs [4, 29].

Rangarajan believes that using formal methods as a collection of mathematical activities and formal logic to specify and prove about systems has many valuable benefits [30]. First, considering formal methods as an early phase in the development process life cycle, results in early detection of defects, so it can play the role of a solution to heavy testing phase on final product which is well known to happen quite often too late in development process life cycle. Moreover, in the testing phase it is impossible to provide coverage of all possible interleaving and event orderings, whereas, by using model checkers and provers as formal analysis tools we can reach more fault conditions, so another benefit of using formal methods can be guarantee of correctness. Finally, the analytical nature of formal methods results in more reliable verification in large and complex systems compared to testing alone.

2.2.3 Refinement

Building a model, usually starts with a very abstract model of the system, and then gradually details are added through several modelling steps in such a way that leads us towards a suitable implementation; this approach is called refinement [8]. In other words, during refinement levels, the model becomes more and more precise and closer to the requirements. Roever and Engelhardt in [8], state that a useful analogy is that of the scientist looking through a microscope. The microscope does not change anything, some previously invisible parts of the reality are now revealed by the microscope.

From a given model M1, a new model M2 can be built as a refinement of M1. In this case, model M1 is called an abstraction of M2, and model M2 will said to be a concrete version of M1. A concrete model is said to refine its abstraction.

Refinement allows us to tackle the system complexity. Using refinement, instead of building a single model in a flat manner, we have a sequence of models, where each of them is supposed to be a refinement of the previous [25].

Refinement calculus is a formalized approach to stepwise refinement for program construction. The refinement calculus is a calculus of program transformation. It starts from abstract specification of a system. It is then refined by a series of transformations into executable program. Refinement calculus is originated by Back [31] and Morgan [32]. In Morgan's book the motivation was to link Z notation to an executable programming notation.

2.3 Overview of Some Formal Modelling Languages

Many formal methods have been proposed in recent years to improve software quality. These include specification and modelling languages as well as formal verification techniques, such as model checking, and theorem proving. Here we are going to briefly summaries some well known existing formal modelling languages:

2.3.1 VDM

The VDM [7, 8], (Vienna Development Method) is one of the longest-established formal methods for the development of computer-based systems, introduced by a research group of IBM laboratory in Vienna in the 1970s. It has grown to include a group of techniques and tools based on a formal specification language - the VDM Specification Language (VDM-SL). Jones claimed that it was developed in an industrial environment and was one of the most widely used formal methods in 1990s [7]. VDM supports writing specification and also discharging proof obligations that ensure that the specification can be proven to be consistent. All specification and proof obligation are written in term of predicates.

Use of VDM starts with a very abstract model and develops this into an implementation. Each step involves Data Reification, then Operation Decomposition. Data reification develops the abstract data types into more concrete data structures, while operation decomposition develops the (abstract) implicit specifications of operations and functions into algorithms that can be directly implemented in a computer language of choice [7].

2.3.2 Z

In 1977, Abrial proposed Z with the help of Schuman and Meyer [33]. It was developed further at Oxford University. The Z notation [5, 6], (pronounced zed) is a formal modelling language based on standard mathematical notation used in set theory and logic. The set theory used includes standard set operators, set comprehension, cartesian products, and power sets. The mathematical logic is a first-order predicate calculus. The Z notation is used for specifying, modelling and reasoning about computing systems. Jacky states that Z is just a notation, it is not a method and it can support many different methods [5]. Also as it mentioned in Section 2.2.1 like other formal notations, Z in not a programming language, so it is not an executable notation. Although Z is more popular than VDM, VDM has the composition and decomposition features [34].

A Z specification describes the state space together with a collection of operations. The Z refinement is defined between two Z specifications, allows both the state space and the individual operations to be refined. Operation refinement is the process of recasting each abstract operation into a concrete operation. Data refinement extends operation refinement by allowing the state space of the concrete operations to be different from the state space of the abstract operations. In order to specification structuring in Z, a schema notation is included in it [6]. Schema notation provides a framework for a textual combination of sections of mathematics. These sections of mathematics are called schemas.

2.3.3 B-Method

The B-Method (also known as classical B) [9, 10, 11, 12] is originally developed by Abrial in the mid 1980s. The B-Method is a model-based method for formal development of computer software systems. It has been used in major safety-critical system applications such as Metro Line 14 in Paris [10].

The B language is based on set theory including sets, relations and functions to define variables and predicate logic to specify invariants (constraints of variables). Generalized substitutions are used to specify operations, which allow deterministic and nondeterministic state transitions. B uses structuring mechanisms (machine, refinement and implementation) in organization of a development.

Compared to Z, B is more focused on refinement rather than just formal specification. In particular, there is better tool support [10] such as Atelier-B [35]. These tools support two main proof activities in B: consistency checking, shows that invariants are preserved by machine operations, and refinement checking, which prove the validity of each refined machine.

2.3.4 CSP

CSP (Communicating Sequential Processes) [36, 37] is a process algebra for modelling parallel processing and interaction between processes. A *process* in CSP is considered as a mathematical abstraction of interactions between a system and its environment. The behaviour of a system is described through processes. CSP allows the refinement of models.

The set of events in which a process P can engage is called its alphabet, written αP and represents the visible interface between the process and its environment [38]. The processes are constrained in the way they can engage in the events of its alphabet. A process interacts with its environment by synchronously engaging in atomic events. A sequence of events is described using a prefix operator "$\rightarrow$". The expression $a \rightarrow P$ describes the process that engages in the event a and then behaves as process P. The environment can decide between two processes using the choice operator "$\Box$". $P \Box Q$ represents the process that offers the choice to the environment between behaving as process P or as process Q. There is also a nondeterministic choice operator "$\sqcap$": $P \sqcap Q$ represents the process that internally chooses between behaving as P or Q, without any environment control. Another operator in CSP in parallel composition of two processes. P and Q interact by synchronising over common events in $\alpha P \cap \alpha Q$, while events not in $\alpha P \cap \alpha Q$ can occur independently. The parallel composition of two processes P and Q is shown by expression $P \parallel Q$. An event common to both P and Q, becomes a single event in $P \parallel Q$ and can be offered by P and Q only when both P and Q are prepared to offer it. The interleaving operator represents completely independent concurrent activity. The process $P ||| Q$ behaves as both P and Q simultaneously. The hiding operator provides a way to abstract processes, by making some events unobservable. A trivial example of hiding is $(a \rightarrow P) \setminus \{a\}$ which, assuming that the event a doesn't appear in P.

2.3.5 Action Systems

Action systems [39, 40] provide a method to program distributed systems in a way that the overall behavior of the system is emphasized. In this manner, the behavior of the system is described in terms of the possible interactions, called actions, that the processes can engage in, rather than in terms of a sequential execution of the processes.

The behavior of a distributed system was usually described in process-based manner. Each process interacts with other processes by sending and receiving messages in a execution of a sequential piece of program. In a process-based approach it is difficult to get a picture of the overall behavior of the system. Whereas action system is a state-based description of a distributed system that concentrates on the overall behavior of the system by defining states and actions, rather than sequential processes.

2.3.6 A Comparison

This section compares the mentioned formalisms, based on comparisons which have been presented in [34], [41] and [42].

Z, VDM and B are model-based methods. In model-based methods, the states and operations are explicitly modelled and the operations transform the system from a state to another state. In model-based approach there is no explicit representation of concurrency. Therefore Z, VDM and B do not support representation and reasoning of concurrency.

Temporal Logic is a logic-based formalism. In Logic-based approach, logics are used to describe system desired properties, including low-level specification, temporal and probabilistic behaviors. Temporal logic and CSP can handle concurrency.

Another common classification of formal approaches from behavioral point is to partition them to state-based and event-based [43]. From this point of view, Z, VDM, B and Temporal logic are state-based, whereas CSP is a event-based formalism. Considering state-based, there is explicit definition of states. Operations have an effect of transforming the system from a state to another state. Whereas in event-based, the focus is on identifying events of the system and then describing in what order these events are allowed to happen.

2.4 Event-B

2.4.1 The Event-B Definition

Event-B [4, 14, 26, 44] is a formal method for specifying, modelling and reasoning about systems. Event-B is an evolution of B-Method [9] developed by Jean-Raymond Abrial. Hallerstede states that Event-B has evolved from B-Method and Action Systems [39, 40]. On the one hand Event-B is a simplification as well as an evolution of B-Method; on the other hand Event-B is influenced by the action systems approach. It has a same structure as an action system which describes the behavior of a reactive system in terms of the guarded actions that can take place during its execution.

Event-B is different than the B-Method in some aspects. The B-Method is organized in a way that is suitable for the development of non-concurrent programs, whereas Event-B is geared toward the development of systems including reactive and concurrent systems.

Event-B is used in modelling and verifying. The modelling notation has been designed to be simple and easily teachable, which is based on set theory and logics. Building a model in Event-B starts with a very abstract level, and continues in different abstraction levels by use of refinement, which will be explained in Section 2.4.3. Event-B use mathematical proof to verify consistency between refinement levels. Association of proof obligations in Event-B permits us to reason about it, see Section 2.4.4. Rodin is a tool platform for modelling and proving in Event-B, will be explained in Section 2.4.5.

2.4.2 Event-B Structure and Notation

A model in Event-B [4, 13, 14] consists of Contexts and Machines. In other words, a model is made of several components of these two types.

Contexts contain the static part (types and constants) of a model while Machines contain the dynamic part (variables and events). Contexts provide axiomatic properties of an Event-B model, whereas Machines provide behavioural properties of an Event-B model. Items of machines and contexts are called modelling elements presented in this section.

There are various relationships between contexts and machines. A context can be "extended" by other contexts and "referenced" or "seen" by machines. A Machine can be "refined" by other machines and can reference to contexts as its static part. Refinement is described more in Section 2.4.3. Machine and context relationship are illustrated in Figure 2.1.

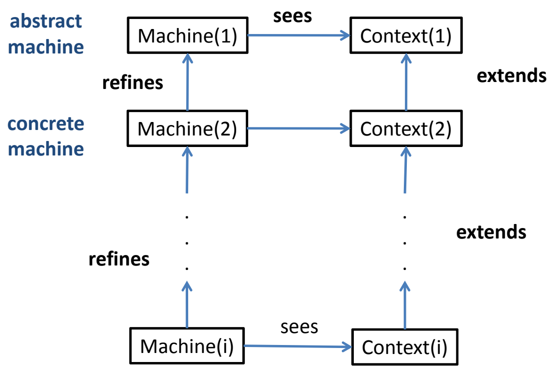


Figure 2.1: Machine and Context Relationships

Recall from Section 2.2.3, from a given machine, $Machine1$ in this case, a new machine, $Machine2$, can be built as a refinement of $Machine1$. In this case, $Machine1$ is called an abstraction of $Machine2$, and $Machine2$ will said to be a concrete version of $Machine1$.

2.4.2.1 Context Structure

The modelling elements of a context [4, 13, 14] are from four types: sets, constants, axioms and theorems. It is illustrated in Figure 2.2. Axioms are various predicate describe the property of sets, constants, theorems. A context can extend more than one context, and also can be seen by several machines in a direct or indirect way. By indirect, we mean that a context may be referenced by a machine whose abstract machine sees that context. Clause “Theorems” lists the various theorems which have to be proved within the context.

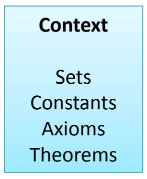


Figure 2.2: Structure of a Context

2.4.2.2 Machine Structure

A Machine [4, 13, 14] consists of variables, invariants, events, theorems and variants, illustrated in Figure 2.3. Variables, v, define the state of a model. Invariants, $I(v)$, constrain variables, and are supposed to hold whenever variables are changed by an event. New events can be defined in a concrete machine, will be described more in Section 2.4.3. In order to prove that they do not take control forever, a new event must decrease a natural number expression called variant [45].

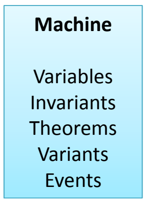


Figure 2.3: Structure of a Machine

2.4.2.3 Events

In Event-B, state of a model is changed by means of event execution. Each event is composed of a name, a set of guards $G(t, v)$ and some actions $S(t, v)$, where t are parameters of the event and v is state of the system which is defined by variables. All events are atomic and can be executed only when their guards hold. When the guards of several events hold at the same time, then only one of those events is chosen nondeterministically to be executed. An event can appear in three forms presented in Table 2.1. In the simplest term, an event contains only some actions, in second form it can composed of guards and actions without parameters, and finally in third form an event has guards, actions and some parameters.

Three Possible Forms of an Event
E = begin S(v) end
E = when G(v) then S(v) end
E = any t when G(t,v) then S(t,v) end

Table 2.1: Event Forms

The action of an event can have a few forms of assignments [13], illustrated in Table 2.2. Here x is a variable, $E(t, v)$ is an expressions, and $P(t, v, x')$ is a predicate. The first assignment form is deterministic. In the second row, the assignment is nondeterministic (for instance, assign a value within a non-empty set). The third row assigns a value to x according to the predicate defined and it is also considered nondeterministic.

Type	**Generalized Substitution**
Deterministic	x := E(t, v)
Nondeterministic	$x :\in E(t, v)$
Nondeterministic	$x :\mid P(t, v, x')$

Table 2.2: Action Forms

2.4.3 Refinement in Event-B

In the Event-B development, rather than having a single large model, it is encouraged to construct the system in a series of successive layers, starting with an abstract representation of the system. The abstract model should provide a simple view of the system, focusing on the main purpose and key features of the system. The details of how the purpose is achieved are ignored in abstraction. Details of functionality of the system are added gradually to the abstract model in a stepwise manner. This process is called refinement.

In the Event-B modelling, we use proof to verify the consistency of a refinement. The semantic of some refinement proof obligations are described in Section 2.4.4.

Types of Refinement in Event-B [8, 13, 46]**:**

Refining an Event-B model can consist of Context extension and Machine refinement. Considering context extension, it is possible to add new sets, constants and properties while retaining the old ones.

Refinement in Event-B has different views or classification. From Event-B notation point of view, refinement of a machine consists of:

1. Refining existing events:

(a) Add new parameters, guards and actions to the existing abstract event: in this case the resulting concrete event is labeled as *extended*. In an *extended* event, the existing parameters, guards and actions can not be modified.

(b) Modify parameters, guards and actions of the existing abstract event: in this case the resulting concrete event is labeled as $non - extended$ $(refine)$. Adding new parameters, guards and actions are allowed too.

In both types the guards of the concrete event must be proved to be stronger than its abstraction (guard strengthening).

2. Add new events
   The new event refines a dummy event in the abstraction which does nothing (*skip*). The new event does not diverge. It means that it should not take control forever. The new event can be labeled as:

   - Convergent: Each convergent event requires a variant ro ensure non-divergence.
   - Anticipated: Events that will be introduced in a future refinement but are declared in anticipation.
   - Ordinary: None of the others and the most commonly used.

3. Add new variables and invariants:
   Introducing new variables usually results in (2) or (1.a) types of refinement. Sometimes abstract variables can be replaced by new concrete variables. In this case the refinement can result in (1.b). Variable replacement is called data-refinement. Sometimes variable replacement results in redundant variables which can be removed.

   A gluing invariant connects the abstract variables to the concrete variables. In other words, it glues the state of the concrete model to that of its abstraction. The invariant of the concrete model including gluing invariants should be preserved for every event.

Each abstract event should be refined by at least one concrete event. One abstract event can be refined by more than one concrete event. It is called event splitting, examples are presented in the case study developments. Also one concrete event can refine more than one abstract event. It is called event merging.

Refinement is the process of enriching or modifying the abstract model in order to introduce new functionality or add details of current functionality. From another view, there are two forms of refinement:

- Vertical Refinement (Structural Refinement): In this from, design details of current functionalities are added. This form of refinement may involve data-refinement (3) and modifying abstract events (1.b). In refinement level the modified events are labeled as non-extended events.

- Horizontal Refinement (Superposition Refinement or Feature Augmentation): New functionalities of the system, which are not addressed in abstract level, are introduced. Usually it can be achieved by introducing new events (2), new variables (3) or extending abstract events (1.a). In refinement level these concrete events are labeled as extended events.

2.4.4 Proof Obligations

There are different proof obligations which are generated by the Event-B tool, Rodin, during development of a system using Event-B [47, 48]. Here we describe some of those which are most important. Considering Figure 2.4, machine $M2$ refines machine $M1$. Both of them see context Ctx. $M2$ contains two events, $evt3$ as a new event and $evt2$ as a refining event. Also it contains some gluing invariants.

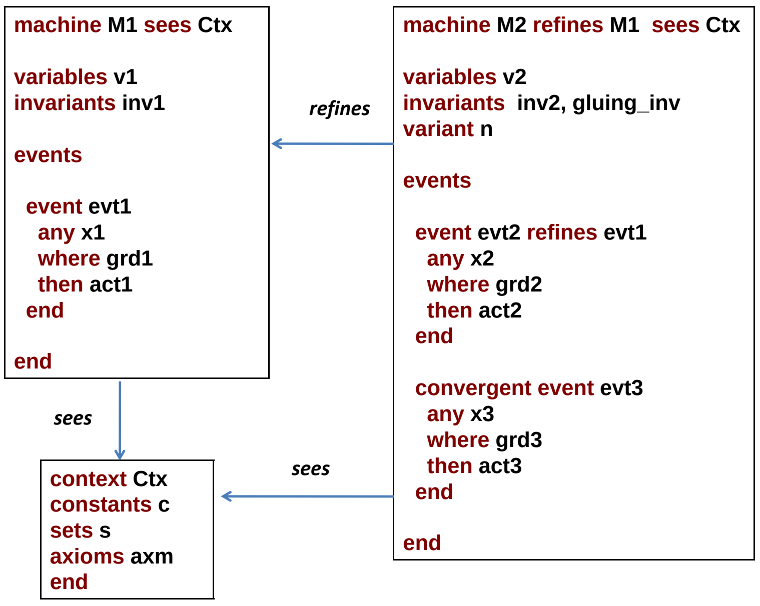


Figure 2.4: An Event-B Model (Context Ctx, Abstract Machine $M1$, Concrete Machine $M2$)

Table 2.3 contains a list of important proof obligation in Event-B modelling.

The last four proof obligations are refinement proof obligations and the last two are the proof obligation generated for defining new events in concrete machine in a new refinement level. Here are some explanation for each mentioned proof obligations:

- **Well-definedness (WD)**: Ensure that a potential ill-defined axiom, theorem, invariant, guard, action, variant is indeed well-defined. For instance for having cardinality of a set, $card(S)$ it should be proved that the set, S, in finite.

Well-definedness	x / WD	x is the name of axiom, theorem, invariant, guard, action, variant
Invariant Preservation	evt / inv / INV	evt is the event name, inv is the invariant name
Feasibility of a nondeterministic event action	evt / act / FIS	evt is the event name, act is the action name
Guard Strengthening	evt / grd / GRD	evt is the concrete event name, grd is the abstract guard name
Action Simulation	evt / act / SIM	evt is the concrete event name, act is the abstract action name
Natural number for a numeric Variant	evt / NAT	evt is the new event name
Decreasing of Variant	evt / VAR	evt is the new event name

Table 2.3: Proof Obligations in Event-B

- **Invariant Preservation (INT)**: Ensure that each invariant is preserved by each event. For instance in Figure 2.4, one of generated proof obligation is $evt1/inv1/$INV, ensuring that $inv1$ is preserved by event $evt1$ in machine $M1$.
- **Feasibility (FIS)**: Ensure that each nondeterministic action is feasible. In Figure 2.4, for event $evt1$ in machine $M1$, this proof obligation is given: $evt1$ / $act1$ / FIS; it means there should exist values for variable $v1$ such that the assignment $act1$ is feasible.
- **Guard Strengthening (GRD)**: Ensure that each abstract guard is no stronger than the concrete ones in the refining event. As a result, when a concrete event is enabled the corresponding abstract one is also enabled. For instance for the model in Figure 2.4, $evt2$ / $grd1$ / GRD ensure that abstract guard $grd1$ is weaker than the guards of the concrete event $evt2$.
- **Simulation (SIM)**: Ensure that each action in a concrete event simulates the corresponding abstract action. When a concrete event executes, the corresponding abstract event is not contradicted. In Figure 2.4 the simulation proof is $evt2$ / $act1$ / SIM.
- **Numeric Variant (NAT)**: Ensures that under the guards of each convergent event a proposed numeric variant is indeed a natural number. $evt3$ / NAT is the proof obligation generated for the model of Figure 2.4.
- **Decreasing of Variant (VAR)**: Ensures that each convergent event decreases the proposed numeric variant. As a consequence the new event does not take control forever. $evt3$ / VAR in Figure 2.4 ensures that event $evt3$ does not take control forever.

2.4.5 Rodin as an Event-B Tool

Rodin [3, 13, 16, 49] is a software tool for formal modelling and proving in Event-B. Rodin has an open platform, and is an extensible and adaptable modelling tool. Butler and Hallerstede state that “the aim with Rodin open tools kernel is to greatly extend the state of the art in formal methods tools, allowing multiple parties to integrate their tools as plug-ins to support rigorous development methods” [16]. They believe that this is likely to have a significant impact on future research in formal methods tools and will encourage greater industrial uptake of these tools. The ProB animator [18], UML-B [19], B2LaTeX [20] and model decomposition [50] are good examples of plug-in developments; ProB is a model checker which checks the consistency of B machines; UML-B maps a graphical formal modelling notation to the Event-B language; B2LaTex is used for translating Event-B models into LaTeX documents; and model decomposition which allows to decomposed a model into sub-models, it will be explained in Section 2.5.

Like programming tools, Rodin carries out many tasks automatically, and provides fast feedback in the case of changes in a model text. Instead of compiling automatically in programming tools, Rodin generates proof obligations and discharges trivial ones automatically; and instead of running a program, Rodin is used to reason about a model.

Rodin is an integration between modelling and proving. As described in previous sections, proving is an essential part of modelling. The proof obligations define what is to be proved for an Event-B model. Discharging all proof obligations of a model shows that all model properties are consistent. Sometimes a model can be changed using proofs errors. When a proof obligation can not be charged, it shows that there is an inconsistency in the model. This leads us to learn more about the system in order to change the model in a consistence way. Therefore during modelling we can learn about system and we can eliminate misunderstandings and learn new requirements by proving the failed proof obligations.

2.4.6 A Comparison Between Event-B and Other Formal Methods

Classical B, Z and VDM have a one-to-one operation refinement, meaning that one abstract operation is refined by only one concrete operation. There is no feature of introducing new events in these formal methods. Whereas Event-B is flexible as it inherits a refinement property from action systems. It is possible to introduce new events during the stepwise refinement steps. Also event merging and event splitting are provided in Event-B refinement.

Although Event-B is an extension of Classical B, there are some differences between them:

- The model structure is different. In Event-B, the context as the static part of the system and the machine as the dynamic part of the system are explicitly separated. Whereas in the B-Method a machine contains both parts.

- In the B-Method, operations are called by other operations. While in Event-B the enabled events are continually executed in a nondeterministic manner. Since in Event-B, we are modelling reactive systems, the events are not called and the model controls its behaviour by nondeterministically choosing the enabled events.

- A B-Method operation contains pre-conditions which express formally what is to be proved when the operation is invoked [51]. The caller of an operation is responsible to make sure that pre-conditions of the called operation are hold before calling it. The called operation can assume that its pre-conditions hold, and it does not need to check its pre-conditions.

  Whereas an Event-B event contain guards. An event can be executed only when its guards hold. In Event-B, enabled events are nondeterministically chosen to execute.

- Refinement is more general in Event-B. Introducing new events is an important ability in Event-B refinement.

2.5 Event-B Model Decomposition

2.5.1 Overview

Model decomposition predated Event-B and is found in action systems [40]. In developing a model in Event-B, one of the key features is introducing new events and new state variables during refinement. As a consequence it usually ends up with many events and many variables in the last refinement level. Dealing with a large number of events and variables can be complex, particulary in some points we need to refine just a few variables and events and so other variables and events play no role in the refinement [52].

Model decomposition in Event-B [53], is intended to decrease the complexity and increase the modularity of a large Event-B model, especially after several layers of refinements. The idea of model decomposition is cutting a huge model into smaller pieces called sub-models, which can more easily deal with than the first model, and each of them can be refined separately.

Distribution of proof obligations into several sub-models is one of the major results of model decomposition, which is expected to be easier to discharge. The further refinements of independent sub-models in parallel is a benefit of model decomposition. Moreover the possibility of team development after model decomposition seems useful in developing a big system.

An overview of the model decomposition in Event-B is illustrated in Figure 2.5. As presented the model becomes bigger during refinement layers and with decomposition it is split into smaller sub-models, then each sub-model can be refined independently.

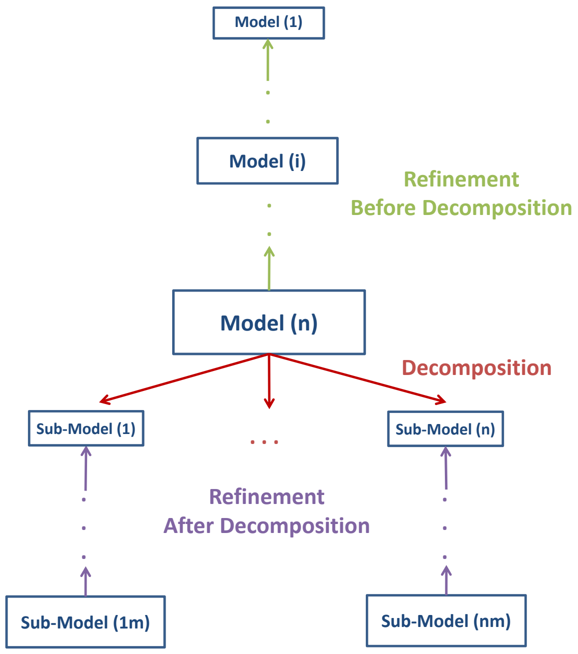


Figure 2.5: Model Decomposition in Event-B

2.5.2 Decomposition Styles

There are two ways of decomposing an Event-B model, shared variable and shared event [54]. The shared event approach seems particularly suitable for message-passing distributed programs, whereas the shared variable approach seems more suitable for concurrent programs [55]. In shared event model decomposition, variables are partitioned among the sub-models, whereas in shared variable approach, events are partitioned among the sub-models. Details are explained in the next section.

A model decomposition plug-in [21, 50, 56] in Rodin platform provides tool support for both styles of model decomposition.

Later in Chapter 7.3, we will see how model decomposition approach in developing the Event-B model of a complex system is useful together with using the atomicity decomposition approach which is the main contribution of this thesis.

2.5.2.1 Shared Variable Style

Shared variable decomposition illustrated in Figure 2.6 is proposed by Abrial [52], Metayar [57] and Hallerstede [58]. Machine *M* is decomposed into machine *M1* and *M2*. The solid lines show relationships between events and variables in each machine.

The shared variable decomposition does not permit events sharing and a variable can be split into different sub-models, this variable is called a shared variable. First the events of *M* are partitioned among *M1* and *M2*. Then the variables of *M* are distributed according to the event partition. *v1* and *v3* are private variables, since they are accessed by events of only one sub-model, *e1* in *M1* and *e4* in *M2* respectively. *v2* is a shared variable which is accessed by event *e2* in *M1* and *e3* in *M2*. External event of *e2_ext* is built in *M2*, since *e2* modifies the shared variable *v2* in *M1*. The invariant distribution is done according to variable distribution. An invariant belongs to a sub-model if all variables used in that invariant belong to that sub-model.

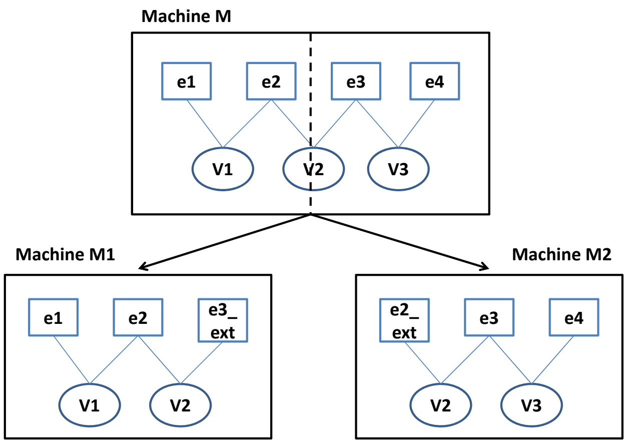


Figure 2.6: Shared Variable Decomposition

2.5.2.2 Shared Event Style

Figure 2.7 illustrates shared event decomposition proposed by Butler [59]. Variables of the machine *M* are partition among the sub-models, *M1* and *M2*. After the variable partition it is necessary to split the events according to the variable partition. Events using variables allocated to different sub-models, *e2* using *v1* from *M1* and *v2* from *M2*, are called shared events and must be split. Part of the shared event which is corresponding to each variable, *e2_1* and *e2_2*, is used to build sub-models events. Invariant distribution is similar to shared variable decomposition.

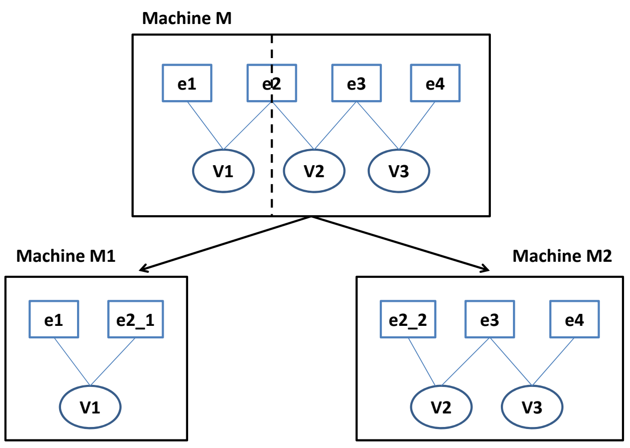


Figure 2.7: Shared Event Decomposition

Chapter 3

Atomicity Decomposition Part 1 - Overview and Background

3.1 Introduction

The atomicity decomposition approach was first introduced by Butler in [24]. In this chapter we present the atomicity decomposition approach from [24], in Section 3.2. As mentioned in Section 1.2, a major contribution of atomicity decomposition approach is structuring refinement in Event-B. To highlight this contribution, Section 3.3 outlines the role of atomicity decomposition diagrams in structuring refinement in Event-B. It is followed by two examples of the atomicity decomposition application from [24], in Section 3.4.

3.2 Overview of Atomicity Decomposition Diagram in Event-B

Although the refinement approach in Event-B, as explained in Sections 2.2.3 and 2.4.3, provides a flexible approach to modelling, it does not have the ability to show the relationship between one abstract event and the corresponding concrete events. The atomicity decomposition approach is intended to make the relationships between abstract and concrete events clearer and easier to manage than simply using the standard Event-B refinement technique. In this approach course-grained atomicity can be refined to more fine-grained atomicity.

The tree structure notation of the atomicity decomposition approach is first introduced by Butler in [24]. The diagrammatic notation is based on JSD structure diagrams by Jackson [7]. In [24] the atomicity decomposition diagram is presented in two examples containing a parallel execution of an event. Before introducing the parallel notation, we generate a simple view of the atomicity decomposition diagram in order to explain the basic features. It is shown in Figure 3.1. The features explained here are from [24].

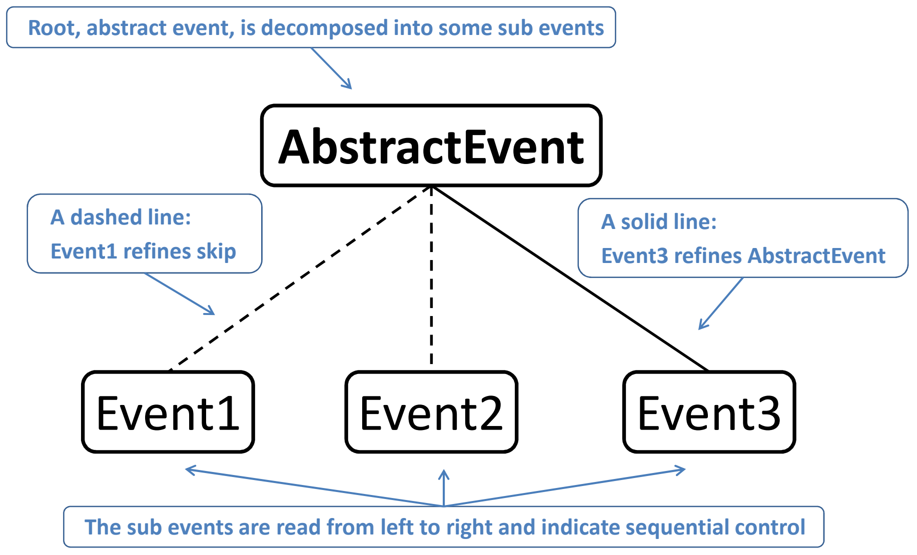

Root, abstract event, is decomposed into some sub events
AbstractEvent
A dashed line:
Event1 refines skip
A solid line:
Event3 refines AbstractEvent
Event1
Event2
Event3
The sub events are read from left to right and indicate sequential control

Figure 3.1: Atomicity Decomposition Diagram

The abstract atomic event, *AbstractEvent*, appears in the root node. The diagram shows how the root is decomposed into some sub-events in the refinement model. The number of sub-events can be one or more. In this case we consider three sub-events to explain the features of the diagram. An important feature of diagram, in common with JSD structure diagrams, is that the sub-events are read from left to right and indicate sequential control from left to right. This means that our diagram indicates that the abstract event is realised in the refinement by firstly executing *Event1*, then executing *Event2* and then executing *Event3*.

Sub-events are treated in two ways, one refines abstract event and the others are viewed as hidden events in the abstract model which refine *skip* in the refinement model. So another important feature is types of lines, solid line and dashed line. The sub-events corresponding to dashed lines, *Event1*, *Event2*, are new events which refine *skip* in the abstract model. The sub-event with a solid line, *Event3*, is a refining event which must be proven to refine the abstract event, *AbstractEvent*. A new event introduced in the refinement model which refines *skip*, can be viewed as a hidden event in the abstract model. This kind of event is not visible to the environment of a system in the abstract model, and therefore they are outside the control of the environment [24].

In this case, *Event1* should execute before *Event2*. Also *Event2* should execute before *Event3*. This is done by some control variables in the refinement model. We will see more about control variables later in this chapter.

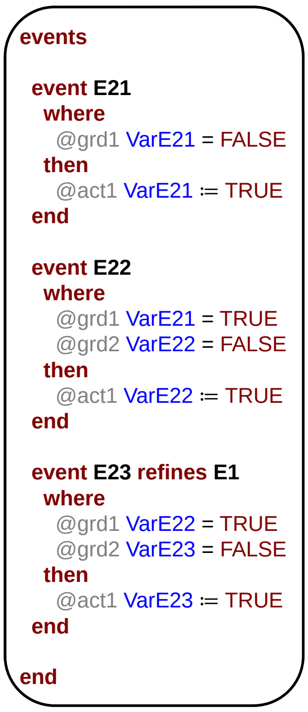
With the aim of making the point more clear, the possible execution traces of the model, called event trace [24], are presented here.

The execution trace of the abstract model contains a single event and is represented as $< AbstractEvent >$. The execution trace of the refinement model events, *Event1*, *Event2* and *Event3*, is $<$ *Event1, Event2, Event3* $>$.

3.3 Event-B Refinement and Atomicity Decomposition Diagrams

One of the important motivations of the atomicity decomposition approach is that it explicitly shows the event ordering and the relationship between an abstract event and the corresponding concrete events, whereas the Event-B text is not able to explicitly show these properties. This can be seen by comparing Figure 3.2 and Figure 3.3.

Assume Event *E21* should execute before event *E22*. And event *E22* should execute before event *E23*. Considering Figure 3.2, the ordering between these events is *implicit*. Whereas the atomicity decomposition diagram in Figure 3.3, *explicitly* shows the event ordering by a sequence execution of events from left to right.



```
events

 event E21
  where
   @grd1 VarE21 = FALSE
  then
   @act1 VarE21 ≔ TRUE
 end

 event E22
  where
   @grd1 VarE21 = TRUE
   @grd2 VarE22 = FALSE
  then
   @act1 VarE22 ≔ TRUE
 end

 event E23 refines E1
  where
   @grd1 VarE22 = TRUE
   @grd2 VarE23 = FALSE
  then
   @act1 VarE23 ≔ TRUE
 end

end
```

Figure 3.2: Event-B Model of Atomicity Decomposition Diagram in Figure 3.3

Considering Figure 3.2, the ordering is implicitly specified by some control variables in the Event-B model. *VarE21*, *VarE22* and *VarE23* are boolean control variables which are initialised to *FALSE*. First event *E21* executes and enables *VarE21* variable. Event

E22 is guarded by *VarE21* variable, *grd1*. Therefore event *E22* can execute only after event *E21* executes. Also event *E23* is guarded by *VarE22*, *grd1*. So event *E23* can execute only after event *E22* executes.

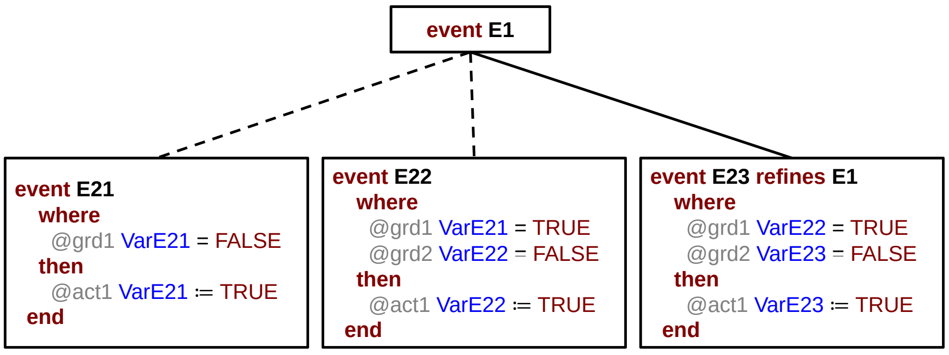


Figure 3.3: Atomicity Decomposition Diagram of Event-B Model in Figure 3.2

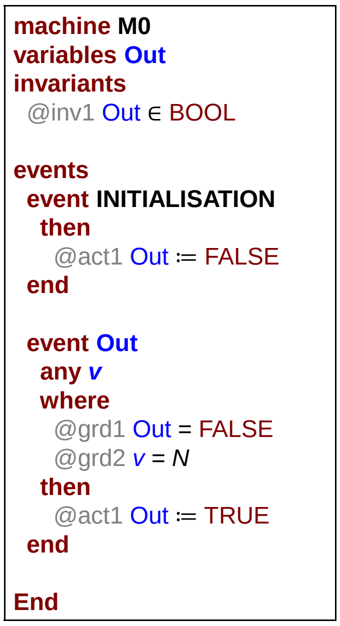
Moreover the diagram explicitly illustrates our intention that the effect achieved by event *E1* at the abstract model is realized at the refinement model by execution of event *E21* followed by event *E22* followed by event *E23*, Figure 3.3. Whereas in the standard Event-B model, Figure 3.2, events *E21* and *E22* are refinements of *skip* and there is no explicit connection to abstract event *E1*. Technically, event *E23* is the only event that refines event *E1* but the diagram indicates that we break the atomicity of abstract event *E1* into three sub-events *E21*, *E2* and *E23*.

3.4 Examples of Application

With the aim of making the application of atomicity decomposition diagrams more clear, two examples from [24] are presented here.

Assume the abstract machine contains a single event *Out*, that simply outputs N exactly for one time. Considering Figure 3.4, there is only one boolean control variable in the machine, called *Out*, which initialised to false. *Out* event can execute only when it has not executed before, *grd1*. In execution it disabled itself, *act1*. The output value is represented in the parameter v, *grd2*.

The output is produced in an atomic event in the abstract machine. We wish to refine the abstract machine by a machine modelling a concurrent accumulation of the output value before outputting it. The refinement structure is presented in an atomicity decomposition diagram in Figure 3.5. The diagram shows that we break the atomicity of abstract *Out* event, to three sub-events. This means that the abstract *Out* event is realised in the refinement by firstly executing the initialisation, then executing the *Increase* event in parallel and then executing *Out* event. The parallel execution here



```
machine M0
variables Out
invariants
 @inv1 Out ∈ BOOL

events
 event INITIALISATION
  then
   @act1 Out ≔ FALSE
 end

 event Out
  any v
  where
   @grd1 Out = FALSE
   @grd2 v = N
  then
   @act1 Out ≔ TRUE
 end

End
```

Figure 3.4: Abstract Model of an Outputting System

is illustrated with a circle containing "all" and name of a parameter. We call it all-replicator, since it replicates the corresponding sub-events with a new parameter, p, and *Increase* event needs to executes for all instances of parameter p before *Out* event execution. Figure 3.5 is slightly different to what Butler presented in [24]. Butler illustrates the parallel execution with a circle containing "par(p)". Since we have improved the atomicity decomposition notations, which will be presented in Chapter 4, we found it more understandable if the diagram presented here is compatible with the improvement of notations in Chapter 4.

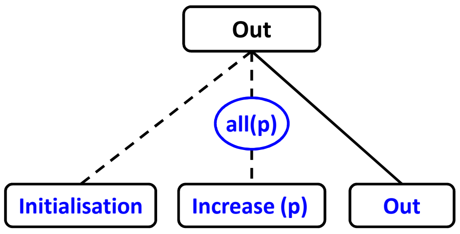


Figure 3.5: Atomicity Decomposition Diagram of an Outputting System

The Event-B model of the refinement machine is presented in Figure 3.6. Each parallel execution of *Increase* event, increments the variable x exactly once. When all N sub-events have incremented x, the value of x is output with execution of *Out* event.

Consider the case where we have two subprocesses, $PROC = \{p1, p2\}$, and $N = 2$. The event traces of the refinement model are as below:

$<$ *Initialisation, Increase(p1), Increase(p2), Out(2)* $>$
$<$ *Initialisation, Increase(p2), Increase(p1), Out(2)* $>$

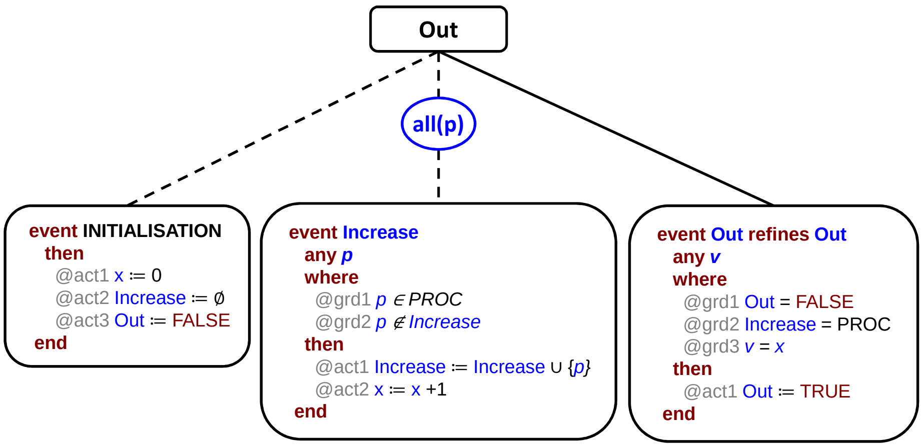


Figure 3.6: Event-B Refinement of an Outputting System

The two possible interleaving of *Increase(p1)* and *Increase(p2)*, represented by two events traces, model their concurrent execution.

As presented in the first example, *Out* event needs to execute only for one time. Therefore we defined the control variable, *Out*, as a boolean variable, which is disabled in the body of *Out* event after the first execution. Whereas sometimes we wish to model a sequence of events which can execute more than one time for different instances of one or more parameters. Second example presents this case. Later in Chapter 4, first case is called Single Instance (SI) and second case is called Multiple Instance (MI). The type of control variables are different in SI and MI. Considering SI, as seen in first example, control variables are boolean, whereas in the MI case, control variables are sets. Having set type enables multiple instances of an event and event interleaving.

As the second example, consider the atomicity decomposition diagram of a file write system in Figure 3.7. The atomicity of the abstract *Write* event is break to three sub-events in the refinement machine, in order to model the writing of individual pages, *PageWrite* event. The writing of the entire file is no longer atomic. The writing of a file is initiated by *StartWrite* event and ended by *EndWrite* event. Multiple file writes are allowed to be taking place simultaneously in an interleaved fashion. This is indicated by a parameter provided in abstract *Write* event, *f*, and inherited with all sub-events. Also in the refinement model, the pages of an individual file *f* can be written in parallel hence an all-replicator over *PageWrite* event replicates its parameter with *p*.

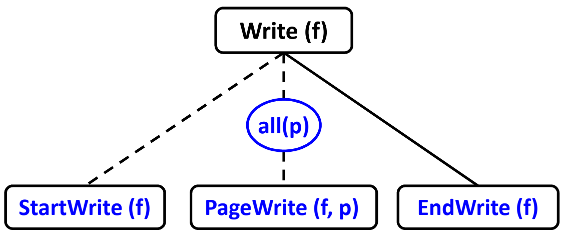


Figure 3.7: Atomicity Decomposition Diagram of File Write

The control variables are sets and the invariants to model event sequencing implied in Figure 3.7 are presented in Figure 3.8. *StartWrite* is a subset of *FILE*, because it is bounded by parameter *f*, (*inv1*). *PageWrite* is a subset of *FILE* × *PAGE*, because it is bounded by parameter *f* and all-replicator parameter *p*, (*inv2*). If a page has been written for a file, then *StartWrite* will already have executed for the file, (*inv3*).

```
invariants
  @inv1 StartWrite ⊆ FILE
  @inv2 PageWrite ⊆ FILE × PAGE
  @inv3 dom(PageWrite) ⊆ StartWrite
```

Figure 3.8: Invariants of File Write Refinement Model

The Event-B model of *StartWrite* and *PageWrite* events are presented in Figure 3.9. The event sequencing is managed with some guards. *PageWrite* is guarded with *StartWrite*, *grd1*, which indicates ordering between *StartWrite* event and each *PageWrite* event.

```
event StartWrite
   any f
   where
     @grd1 f ∈ file
     @grd2 f ∉ StartWrite
   then
     @act1 StartWrite ≔ StartWrite ∪ {f}
 end

event PageWrite
   any f p
   where
     @grd1 f ∈ StartWrite
     @grd2 f ↦ p ∉ PageWrite
   then
     @act1 PageWrite ≔ PageWrite ∪ { f ↦ p }
 end
```

Figure 3.9: Event-B Model of File Write

The accurate explanation of Event-B model derived from atomicity decomposition diagrams are presented in a pattern based style in Chapter 4. In this section, by using some examples, we tries to make the overall benefits of the atomicity decomposition approach more clear.

3.5 Conclusion

This chapter introduced the atomicity decomposition diagram notation. We have outlined how atomicity decomposition diagrams help to structure refinement in Event-B by showing the relationships between events of different refinement levels and by providing an explicit visual view of the ordering between events. Each node presents one event. The root node contains the name of an abstract event and the child nodes contain the names of concrete sub-events. A refining relationship between an abstract event and a concrete event is indicated with a solid line in the diagram between these two event nodes, and a non-refining relationship is indicated with a dashed line. The ordering between events is indicated with a sequence from left to right in the diagram.

To make the application of atomicity decomposition diagrams more clear and to highlight the benefits of atomicity decomposition diagrams in structuring refinement, two examples have been outlined. First example covers the case when a single instance (SI) of event executions is need, whereas the second one shows the multiple instance (MI) case.

This chapter presented background material required to understand the atomicity decomposition patterns in Chapter 4 and description of the atomicity decomposition language in Chapter 5.

Chapter 4

Atomicity Decomposition Part 2 - Patterns and Features

4.1 Introduction

The features of the atomicity decomposition approach in [24] are introduced in Chapter 3. Using these features we have developed two case studies. These developments helped us to improve and expand the atomicity decomposition approach by discovering new constructors and features. This chapter presents the constructor patterns and features in Section 4.2 and Section 4.3 respectively. Each pattern outlines the intention and diagrammatic notation of a decomposing constructor and the way that it is encoded in the Event-B model. The related works are compared with the atomicity decomposition approach in Section 4.5.

More formal and general descriptions of the atomicity decomposition semantic and translation rules to the Event-B are presented in Chapter 5. This chapter helps to understand the contents of Chapter 5.

4.2 Atomicity Decomposition Diagram Patterns

4.2.1 Introduction

This section presents the atomicity decomposition constructors in a pattern-based style. Each pattern outlines one constructor in one level of refinement. The combination of different patterns in more than one level of refinement will be presented via formal description of the atomicity decomposition language and translation rules in Chapter 5.

In the atomicity decomposition approach, we found some common and reusable constructors (as solutions) to some common intentions (as problems). These recurring problem-solution pairings motivated us to use a pattern-based approach to introduce the atomicity decomposition constructors. Moreover organizing the problems and solutions in a pattern-based approach is easy to read, understand and apply.

In total, eight constructor patterns have been delineated. The constructor patterns are divided to four distinct groups:

- Sequence pattern, Section 4.2.2.
- Loop pattern, Section 4.2.3.
- Logical constructor patterns: and-constructor, Section 4.2.4, or-constructor, Section 4.2.5, xor-constructor, Section 4.2.6.
- Replicator patterns: all-replicator, Section 4.2.7, some-replicator, Section 4.2.8, one-replicator, Section 4.2.9.

The logical constructors, including the and-constructor, the or-constructor and the xor-constructor, introduce logical relations between two or more sub-events.

Each replicator constructor, including the all-replicator, the some-replicator and the one-replicator, introduces a new parameter to its related sub-event and replicates the dimension of the related sub-event.

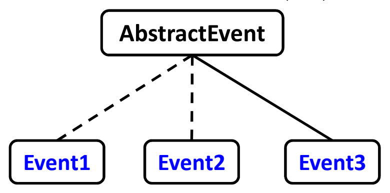
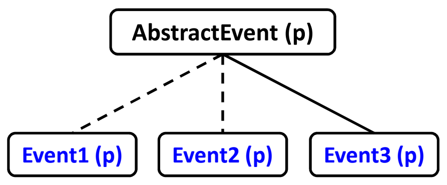
The sequence pattern and the all-replicator pattern have been introduced in [24]. The examples of these two constructors from [24] have been presented in Section 3.4. Here we present them in a way that follows the pattern based style. The other constructs and corresponding Event-B models are derived from developing our case studies. The case study developments are presented in Chapter 7.

4.2.2 Sequence Pattern

Each pattern is presented in a table. The sequence pattern is presented in Table 4.1. Each pattern table includes the name of the pattern in the first row, followed by a diagrammatic representation of the atomicity decomposition diagram of the pattern for single instance execution (SI) on the left and multiple instances execution (MI) on the right. It is followed by the Event-B model generated from the atomicity decomposition diagrams. The Event-B model contains the invariants and events separately for the SI case and the MI case, labeled as "SI/MI Invariants" and "SI/MI Events". The Event-B model shown in the table is part of the model which is generated from atomicity decomposition diagrams, user defined Event-B elements like events can be included in the Event-B model but not in any atomicity decomposition diagram. The table interpretation just described, is used for all patterns' tables.

Name: Sequence

Diagrammatic Representation

Single Instance(SI)	Multiple Instance(MI)
	

Event-B Model

Single Instance(SI) Invariants:

```
invariants
  @inv_Event1_type Event1 ∈ BOOL
  @inv_Event2_seq Event2 = TRUE ⇒ Event1 = TRUE
  @inv_Event3_seq Event3 = TRUE ⇒ Event2 = TRUE
  @inv_Event3_gluing Event3 = AbstractEvent
```

Multiple Instance(MI) Invariants:

```
invariants
  @inv_Event1_type Event1 ⊆ TYPE(p)
  @inv_Event2_seq Event2 ⊆ Event1
  @inv_Event3_seq Event2 ⊆ Event3
  @inv_Event3_gluing Event3  = AbstractEvent
```

Single Instance(SI) Events:

```
event Event1
  where
   @grd Event1 = FALSE
  then
   @act Event1 ≔ TRUE
 end
```

```
event Event2
  where
   @grd_seq Event1 = TRUE
   @grd Event2 = FALSE
  then
   @act Event2 ≔ TRUE
 end
```

```
event Event3 refines AbstractEvent
  where
   @grd_seq Event2 = TRUE
   @grd Event3 = FALSE
  then
   @act Event3 ≔ TRUE
 end
```

Multiple Instance(MI) Events:

```
event Event1
  any p where
   @grd  p ∉ Event1
  then
   @act Event1 ≔ Event1 ∪ { p }
 end
```

```
event Event2
  any p where
   @grd_seq p ∈ Event1
   @grd  p ∉ Event2
  then
   @act Event2 ≔ Event2 ∪ { p }
 end
```

```
event Event3 refines AbstractEvent
  any p where
   @grd_seq p ∈ Event2
   @grd  p ∉ Event3
  then
   @act Event3 ≔ Event3 ∪ { p }
 end
```

Table 4.1: Sequence Pattern

Intention: The atomicity of an abstract event, *AbstractEvent*, is decomposed to sequencing of two or more concrete sub-events. In other words, the behaviour exhibited by an abstract event is realised by the sequential execution of one or more concrete events in the refinement level. Since we are able to describe the features of the sequence pattern by having three sub-events, we minimise the number of sub-events to three, $Event1$, $Event2$ and $Event3$.

Diagrammatic Representation: The name of the abstract event appears in the root node, and sub-events' names appear in leaf nodes in sequence from left to right. A leaf is a node without any child node.

In decomposing the atomicity of an event, two cases are considered. First when a single execution of an event is needed. In this case, there is no control parameter for the event. Moreover control variables are defined with boolean type, since we do not need to record the execution of events for different instances of the parameter(s). This case is called Single Instance (SI). The second case is when multiple instances of an event are needed. It is called Multiple Instances (MI). In this case, there are one or more control parameters for the events. In the diagrammatic representation, control parameter(s) name(s) appear in between parentheses after the event name. In the table, p represents a list of parameters, $p_1, ..., p_n$. We use a set type for control variables. Using sets, enables multiple instances of an event and event interleaving.

Restrictions: One and only one of the leaves in an atomicity decomposition diagram is connected to the root event with a solid line. Other leaves have to connect with dashed lines. This restriction is referred to as the "single solid line" rule in the rest of patterns.

This restriction can raise two questions:

- First, where is the leaf placed with solid line in the sequence of sub-events in the atomicity decomposition of an abstract event?
- Second, why only one leaf with the solid line can be placed in the atomicity decomposition of an abstract event?

The first question is answered in the next two paragraphs. The short answer for the second question is that this restriction is a result of restrictions in the Event-B model. Since there can be only one occurrence of the abstract event in the refinement level, there is only one refining event (leaf with the solid line). The second question is clarified at the end of this section using examples of event traces.

In the Event-B model, the EQL (Equality of preserved variable) proof obligation, $(evt/v/EQL)$, ensures that an abstract variable v is preserved in the concrete event *evt*. It means that the EQL proof obligation does not allow an abstract variable to be changed in a new event which refines *skip*. The abstract variable v can be modified only by a concrete event that refines the abstract event which modifies variable v. Also the SIM (Simulation) proof obligations ensure that each action in a concrete event simulates the corresponding abstract action. It means when a concrete event executes, the corresponding abstract event is not contradicted.

The leaf corresponding to the solid line is encoded to an event which refines the abstract event, appearing as the root node. Considering the limitation which EQL and SIM proof obligations make in the Event-B model, the refining event is the event which simulates the main behaviour of the abstract event by modifying the corresponding abstract variable(s). In our patterns we consider it as the last event, $Event3$.

Event-B Model:

Semantics are given to an atomicity decomposition diagram by generating an Event-B model from it. We now explain how an atomicity decomposition diagram of the sequence pattern is encoded as an Event-B model. The encoded Event-B model for the sequence pattern is presented in Table 4.1.

The middle sub-event in the sequence pattern is replaced by a constructor in the rest of patterns, which are described later. Each constructor can be placed as the first or the last sub-event of the diagram too; the reason that we consider it as the middle sub-event is to show the effect of the previous sub-event (the first sub-event) on the constructor, and the effect of the constructor on the next sub-event (the last sub-event). The sequence pattern is considered as a basic pattern for the rest of atomicity decomposition patterns. Therefore most of the translation rules from the diagram to the Event-B model which are explained in this pattern, are true for the rest of patterns.

For each leaf, a node without any child node, one control variable and one event are generated. The generated event name and variable name are same as the leaf name. Recalling event labeling in Section 2.4.3, all generated new events are labeled as ordinary events. Ordering between leaves is achieved by generating some actions and guards in generated events. The generated event corresponding to the leaf with the solid line refines the abstract event. The leaf with the solid line can have the same name as the abstract event, since it refines the abstract event. In the diagrams of Table 4.1 the rightmost event can have the same name as the abstract event.

Considering the SI case, the boolean control variable's value in the related event, is assigned to TRUE. This assignment enables the next event's guard in sequence. For example, in event $Event1$, variable $Event1$ is assigned to $TRUE$, indicating that event $Event1$ executes. This assignment enables guard ($Event1 = TRUE$) in event $Event2$. We do not need the sequencing guard in the first event, as there is no event before it in sequence. Another guard is generated for each generated event too. This guard indicates that the current event has not executed before, i.e., ($Event1 = FALSE$) in event $Event1$.

In the MI case, each event corresponding to a leaf gives rise to a set control variable whose type is based on the type of the parameter(s) of the leaf. In the table, p represents a list of parameters, $p_1, ..., p_n$, of type $TYPE(p_1) \times ... \times TYPE(p_n)$. When an event executes for a specific value of the instance parameter(s), the value is added to the set control variable in the action of that event. This enables the next event's guard in sequence. For example, in event $Event1$, the parameter value is added to the set variable $Event1$. This action enables the next event's guard, $(p \in Event1)$ in event $Event2$. Another guard in each event checks that the event has not executed before, i.e., $(p \notin Event1)$ in event $Event1$.

For each leaf an invariant is generated. The invariants states the sequencing conditions. For example in the SI case, $(Event2 = TRUE \Rightarrow Event1 = TRUE)$ is a condition to show that $Event1$ should executes before $Event2$. In the MI case, the subset invariant $(Event2 \subseteq Event1)$ shows that for instances of variable $Event2$, event $Event1$ has executed before. For the first leaf, we do not need a sequencing invariant. Instead a typing invariant is generated.

A gluing invariant is generated for a leaf with solid line. Leaf $Event3$ connects to the root node with solid line, so the gluing invariant $(Event3 = \mathit{AbstractEvent})$ is generated.

To make the use of gluing invariant clear, consider a case when machine *M2* refines machine *M1*. Atomicity decomposition diagrams help illustrate the relation between abstract events of *M1* and concrete events of *M2*. Each event E of *M2* corresponding to a leaf with solid line in diagrams, either refines an abstract event A of *M1*, or it is a new event corresponding to a leaf with dashed line refining *skip*. The proof obligations defined for Event-B refinement are based on the following proof rule that makes use of a gluing invariant Inv_Gluing.

- Each *M2.E* refines *M1.A* under Inv_Gluing, if A is defined.
- Each *M2.E* refines *skip* under Inv_Gluing, if E is a new event.

Therefore in order to discharge the refinement proof obligations, some gluing invariants, which define the relationship between abstract variable and concrete variables, are needed.

Event Execution Trace Examples:

Considering the SI case in the sequence pattern, the single event trace of the refinement model is as follow:

< *Event1, Event2, Event3* >

Each event trace represents a record of a possible execution trace of the model. It is instructive to relate the event trace of the refinement model with the event trace of the abstract model. The single event trace of the abstract model is

$< \mathit{AbstractEvent} >$

If we remove *Event1* and *Event2* from the trace of the refinement model, we get the trace of the abstract model (considering *Event3* refines *AbstractEvent*):

$< \mathit{Event1,\ Event2,\ Event3} > \setminus \{\mathit{Event1,\ Event2}\} = < \mathit{Event3} > = < \mathit{AbstractEvent} >$

Removing events from a trace is the standard way of giving a semantic to hidden events [24, 26] and is used, for example, in CSP. By treating *Event1* and *Event2* as hidden events, traces of the refinement model looks like traces of the abstract model. This illustrates a semantics of refinement of Event-B models. Machine *M1* is a refinement of machine *M0* since any trace of *M1* in which the new events are hidden is also a trace of *M0*. In this point the answer for the second question raised in the *Restriction* part can be made clear. If more than one leaf refines the abstract event in the atomicity decomposition of the abstract event, the refinement semantics in Event-B is violated. Because removing hidden events from the refinement trace does not result in the same abstract trace.

As mentioned in the explanation of the Event-B model, using the set type for control variables, enables multiple instances of an event in an event trace. To make this point clear, we provide some examples of event traces for the MI case here. Considering the MI case in the sequence pattern, assume the case where we have two instances of the parameter, (*p1* and *p2*), two examples of possible event traces are as follows :

$< \mathit{Event1(p1),\ Event2(p1),\ Event3(p1),\ Event1(p2),\ Event2(p2),\ Event3(p2)} >$
$< \mathit{Event1(p1),\ Event1(p2),\ Event2(p1),\ Event2(p2),\ Event3(p1),\ Event3(p2)} >$

To clarify the sequencing conditions modelled with subset invariants in the MI case, we explain the sequencing invariant, ($Event2 \subseteq Event1$). This invariant holds in the above two event traces. For example in the second trace, after execution of $Event2(p1)$, set variable $Event2 = \{p1\}$ is a subset of set variable $Event1 = \{p1, p2\}$.

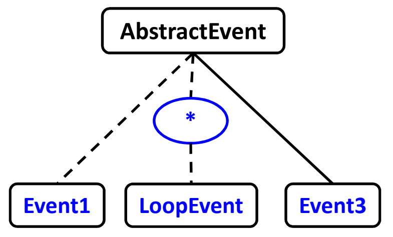
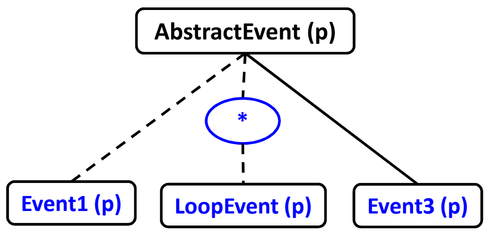
4.2.3 Loop Pattern

The loop pattern is presented in Table 4.2. The table interpretation is the same as what described in term of the sequence pattern table interpretation in Section 4.2.2.

Intention: In the sequence of sub-events, zero or more execution of an event is needed.

Name: Loop

Diagrammatic Representation

Single Instance(SI)	Multiple Instance(MI)
	 AbstractEvent (p) * Event1 (p) LoopEvent (p) Event3 (p)

Event-B Model

Single Instance(SI) Invariants:

```
invariants
  @inv_Event1_type Event1 ∈ BOOL
  @inv_Event3_seq Event3 = TRUE ⇒ Event1 = TRUE
  @inv_Event3_gluing Event3 = AbstractEvent
```

Multiple Instance(MI) Invariants:

```
invariants
@inv_Event1_type Event1 ⊆ TYPE(p)
@inv_Event3_seq Event3 ⊆ Event1
@inv_Event3_gluing Event3  = AbstractEvent
```

Single Instance(SI) Events:

```
event Event1
  where
    @grd Event1 = FALSE
  then
    @act Event1 ≔ TRUE
 end

event LoopEvent
  where
    @grd_seq Event1 = TRUE
    @grd_loop Event3 = FALSE
 end

event Event3 refines AbstractEvent
  where
    @grd_seq Event1 = TRUE
    @grd Event3 = FALSE
  then
    @act Event3 ≔ TRUE
 end
```

Multiple Instance(MI) Events:

```
event Event1
  any p where
    @grd  p ∉  Event1
  then
    @act Event1 ≔ Event1 ∪ { p }
 end

event LoopEvent
  any p
  where
    @grd_seq p ∈  Event1
    @grd_loop p ∉  Event3
 end

 event Event3 refines AbstractEvent
  any p
  where
    @grd_seq p ∈  Event1
    @grd  p ∉  Event3
  then
    @act Event3 ≔ Event3 ∪ { p }
 end
```

Table 4.2: Loop Pattern

Diagrammatic Representation: The loop constructor appears as a circle containing a star. The node connected to the loop, *LoopEvent*, can execute zero or more time after execution of previous sub-event, $Event1$, and before execution of next sub-event, $Event3$, in sequence.

Restrictions: The loop constructor is always connected to the root node with a dashed line. Since the loop event can execute for more than one time, a loop with a solid line does not follow the single solid line rule, which has been explained in the Sequence Pattern (Section 4.2.2). This is clarified at the end of this section using examples of event trace.

Event-B Model:

The encoded Event-B model for the loop pattern is presented in Table 4.2. No control variable is generated for a loop leaf, since we do not need to record the loop event execution. Therefore there is no action for the loop event, *LoopEvent* here.

A guard is generated in the loop event to check that next event has not executed before, i.e., guard ($Event3 = FALSE$) in the SI case and guard ($p \notin Event3$) in the MI case.

The event after the loop event, is guarded by the execution condition of the event before the loop event. Considering the SI case, guard ($Event1 = TRUE$) and considering the MI case guard ($p \in Event1$) in event $Event3$, both check the execution of the event before the loop, $Event1$. This guard allows zero executions of the loop event. Right after execution of event before the loop, with zero execution of the loop event, the event after the loop can execute. That is why we do not need a variable and an action to record the loop execution.

An invariant is generated to show the sequencing between the event before the loop, $Event1$, and the event after the loop, $Event3$. The way that sequencing invariant is described is same as what described in the sequence pattern in Section 4.2.2.

Event Execution Trace Examples:

Considering the SI case diagram in Table 4.2, the event trace of the model in case of zero execution of the loop is:

< *Event1, Event3* >

And the event trace of the model in case of two executions of the loop is:

< *Event1, LoopEvent, LoopEvent, Event3* >

As mentioned in the restriction, a loop with a solid line is not allowed due to the Event-B restrictions. Assume the loop in the SI case diagram in Table 4.2 is connected to the abstract event with a solid line, and the other two sub-events are connected with dashed lines. If we remove the hidden sub-events (sub-events with dashed line) from the above event trace, the result is as follow:

$< LoopEvent,\ LoopEvent >$

Considering what has been explained in the Sequence Pattern in Section 4.2.2 about removing events from a trace, the just mentioned trace is supposed to be same as the abstract event trace, $< AbstractEvent >$, but it is not. Therefore the loop constructor in an atomicity decomposition diagram is always connected to the abstract event with a dashed line.

4.2.4 and-constructor Pattern

The and-constructor pattern is presented in Table 4.3. The table interpretation is same as what was described in terms of the sequence pattern table interpretation in Section 4.2.2.

Intention: The intention is to execute all two or more available sub-events in any order, in the right place in the sequence of other sub-events.

Diagrammatic Representation: The intention stated above is presented in atomicity decomposition diagram with the and-constructor, a circle containing *and*. All nodes connected to the and-constructor execute in any order in the sequence of other sub-events. For simplicity, in this pattern we consider two leaves for the and-constructor.

Restrictions: There are at least two nodes connected to the and-constructor. Following single solid line rule, the and-constructor is always connected to the root node with a dashed line, and all of the corresponding and-constructor events, $AndEvent1$ and $AndEvent2$ here, inherit dashed line from the and-constructor.

Event-B Model:

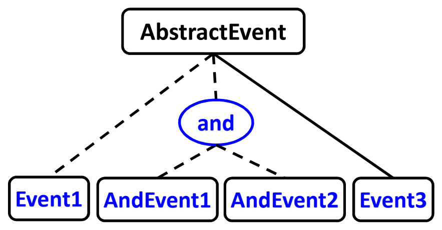
The encoded Event-B model for the and-constructor pattern is presented in Table 4.3. Each and-constructor event can execute only after execution of previous event, $Event1$. This is ensured with a guard, explained in the sequence pattern. The next event after the and-constructor can execute only after execution of all and-constructor events. Therefore a guard is generated in the event after the and-constructor, to ensures that all of the and-constructor events execute before. This guard is a logical conjunction between corresponding control variables generated for the and-constructor leaves. Considering the SI case, guard $(AndEvent1 = TRUE \wedge AndEvent2 = TRUE)$, and in the MI case guard $(p \in AndEvent1 \cap AndEvent2)$, are generated.

Comparing to sequence pattern invariants, the sequencing invariants for the event after the and-constructor is slightly changed in order to show the logical conjunction between control variables of the and-constructor events.

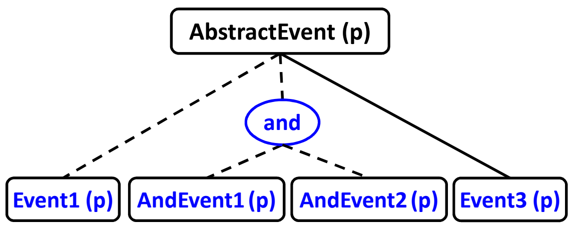
Name: and-constructor

Diagrammatic Representation

Single Instance(SI)



Multiple Instance(MI)



Event-B Model

Single Instance(SI) Invariants:

```
Invariants
  @inv_Event1_type Event1 ∈ BOOL
  @inv_AndEvent1_seq AndEvent1 = TRUE ⇒ Event1 = TRUE
  @inv_AndEvent2_seq AndEvent2 = TRUE ⇒ Event1 = TRUE
  @inv_Event3_seq Event3 = TRUE ⇒ (AndEvent1 = TRUE ∧ AndEvent2 = TRUE)
  @inv_Event3_gluing Event3 = AbstractEvent
```

Multiple Instance(MI) Invariants:

```
invariants
  @inv_Event1_type Event1 ⊆ TYPE(p)
  @inv_AndEvent1_seq AndEvent1 ⊆ Event1
  @inv_AndEvent2_seq AndEvent2 ⊆ Event1
  @inv_Event3_seq Event3 ⊆ AndEvent1 ∩ AndEvent2
  @inv_Event3_gluing Event3 = AbstractEvent
```

Single Instance(SI) Events:

```
event Event1
  where
    @grd Event1 = FALSE
  then
    @act Event1 ≔ TRUE
 end
```

```
event AndEvent1
  where
    @grd_seq Event1 = TRUE
    @grd AndEvent1 = FALSE
  then
    @act AndEvent1 ≔ TRUE
 end
```

```
event AndEvent2
  where
    @grd_seq Event1 = TRUE
    @grd AndEvent2 = FALSE
  then
    @act AndEvent2 ≔ TRUE
 end
```

```
event Event3 refines AbstractEvent
  where
    @grd_seq AndEvent1 = TRUE ∧ AndEvent2 = TRUE
    @grd Event3 = FALSE
  then
    @act Event3 ≔ TRUE
 end
```

Multiple Instance(MI) Events:

```
event Event1
  any p where
   @grd  p ∉ Event1
  then
   @act Event1 ≔ Event1 ∪ { p }
 end
```

```
event AndEvent1
  any p
  where
   @grd_seq p ∈  Event1
   @grd p ∉ AndEvent1
  then
   @act AndEvent1 ≔ AndEvent1 ∪ { p }
 end
```

```
event AndEvent2
  any p
  where
   @grd_seq p ∈  Event1
   @grd p ∉ AndEvent2
  then
   @act AndEvent2 ≔ AndEvent2 ∪ { p }
 end
```

```
event Event3 refines AbstractEvent
  any p
  where
   @grd_seq p ∈  (AndEvent1  ∩ AndEvent2)
   @grd p ∉ Event3
  then
   @act Event3 ≔ Event3 ∪ { p }
 end
```

Table 4.3: and-constructor Pattern

Event Execution Trace Examples:

Considering the SI case diagram in Table 4.3, the event traces of the model are as follows:

$< Event1,\ AndEvent1,\ AndEvent2,\ Event3 >$
$< Event1,\ AndEvent2,\ AndEvent1,\ Event3 >$

4.2.5 or-constructor Pattern, Multiple Choice

The or-constructor pattern is presented in Table 4.4. The table interpretation is the same as what was described in term of sequence pattern table interpretation in Section 4.2.2.

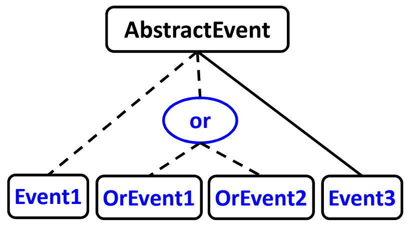
Intention: The intention is to execute one or more sub-events from two or more available sub-events, in any order, in the right place in the sequence of other sub-events.

Diagrammatic Representation: The intention stated above is presented in atomicity decomposition diagram with the or-constructor, a circle containing *or*. One or more nodes connected to the or-constructor execute in any order in the sequence of other sub-events. For simplicity, in this pattern we consider two leaves for the or-constructor.

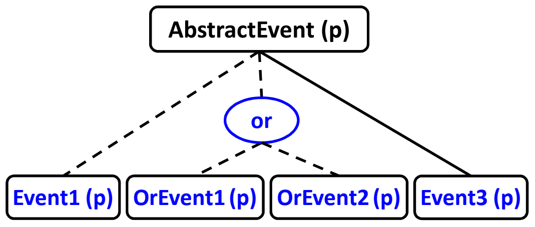
Name: or-constructor

Diagrammatic Representation

Single Instance(SI)



Multiple Instance(MI)



Event-B Model

Single Instance(SI) Invariants:

```
Invariants
  @inv_Event1_type Event1 ∈ BOOL
  @inv_OrEvent1_seq OrEvent1 = TRUE ⇒ Event1 = TRUE
  @inv_OrEvent2_seq OrEvent2 = TRUE ⇒ Event1 = TRUE
  @inv_Event3_seq Event3 = TRUE ⇒ (OrEvent1 = TRUE ∨ OrEvent2 = TRUE)
  @inv_Event3_gluing Event3 = AbstractEvent
```

Multiple Instance(MI) Invariants:

```
invariants
  @inv_Event1_type Event1 ⊆ TYPE(p)
  @inv_OrEvent1_seq OrEvent1 ⊆ Event1
  @inv_OrEvent2_seq OrEvent2 ⊆ Event1
  @inv_Event3_seq Event3 ⊆ OrEvent1 ∪ OrEvent2
  @inv_Event3_gluing Event3 = AbstractEvent
```

Single Instance(SI) Events:

```
event Event1
  where
    @grd Event1 = FALSE
  then
    @act Event1 ≔ TRUE
 end
```

```
event OrEvent1
  where
    @grd_seq Event1 = TRUE
    @grd OrEvent1 = FALSE
  then
    @act OrEvent1 ≔ TRUE
 end
```

```
event OrEvent2
  where
    @grd_seq Event1 = TRUE
    @grd OrEvent2 = FALSE
  then
    @act OrEvent2 ≔ TRUE
 end
```

```
event Event3 refines AbstractEvent
  where
    @grd_seq OrEvent1 = TRUE ∨ OrEvent2 = TRUE
    @grd Event3 = FALSE
  then
    @act Event3 ≔ TRUE
 end
```

Multiple Instance(MI) Events:

```
event Event1
  any p where
    @grd  p ∉ Event1
  then
    @act Event1 ≔ Event1 ∪ { p }
 end
```

```
event OrEvent1
  any p
  where
    @grd_seq p ∈  Event1
    @grd p ∉ OrEvent1
  then
    @act OrEvent1 ≔ OrEvent1 ∪ { p }
 end
```

```
event OrEvent2
  any p
  where
    @grd_seq p ∈  Event1
    @grd p ∉ OrEvent2
  then
    @act OrEvent2 ≔ OrEvent2 ∪ { p }
 end
```

```
event Event3 refines AbstractEvent
  any p
  where
    @grd_seq p ∈  (OrEvent1  ∪ OrEvent2)
    @grd_ p ∉ Event3
  then
    @act Event3 ≔ Event3 ∪ { p }
 end
```

Table 4.4: or-constructor Pattern

Restrictions: There are at least two nodes connected to the or-constructor. Following single solid line rule, the or-constructor is always connected to the root node with dashed line, and all of the corresponding or-constructor events, $OrEvent1$ and $OrEvent2$ here, inherit dashed line from the or-constructor.

Event-B Model:

The encoded Event-B model for the or-constructor pattern is presented in Table 4.4. Each or-constructor event can execute only after execution of previous event, $Event1$. This is ensured with a guard, explained in sequence pattern. Next event after the or-constructor in sequence can execute only after execution of at least one of the or-constructor events. Therefore a guard is generated in the event after the or-constructor, to ensures that at least one of the or-constructor events executes before. This guard is a disjunction between the corresponding control variables generated for the or-constructor events. Considering the SI case, guard $(OrEvent1 = TRUE \vee OrEvent2 = TRUE)$, and in the MI case guard $(p \in OrEvent1 \cup OrEvent2)$, are generated.

Comparing to sequence pattern invariants, the sequencing invariants for the event after the or-constructor is changed in order to show the disjunction between control variables of the or-constructor events.

Event Execution Trace Examples:

Considering the SI case diagram in Table 4.4, the event traces of the model are as follows:

$< Event1, OrEvent1, Event3 >$
$< Event1, OrEvent2, Event3 >$
$< Event1, OrEvent1, OrEvent2, Event3 >$
$< Event1, OrEvent2, OrEvent1, Event3 >$

4.2.6 xor-constructor Pattern, Exclusive Choice

The xor-constructor pattern is presented in Table 4.5. The table interpretation is the same as what was described in term of sequence pattern table interpretation in Section 4.2.2.

Intention: The intention is to execute exactly one event from two or more available sub-events, in the right place in the sequence of other sub-events.

Diagrammatic Representation: The intention stated above is presented in the atomicity decomposition diagram with the xor-constructor, a circle containing *xor*. Exactly one of the nodes connected to the xor-constructor executes in the sequence of other sub-events. The xor-constructor can connect to the root node either with solid line or dashed line. Since only one of the xor-constructor events execute in this pattern, so having solid line for the xor-constructor follows the single solid line rule. It is clarified in examples of event trace at the end of this section. For simplicity, in this pattern we consider two leaves for the xor-constructor.

Restrictions: There are at least two nodes connected to the xor-constructor.

Event-B Model:

The encoded Event-B model for the xor-constructor pattern is presented in Table 4.5. The Event-B model is almost like the or-constructor pattern. In each xor-constructor event, a guard is needed to ensure that other xor-constructor events have not executed. For example, in the SI case, guard $XorEvent2 = FALSE$ is generated in $XorEvent1$, and considering the MI case, guard $p \notin XorEvent2$ is generated in $XorEvent1$.

Also an extra invariant is provided to show that at any time only one of the xor-constructor events has executed or none of them has executed. In the SI case, invariant

$partition(\{XorEvent1, XorEvent2\} \cap \{TRUE\},$
$\{XorEvent1\} \cap \{TRUE\}, \{XorEvent2\} \cap \{TRUE\})$

shows that at any time only one the control boolean variables'value can be $TRUE$. And in the MI case invariant

$partition((XorEvent1 \cup XorEvent2), XorEvent1, XorEvent2)$

shows that the set control variables are disjoints. The *partition* operator in event-B is defined as follows:

$partition(E_0, E_1, ..., E_n) \equiv (E_0 = E_1 \cup ... \cup E_n) \wedge (i \neq j \Rightarrow E_i \cap E_j = \varnothing)$

If the xor-constructor is provided with a solid line, the each xor-constructor sub-event refines the abstract event. Also a gluing invariant is needed. The just stated invariants in the SI case and the MI case respectively are changed to:

$partition(\{AbstractEvent\} \cap \{TRUE\},$
$\{XorEvent1\} \cap \{TRUE\}, \{XorEvent2\} \cap \{TRUE\})$

$partition(AbstractEvent, XorEvent1, XorEvent2)$

These gluing invariant not only describe the exclusive choice property, but also they describe the relation between abstract variable and the xor-constructor control variables. Considering *partition* definition, the gluing invariants in the SI case and the MI case respectively describe:

$\{AbstractEvent\} \cap \{TRUE\} = (\{XorEvent1\} \cap \{TRUE\}) \cup (\{XorEvent2\} \cap \{TRUE\})$

$AbstractEvent = XorEvent1 \cup XorEvent2$

Event Execution Trace Examples:

Considering the SI case diagram in Table 4.5, the event traces of the model are as follows:

< *Event1, XorEvent1, Event3* >
< *Event1, XorEvent2, Event3* >

As mentioned above, the xor-constructor can be connected to the root node with a solid line. Assume the xor-constructor in the SI case diagram in Table 4.2 is connected to the abstract event with a solid line, and the other two sub-events are connected with dashed lines. If we remove the hidden sub-events (sub-events with dashed line) from the above event traces, the results are as follows:

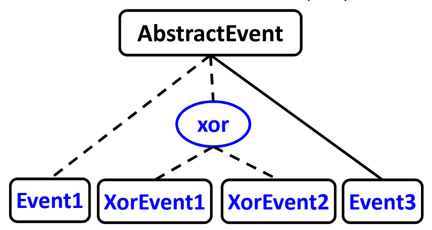
< *XorEvent1* >
< *XorEvent2* >

Considering what has been explained in the Sequence Pattern in Section 4.2.2 about removing events from a trace, the just mentioned traces are same as the abstract event trace, < *AbstractEvent* >, since both xor-constructor events refine the *AbstractEvent*.

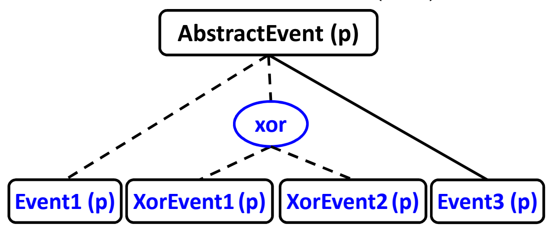
Name: xor-constructor

Diagrammatic Representation

Single Instance(SI)



Multiple Instance(MI)



Event-B Model

Single Instance(SI) Invariants:

```
invariants
  @inv_Event1_type Event1 ∈  BOOL
  @inv_XorEvent1_seq XorEvent1 = TRUE ⇒ Event1 = TRUE
  @inv_XorEvent2_seq XorEvent2 = TRUE ⇒ Event1 = TRUE
  @inv_Event3_seq Event3 = TRUE ⇒ (XorEvent1 = TRUE ∨ XorEvent2 = TRUE)
  @inv_xor partition( {XorEvent1, XorEvent2} ∩ {TRUE} ,
                      {XorEvent1} ∩ {TRUE}, {XorEvent2} ∩ {TRUE})
  @inv_Event3_gluing Event3 = AbstractEvent
```

Multiple Instance(MI) Invariants:

```
invariants
  @inv_Event1_type Event1 ⊆ TYPE(p)
  @inv_XorEvent1_seq XorEvent1 ⊆ Event1
  @inv_XorEvent2_seq XorEvent2 ⊆ Event1
  @inv_Event3_seq Event3 ⊆ XorEvent1 ∪ XorEvent2
  @inv_xor partition((XorEvent1 ∪ XorEvent2), XorEvent1, XorEvent2)
  @inv_Event3_gluing Event3 = AbstractEvent
```

Single Instance(SI) Events:

```
event Event1
  where
    @grd Event1 = FALSE
  then
    @act Event1 ≔ TRUE
end
```

```
event XorEvent1
  where
    @grd_seq Event1 = TRUE
    @grd XorEvent1 = FALSE
    @grd_xor XorEvent2 = FALSE
  then
    @act XorEvent1 ≔ TRUE
end
```

```
event XorEvent2
  where
    @grd_seq Event1 = TRUE
    @grd XorEvent2 = FALSE
    @grd_xor XorEvent1 = FALSE
  then
    @act XorEvent2 ≔ TRUE
end
```

```
event Event3 refines AbstractEvent
  where
    @grd_seq XorEvent1 = TRUE ∨ XorEvent2 = TRUE
    @grd Event3 = FALSE
  then
    @act Event3 ≔ TRUE
end
```

Multiple Instance(MI) Events:

```
event Event1
  any p where
   @grd  p ∉ Event1
  then
   @act Event1 ≔ Event1 ∪ { p }
 end
```

```
event XorEvent1
  any p
  where
   @grd_seq p ∈  Event1
   @grd p ∉ XorEvent1
   @grd_xor p ∉ XorEvent2
  then
  @act XorEvent1 ≔ XorEvent1 ∪ { p }
 end
```

```
event XorEvent2
  any p
  where
   @grd_seq p ∈  Event1
   @grd p ∉ XorEvent2
   @grd_xor p ∉ XorEvent1
  then
  @act XorEvent2 ≔ XorEvent2 ∪ { p }
 end
```

```
event Event3 refines AbstractEvent
  any p
  where
   @grd_seq p ∈  (XorEvent1  ∪ XorEvent2)
   @grd p ∉ Event3
  then
   @act Event3 ≔ Event3 ∪ { p }
 end
```

Table 4.5: xor-constructor Pattern

4.2.7 all-replicator Pattern

The all-replicator pattern is presented in Table 4.6. The table interpretation is the same as what was described in term of sequence pattern table interpretation in Section 4.2.2.

Intention: The intention is to execute a sub-event for all instances of a new parameter, in the right place in the sequence of other sub-events. The all-replicator is a generalisation of the and-constructor.

Diagrammatic Representation: The all-replicator is presented with a circle containing *all* flowed by name of a new parameter.

Restrictions: Based on the single solid line rule, the all-replicator is always connected to the root event with dashed line, since the all-replicator event can execute for more than one time depending on the number of new introduced *all* parameter instances.

Event-B Model:

The encoded Event-B model for the all-replicator pattern is presented in Table 4.6. The all-replicator parameter, $p2$, is added to the sub-event connected to the all-replicator, $AllEvent$, as a new dimension.

The type of generated control variable for the all-replicator event has got one more dimension compared with other sub-events. Because the all-replicator introduces a new parameter. An invariant is generated to define the type of the all-replicator control variable. For instances considering the SI case, variable $AllEvent$ is a subset of type of new parameter $p2$, $TYPE(p2)$, whereas other control variables are boolean variables. In the MI case $AllEvent$'s variable is defined as a cartesian product of the root parameter's type $TYPE(p1)$ and the all-replicator parameter's type, $TYPE(p2)$.

The event after the all-replicator event in sequence, $Event3$, can execute only after execution of the all-replicator event, $AllEvent$, for all instances of the new parameter, $p2$. A guard is generated in next event, $Event3$, to ensure this property. Guard $(AllEvent = TYPE(p2))$ in event $Event3$ in the SI case, ensures that event $AllEvent$ has executed for all instances of $p2$ before. Also considering the MI case, guard $(AllEvent[\{p1\}] = TYPE(p2))$ plays same role. Relational image r[S] in Event-B is defined as below:

$r[S] = \{y | \exists x.x \in S \land x \mapsto y \in r\}$

Considering relational image definition, guard $(AllEvent[\{p1\}] = TYPE(p2))$ ensures that for $p1$, $AllEvent$ has executed for all instances of $p2$ from set $TYPE(p2)$.

An invariant is generated to model the all-replicator condition: $(p1 \in Event3 \Rightarrow AllEvent[\{p1\}] = TYPE(p2))$ in MI case and $(Event3 = TRUE \Rightarrow AllEvent = TYPE(p2)$ in the SI case.

Event Execution Trace Examples:

Considering the SI case diagram in Table 4.6, assume $p2 \in \{a, b\}$, the the event traces of the model are as follows:

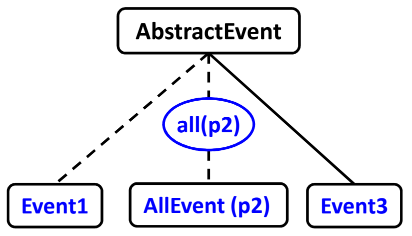
< Event1, AllEvent(a), AllEvent(b), Event3 >
< Event1, AllEvent(b), AllEvent(a), Event3 >

Number of executions of $AllEvent$ is always equal to cardinality of the all-replicator parameter's type set. In this example $AllEvent$ executes for two times, since $card(\{a, b\}) = 2$.

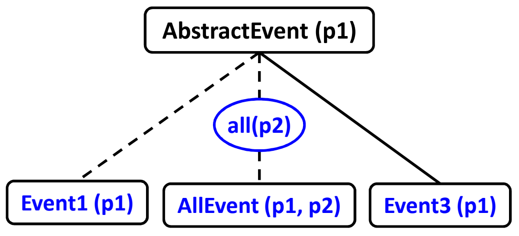
Name: all-replicator

Diagrammatic Representation

Single Instance(SI)



Multiple Instance(MI)



Event-B Model

Single Instance(SI) Invariants:

```
invariants
  @inv_Event1_type Event1 ∈  BOOL
  @inv_AllEvent_type AllEvent ⊆ TYPE(p2)
  @inv_AllEvent_seq AllEvent ≠ ∅ ⇒ Event1 = TRUE
  @inv_Event3_seq Event3 = TRUE ⇒ AllEvent = TYPE(p2)
  @inv_Event3_gluing Event3 = AbstractEvent
```

Multiple Instance(MI) Invariants:

```
Invariants
  @inv_Event1_type Event1 ⊆ TYPE(p1)
  @inv_AllEvent_type  AllEvent ⊆ TYPE(p1) × TYPE(p2)
  @inv_AllEvent_seq  dom( AllEvent ) ⊆ Event1
  @inv_Event3_seq  p1 ∈ Event3 ⇒  AllEvent [ { p1 } ] = TYPE(p2)
  @inv_Event3_gluing Event3 = AbstractEvent
```

Single Instance(SI) Events:

```
event Event1
   where
    @grd Event1 = FALSE
   then
    @act Event1 ≔ TRUE
 end
```

```
event AllEvent
   any p2
   where
    @grd_seq Event1 = TRUE
    @grd p2 ∉ AllEvent
   then
    @act AllEvent ≔ AllEvent ∪ { p2 }
 end
```

```
event Event3 refines AbstractEvent
   where
   @grd_seq AllEvent = TYPE(p2)
   @grd Event3 = FALSE
then
    @act Event3 ≔ TRUE
 end
```

Multiple Instance(MI) Events:

```
event Event1
   any p where
    @grd  p ∉ Event1
   then
    @act Event1 ≔ Event1 ∪ { p }
 end
```

```
event AllEvent
   any p1 p2
   where
    @grd_seq p1 ∈  Event1
    @grd p1 ↦ p2 ∉ AllEvent
   then
    @act AllEvent ≔ AllEvent ∪ { p1 ↦ p2 }
 end
```

```
event Event3  refines AbstractEvent
   any p1
   where
    @grd_seq AllEvent [ { p1 } ] = TYPE(p2)
    @grd  p1 ∉  Event3
   then
    @act Event3 ≔ Event3 ∪ { p1 }
 end
```

Table 4.6: all-replicator Pattern

4.2.8 some-replicator Pattern

The some-replicator pattern is presented in Table 4.7. The table interpretation is the same as what was described in term of sequence pattern table interpretation in Section 4.2.2.

Intention: The intention is to execute a sub-event for one or more instances of a new parameter, in the right place in the sequence of other sub-events. The some-replicator is a generalisation of the or-constructor.

Diagrammatic Representation: The some-replicator is presented with a circle containing *some* followed by name of a new parameter.

Restrictions: Based on the single solid line rule, the some-replicator is always connected to the root event with dashed line, since the some-replicator event can execute for more than one time.

Event-B Model:

The encoded Event-B model for the some-replicator pattern is presented in Table 4.7. The some-replicator parameter, $p2$, is added to the sub-event connected to the some-replicator, $SomeEvent$, as a new dimension.

The type of generated control variable for the some-replicator event is defined with an invariant as described in the all-replicator pattern.

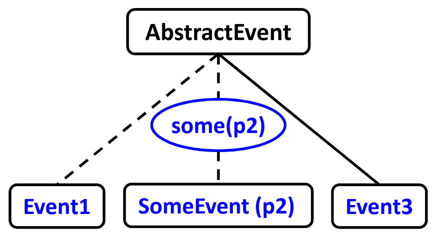
The event after the some-replicator event in the sequence, $Event3$, can execute only after execution of the some-replicator event, $SomeEvent$, at least for one of the instances of the new parameter, $p2$. The sequencing guard ($SomeEvent \neq \varnothing$) in event $Event3$ in the SI case, ensures that event $SomeEvent$ has executed for one or more instances of $p2$ before. Also considering the MI case, guard ($p1 \in dom(SomeEvent)$) ensures that $card(SomeEvent[\{p1\}]) \geq 1$. It means for $p1$, event $Event3$ executes for at least one instance of $p2$.

The sequencing invariant generated for $Event3$, ($Event3 \subseteq dom(SomeEvent)$), also shows one or more execution of $SomeEvent$ before execution of $Event3$.

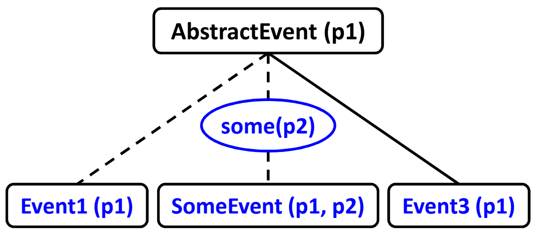
Name: some-replicator

Diagrammatic Representation

Single Instance(SI)



Multiple Instance(MI)



Event-B Model

Single Instance(SI) Invariants:

```
invariants
  @inv_Event1_type Event1 ∈  BOOL
  @inv_SomeEvent_type SomeEvent ⊆ TYPE(p2)
  @inv_SomeEvent_seq SomeEvent ≠ ∅ ⇒ Event1 = TRUE
  @inv_Event3_seq Event3 = TRUE ⇒ SomeEvent ≠ ∅
  @inv_Event3_gluing Event3 = AbstractEvent
```

Multiple Instance(MI) Invariants:

```
Invariants
  @inv_Event1_type Event1 ⊆ TYPE(p1)
  @inv_SomeEvent_type  SomeEvent ⊆ TYPE(p1) × TYPE(p2)
  @inv_SomeEvent_seq  dom( SomeEvent ) ⊆ Event1
  @inv_Event3_seq  Event3 ⊆  dom( SomeEvent )
  @inv_Event3_gluing Event3 = AbstractEvent
```

Single Instance(SI) Events:

```
event Event1
  where
    @grd Event1 = FALSE
  then
    @act Event1 ≔ TRUE
 end
```

```
event SomeEvent
  any p2
  where
    @grd_seq Event1 = TRUE
    @grd p2 ∉ SomeEvent
  then
    @act SomeEvent ≔ SomeEvent ∪ { p2 }
 end
```

```
event Event3 refines AbstractEvent
  where
   @grd_seq SomeEvent ≠ ∅
   @grd Event3 = FALSE
then
    @act Event3 ≔ TRUE
 end
```

Multiple Instance(MI) Events:

```
event Event1
  any p where
    @grd  p ∉ Event1
  then
    @act_Event1 Event1 ≔ Event1 ∪ { p }
 end
```

```
 event SomeEvent
  any p1 p2
  where
    @grd_seq p1 ∈  Event1
    @grd p1 ↦ p2 ∉ SomeEvent
  then
    @act SomeEvent ≔ SomeEvent ∪ { p1 ↦ p2 }
 end
```

```
event Event3  refines AbstractEvent
  any p1
  where
    @grd_seq p1 ∈  dom( SomeEvent )
    @grd  p1 ∉  Event3
 then
    @act Event3 ≔ Event3 ∪ { p1 }
 end
```

Table 4.7: some-replicator Pattern

Event Execution Trace Examples:

Considering the SI case diagram in Table 4.7, assume $p2 \in \{a, b\}$, the event traces of the model are as follows:

< Event1, AllEvent(a), AllEvent(b), Event3 >
< Event1, AllEvent(b), AllEvent(a), Event3 >
< Event1, AllEvent(a), Event3 >
< Event1, AllEvent(b), Event3 >

The number of the some-replicator event execution is always less than or equal to the cardinality of the some-replicator parameter's type set. In above event traces, $SomeEvent$ executes for one or two times, since $card(\{a, b\}) = 2$.

4.2.9 one-replicator Pattern

The one-replicator pattern is presented in Table 4.8. The table interpretation is the same as what was described in term of sequence pattern table interpretation in Section 4.2.2.

Intention: The intention is to execute a sub-event for exactly one instance of a new parameter, in the right place in the sequence of other sub-events. The one-replicator is a generalisation of the xor-constructor.

Diagrammatic Representation: The one-replicator is presented with a circle containing *one* flowed by name of a new parameter. Following the single solid line rule, the one-replicator can be connected to the root event with either dashed line of solid line, since the one-replicator event can execute for only one instance.

Event-B Model:

The encoded Event-B model for the one-replicator pattern is presented in Table 4.8. The one-replicator parameter, $p2$, is added to the sub-event connected to the one-replicator, $OneEvent$, as a new dimension.

Type of generated control variable for the one-replicator event is defined with an invariant as described in the all-replicator pattern.

The event after the one-replicator event in the sequence, $Event3$, can execute only after execution of the one-replicator event, $OneEvent$, for exactly one of the instances of the new parameter, $p2$. The sequencing guard in event $Event3$ is same as the one in the some-replicator pattern. In order to restrict the number of the one-replicator event executions, we provide a guard in the one-replicator event. Considering the SI case, the guard ($OneEvent = \varnothing$) in event $OneEvent$ ensures that event $OneEvent$ can execute only for one time. And in the MI case, guard $(p1 \notin dom(OneEvent))$ ensures that for $p1$, event $OneEvent$ can execute only for one instance of $p2$.

An invariant is generated to show that the one-replicator event can execute only for one time (for each instance of event parameter in the MI case). In the SI case, $(card(OneEvent) \leqslant 1)$, and the MI case, invariant $(\forall p.card(OneEvent[\{p\}]) \leqslant 1)$.

A gluing invariant is generated for the one-replicator with the solid line. The gluing invariant in the SI case and the MI case respectively are as follows:

$OneEvent \neq \varnothing \Leftrightarrow AbstractEvent = TRUE$

$dom(OneEvent) = AbstractEvent$

Event Execution Trace Examples:

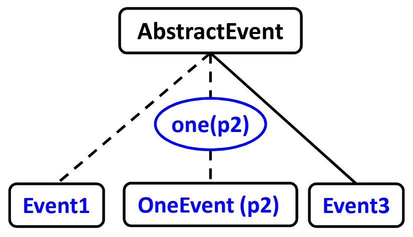
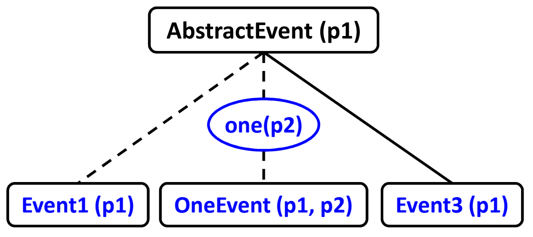
Considering the SI case diagram in Table 4.8, assume $p2 \in \{a, b\}$, the event traces of the model are as follows:

< Event1, OneEvent(a), Event3 >
< Event1, OneEvent(b), Event3 >

The one-replicator event can execute exactly for one instance of the new parameter.

Name: one-replicator

Diagrammatic Representation

Single Instance(SI)	Multiple Instance(MI)
 AbstractEvent one(p2) Event1 OneEvent (p2) Event3	 AbstractEvent (p1) one(p2) Event1 (p1) OneEvent (p1, p2) Event3 (p1)

Event-B Model

Single Instance(SI) Invariants:

```
invariants
  @inv_Event1_type Event1 ∈ BOOL
  @inv_OneEvent_type OneEvent ⊆ TYPE(p2)
  @inv_OneEvent_seq OneEvent ≠ ∅ ⇒ Event1 = TRUE
  @inv_Event3_seq Event3 = TRUE ⇒ OneEvent ≠ ∅
  @inv_OneEvent_one card(OneEvent) ≤ 1
  @inv_Event3_gluing Event3 = AbstractEvent
```

Multiple Instance(MI) Invariants:

```
invariants
  @inv_Event1_type Event1 ⊆ TYPE(p1)
  @inv_OneEvent_type  OneEvent ⊆ TYPE(p1) × TYPE(p2)
  @inv_OneEvent_seq  dom( OneEvent ) ⊆ Event1
  @inv_Event3_seq  Event3 ⊆  dom( OneEvent )
  @inv_OneEvent_one ∀p· card( OneEvent [{p}] ) ≤ 1
  @inv_Event3_gluing Event3 = AbstractEvent
```

Single Instance(SI) Events:

```
event Event1
  where
   @grd Event1 = FALSE
  then
   @act Event1 ≔ TRUE
 end
```

```
event OneEvent
  any p2
  where
   @grd_seq Event1 = TRUE
   @grd p2 ∉ OneEvent
    @grd_one OneEvent = ∅
  then
   @act  OneEvent ≔ OneEvent ∪ { p2 }
 end
```

```
event Event3 refines AbstractEvent
  where
  @grd_seq OneEvent ≠ ∅
  @grd Event3 = FALSE
then
   @act Event3 ≔ TRUE
 end
```

Multiple Instance(MI) Events:

```
event Event1
  any p where
   @grd  p ∉ Event1
  then
   @act_Event1 Event1 ≔ Event1 ∪ { p }
 end
```

```
event OneEvent
  any p1 p2
  where
   @grd_seq p1 ∈  Event1
   @grd p1 ↦ p2 ∉ SomeEvent
   @grd_one p1 ∉ dom( OneEvent )
  then
   @act OneEvent ≔ OneEvent ∪ { p1 ↦ p2 }
 end
```

```
event Event3 refines AbstractEvent
  any p1
  where
   @grd_seq p1 ∈  dom( OneEvent )
   @grd  p1 ∉  Event3
  then
   @act Event3 ≔ Event3 ∪ { p1 }
end
```

Table 4.8: one-replicator Pattern

4.3 Additional Features of the Atomicity Decomposition Approach

4.3.1 The Most Abstract Level

The most abstract level of an Event-B model is illustrated in a diagram that aids understanding, shown in Figure 4.1. The name of a process in the system appears in an oval as the root node, and the names of most abstract events appear in the leaves in order from left to right. All lines have to be dashed lines, since all of leaves are the most abstract events and do not refine the root node. The Event-B model is the same as presented in patterns, Section 4.2. The only difference is that in the most abstract level, there is no refining event (no solid line) and no gluing invariant in the Event-B model.

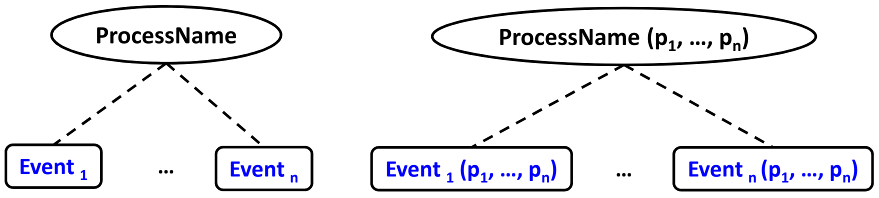

ProcessName
Event $_1$
...
Event $_n$
ProcessName (p_1, ..., p_n)
Event $_1$ (p_1, ..., p_n)
...
Event $_n$ (p_1, ..., p_n)

Figure 4.1: The Most Abstract Level Diagrams

4.3.2 Combined Atomicity Decomposition Diagram

In an atomicity decomposition diagram, root node, $AbstractEvent$ in described patterns in Section 4.2, is one of the events in $(i)^{th}$ refinement level which decomposed into some sub-events in $(i+1)^{th}$ refinement level. Later each sub-events can be further decomposed to some other sub-events in the next refinement level, $(i+2)^{th}$ refinement level, and so on. The reason in the patterns we called the root node, $AbstractEvent$, is that comparing with sub-events, $AbstractEvent$ is placed in an earlier level of refinement which can be considered as an abstract level for the sub-events refinement level.

Starting from the most abstract level diagram, the atomicity decomposition diagrams for different events can be combined in a single diagram. An example is illustrated in Figure 4.2. In this example, there are four abstract events, $Event_1$, $Event_2$, $Event_3$ and $Event_4$, in the most abstract level. In the first refinement level, $Event_2$ is decomposed to $Event_5$ followed by one instance of $Event_6$. Also $Event_4$ is decomposed to three sequential sub-events, $Event_7$, followed by a loop constructor applied to $Event_8$, followed by $Event_9$.

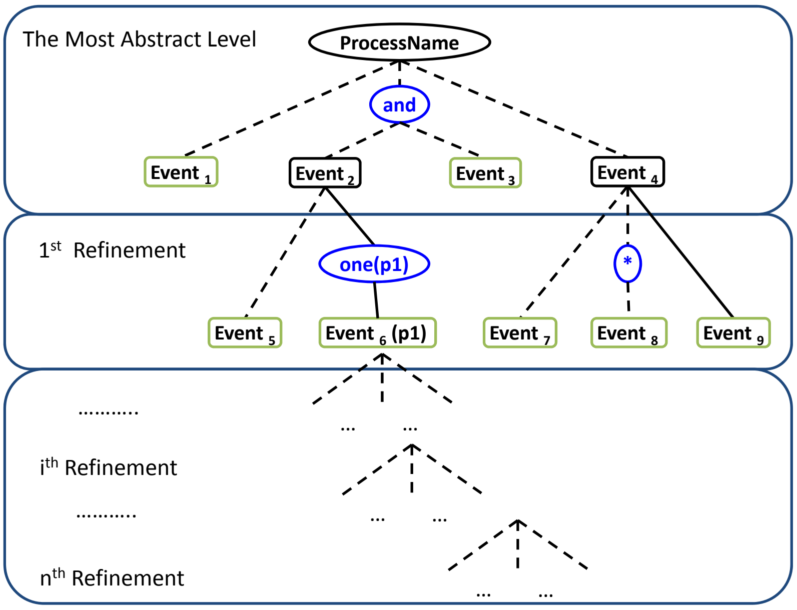


Figure 4.2: Combined Atomicity Decomposition Diagram

The combined atomicity decomposition diagram provides the overall visualization of the refinement structure. The benefits of combined atomicity decomposition diagram will be explained more in the evaluation chapter, Section 8.4. In a combined atomicity decomposition diagram, each leaf is encoded as one event in the Event-B model. A leaf is a node without any child. For example, in the first refinement level of Figure 4.2, the leaves are $Event_1$, $Event_5$, $Event_6$, $Event_3$, $Event_7$, $Event_8$, $Event_9$.

The general atomicity decomposition language which describes the structure of the combined atomicity decomposition diagram and translation rules to the Event-B model are presented in Chapter 4.

4.3.3 Several Atomicity Decompositions for a Single Event

A single event can be decomposed to some sub-events in different styles. In other words several atomicity decomposition diagrams can be defined for a same root node. An example is illustrated in Figure 4.3. *Event_a* is decomposed in two different diagrams in the next refinement level. The Event-B model follows the rules that presented in patterns, Section 4.2.

The benefits of having several atomicity decompositions for a single event will be highlighted later in Section 8.2.

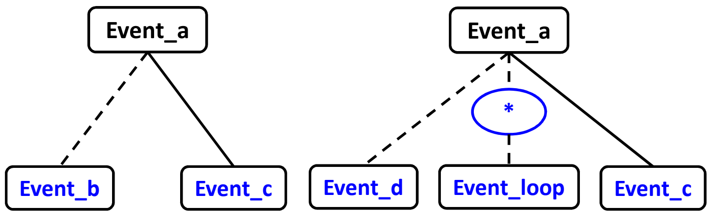


Figure 4.3: Several Atomicity Decomposition for a Single Event, *Event_a*

4.3.4 Strong Sequencing versus Weak Sequencing

In a combined atomicity decomposition diagram, there are two approaches of sequencing applied to a single root event: Strong Sequencing and Weak Sequencing. Strong/weak sequencing property is applied to each single atomicity decomposition of a root event. If strong sequencing is applied to a root event, then there is a sequencing between all sub-events of that root and the previous and next sub-events of the earlier refinement level. Whereas in the case of weak sequencing, the sequencing is applied only to the sub-event with solid line of the root and the previous and next sub-events of the earlier refinement level.

To make the point clear, an example of a combined atomicity decomposition diagram is presented in Figure 4.4. *Event_a* is decomposed to four sub-events, *Event_b*, *Event_c*, *Event_d* and *Event_a*, in $(i)^{th}$ refinement level. Then *Event_c* is decomposed to three sub-events, *Event_f*, *Event_c* and *Event_g* in $(i+1)^{th}$ refinement level.

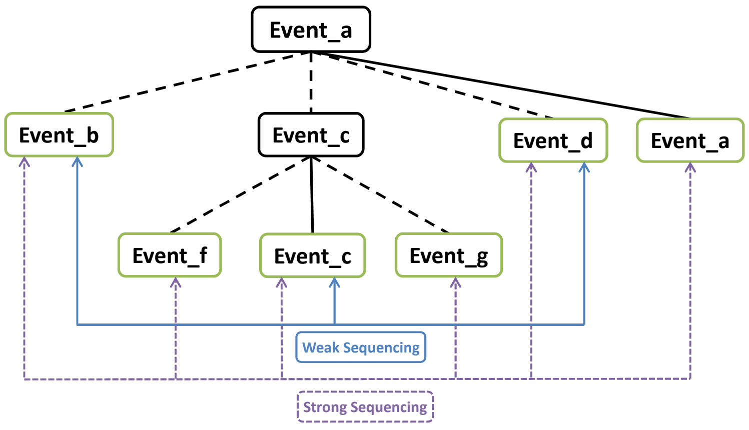


Figure 4.4: Strong Sequencing, Weak Sequencing

Assume atomicity decomposition of *Event_c* root event has strong sequencing, then the only possible event trace is:

$< Event\_b, Event\_f, Event\_c, Event\_g, Event\_d, Event\_a >$

Whereas if atomicity decomposition of *Event_c* has weak sequencing, then on one hand there is an ordering just between the leaf with solid line, *Event_c* and the previous and next leaves in sequence, $Event\_b$ and $Event\_d$ respectively. And on the other hand there is no ordering constraints between $Event\_b$ and $Event\_f$, and between $Event\_g$ and $Event\_d$. Therefore, because of weak sequencing, there are more than one possible event trace:

$< Event\_b, Event\_f, Event\_c, Event\_g, Event\_d, Event\_a >$
$< Event\_b, Event\_f, Event\_c, Event\_d, Event\_g, Event\_a >$
$< Event\_f, Event\_b, Event\_c, Event\_g, Event\_d, Event\_a >$
$< Event\_f, Event\_b, Event\_c, Event\_d, Event\_g, Event\_a >$

In all of possible event traces, $Event\_c$ executes after execution of $Event\_b$, before $Event\_d$. It is important to mention that in a single atomicity decomposition, there is always an ordering between sub-events of the root event, in both strong and weak sequencing approaches. For example, $Event\_f$, $Event\_c$ and $Event\_g$ always execute in order.

The weak and strong sequencing is managed with some invariants and guards. The general translation rules to the Event-B model are presented in Chapter 4.

The most abstract atomicity decomposition diagram always has a strong sequencing, since the most abstract diagram is placed in the top level of combined an atomicity decomposition diagram.

4.3.5 Loop Resetting Event

As described in the Loop Pattern in Section 4.2.3, if the loop event is a single event, then we do not consider a variable for the loop event. Considering the example in Figure 4.5, in decomposing the atomicity of *Event_a*, *Event_c* can execute zero or more time before execution of *Event_d*. And the execution of *Event_d* here, does not depend on the loop execution.

In the next refinement level, the loop event is decomposed to some sub-events. So we have to consider some control variables to manage the ordering between the loop events, *Event_e*, *Event_f* and *Event_g*. Also a resetting event is needed to reset the loop control variables to enable more than one execution of the loop. Furthermore an extra guard is needed in *Event_d* to ensure that *Event_d* does not execute in the middle of the execution of the loop events.

Loop resetting can be done in three ways. Each of them for the example in Figure 4.5, is illustrated with a state diagram and its Event-B model in Figure 4.6, Figure 4.7 and Figure 4.8.

First, as illustrated in Figure 4.6, the reset event is considered as a separate event, called *Reset* here. The ordering between loop events are managed with some control variables,

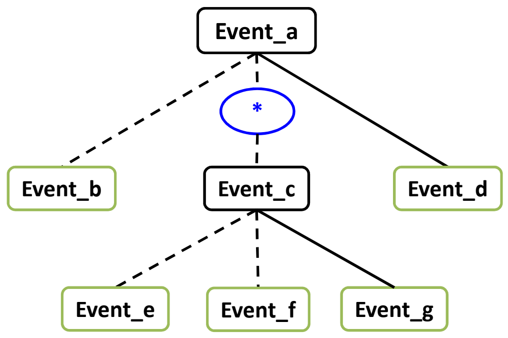
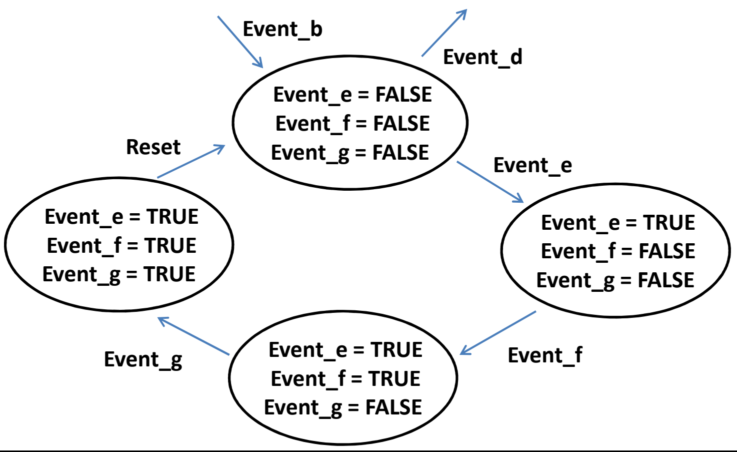


Figure 4.5: Loop Resetting Example

Event_e, *Event_f* and *Event_g*. The first event in the Loop, *Event_e* checks that the next event after the loop has not execute before, ($Event\_d = FALSE$). A guard in *Event_d* ensures that it can not execute in the middle of the loop, ($Event\_e = FALSE$).

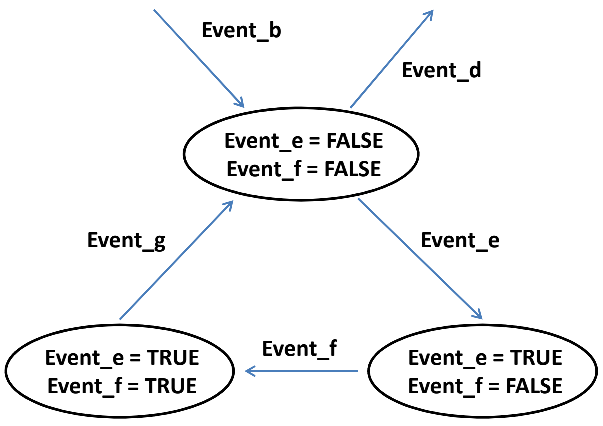


```
event Event_e where Event_b = TRUE ∧ Event_d = FALSE ∧ Event_e = FALSE then Event_e ≔ TRUE end
event Event_f  where Event_e = TRUE ∧ Event_f = FALSE  then  Event_f ≔ TRUE  end
event Event_g  where Event_f = TRUE ∧  Event_g = FALSE  then  Event_g ≔ TRUE end

event Reset where  Event_g = TRUE  then  Event_e ≔ FALSE,
                                         Event_f ≔ FALSE,
                                         Event_g ≔ FALSE  end

event Event_d where Event_b = TRUE ∧ Event_e = FALSE ∧ Event_d = FALSE then Event_d ≔ TRUE end
```

Figure 4.6: Loop Resetting as a Separate Event

Second, as illustrated in Figure 4.7, the resetting is merged in the last event of the loop, *Event_g*. In this case we do not need a control variable for the last event, since the last event resets the loop.

Last, as illustrated in Figure 4.8, the resetting is merged in the first event of the loop, *Event_e*. In this case we have to consider a separate event for the first event of the loop, *Event_e1*. The resetting is done in *Event_e2*. In this case *Event_d*'s guard is complex,



```
event Event_e where Event_b = TRUE ∧ Event_d = FALSE ∧ Event_e = FALSE then Event_e ≔ TRUE end

event Event_f  where  Event_e = TRUE ∧ Event_f = FALSE  then  Event_f ≔ TRUE  end

event Event_g where  Event_f = TRUE  then  Event_e ≔ FALSE,
                                           Event_f ≔ FALSE   end

event Event_d where Event_b = TRUE ∧ Event_e = FALSE ∧ Event_d = FALSE then Event_d ≔ TRUE end
```

Figure 4.7: Loop Resetting in the Last Event

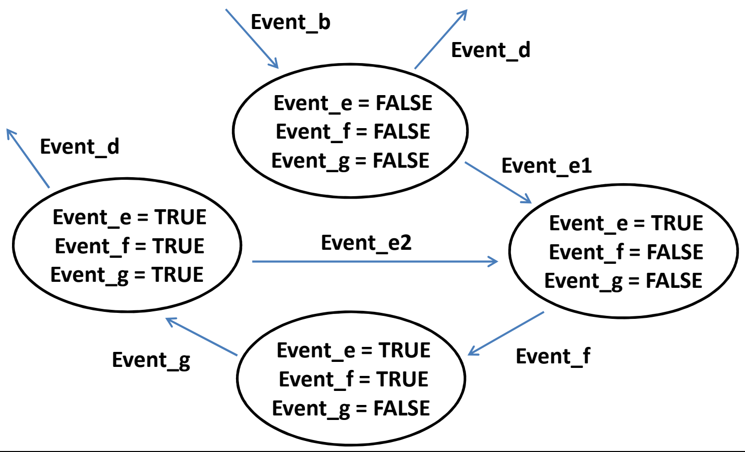
since we need to consider two cases. First zero execution of the loop, ($Event\_e = FALSE$) and second, one or more execution(s) of the loop, ($Event\_g = TRUE$).

We adopted the separate resetting event for the loop in Figure 4.6. Considering the example in Figure 4.9, assume the case when the first sub-event in decomposing the loop event, $Event\_c$, is either the and-constructor or the or-constructor or the xor-constructor. Then the resetting approach presented in Figure 4.8, needs to be applied to all of the constructor children, $Event\_e$ and $Event\_f$ here. Also in the resetting approach presented in Figure 4.7, if the last sub-event is either the and-constructor or the or-constructor or the xor-constructor, then the resetting needs to be applied to all of the constructor children, and this can make the Event-B model large and complex comparing to the approach when we provide the separate resetting event.

Using a separate event to reset loop, the Event-B model of the example presented in Figure 4.5, is presented in Figure 4.10, in the MI case (having one parameter).

4.4 Different Approaches to Model Ordering in Event-B

In the described patterns in Section 4.2, we used subset relationships to manage ordering between events. A simple example is presented in Figure 4.11. The subset invariant ($B \subseteq A$), specifies one variable as a subset of the other. The first event, A, is only enabled when parameter x is not is the A set. The action of the event adds the parameter



event Event_e1 where Event_b =TRUE ∧ Event_d=FALSE ∧ Event_e =FALSE **then** Event_e≔ TRUE **end**
event Event_e2 where Event_g = TRUE ∧ Event_d= FALSE **then** Event_f≔ FALSE,
Event_g≔ FALSE **end**

event Event_f where Event_e = TRUE ∧ Event_f = FALSE **then** Event_f ≔ TRUE **end**
event Event_g where Event_f = TRUE ∧ Event_g = FALSE **then** Event_g ≔ TRUE **end**

event Event_d where Event_b = TRUE ∧ (Event_e = FALSE ∨ Event_g = TRUE) ∧ Event_d = FALSE
then Event_d ≔ TRUE **end**

Figure 4.8: Loop Resetting in the First Event

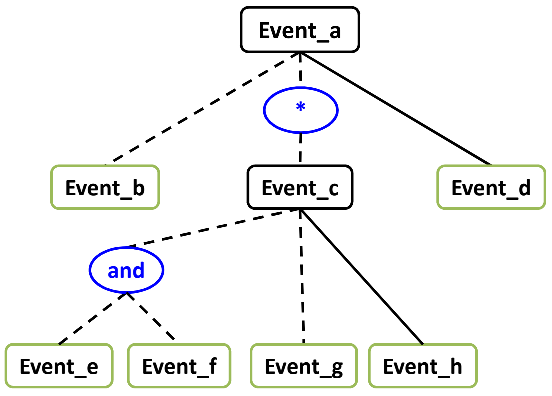


Figure 4.9: Loop Resetting Example

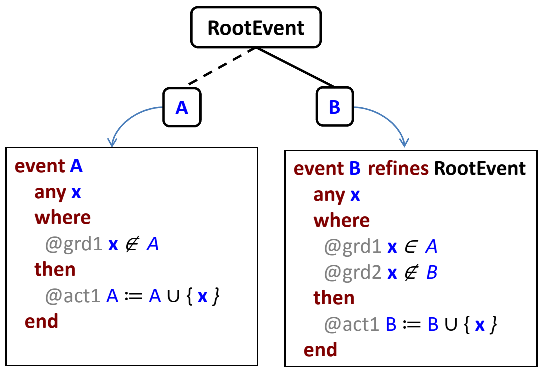
event Event_e where p_1 ∈ Event_b ∧ p_1 ∉ Event_d ∧ p_1 ∉ Event_e **then** Event_e ≔ Event_e ∪ { p_1}
event Event_f where p_1 ∈ Event_e ∧ p_1 ∉ Event_f **then** Event_f ≔ Event_f ∪ { p_1} **end**
event Event_g where p_1 ∈ Event_f ∧ p_1 ∉ Event_g **then** Event_g ≔ Event_g ∪ { p_1} **end**

event Reset where p_1 ∈ Event_g **then** Event_e ≔ Event_e / { p_1},
Event_f ≔ Event_f / { p_1},
Event_g ≔ Event_g / { p_1} **end**

event Event_d where p_1 ∈ Event_b ∧ p_1 ∉ Event_e ∧ p_1 ∉ Event_d **then** Event_d ≔ Event_d ∪ { p_1}

Figure 4.10: Loop Resetting with Parameter

to the set variable A. The second event can only execute when the parameter is in A set and not in B set. The action of the event then adds the parameter to the B set.

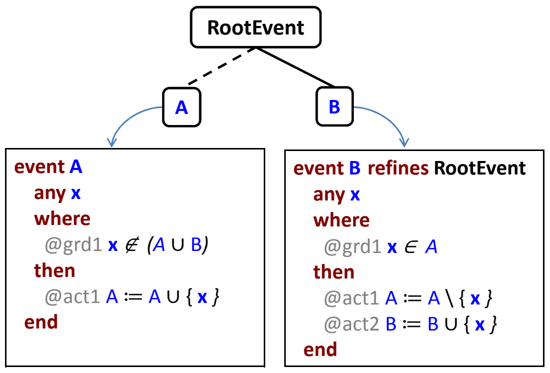


```
event A
  any x
  where
   @grd1 x ∉ A
  then
   @act1 A ≔ A ∪ { x }
 end
```

```
event B refines RootEvent
  any x
  where
   @grd1 x ∈ A
   @grd2 x ∉ B
  then
   @act1 B ≔ B ∪ { x }
 end
```

Figure 4.11: Subset Sets

An alternative is to use disjoint sets [60] and to remove the parameter from one set before it can move to the next set. Figure 4.12 shows an example that used disjoint sets to model ordering between two events. The variables A and B are modelled as disjoint, $(A \cap B = \varnothing)$. The event A takes a parameter that is neither of the sets and adds it to A set. The event B takes a parameter that is in the A set, removes it and adds it to B set.



```
event A
  any x
  where
   @grd1 x ∉ (A ∪ B)
  then
   @act1 A ≔ A ∪ { x }
 end
```

```
event B refines RootEvent
  any x
  where
   @grd1 x ∈ A
  then
   @act1 A ≔ A \ { x }
   @act2 B ≔ B ∪ { x }
 end
```

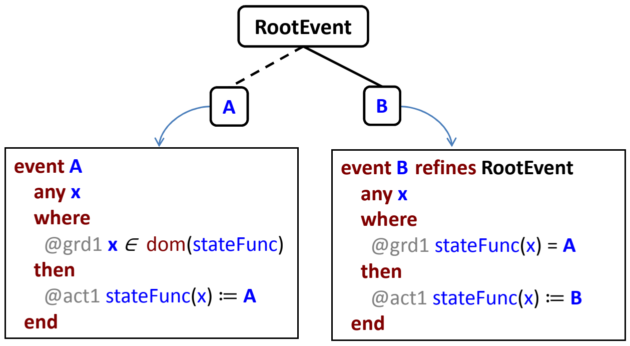
Figure 4.12: Disjoint Sets

Another alternative is to use function, Figure 4.13. A set represents possible states of a parameter, and a function shows a relation between a parameter and its state:

$STATES = \{A, B\}$
$stateFun : PAR\_SET \rightarrow STATE$

Each event change the value of $stateFunc$ to a new value.

A state machine in UML-B can be encoded in Event-B using disjoint sets representation or state function representation [61]. This two styles are introduced in [44].



```
event A
  any x
  where
    @grd1 x ∈ dom(stateFunc)
  then
    @act1 stateFunc(x) ≔ A
  end
```

```
event B refines RootEvent
  any x
  where
    @grd1 stateFunc(x) = A
  then
    @act1 stateFunc(x) ≔ B
  end
```

Figure 4.13: State Function

In [24], the subset approach is used. We adopted the subset approach as well. One of the advantages of using the subset relationships in the Event-B models, is that the subset relationships between the control variables that represent different states of the model can be specified in the invariants of the model. Considering Figure 4.11, invariant ($B \subseteq A$) specifies the ordering relationship between A and B control variables. This ensures that the orderings are upheld in the Event-B model more strongly than if specified only in the event guards.

Moreover, having disjoint set variables would not allow us to model the and-constructor, the or-constructor, the all-replicator and one-replicator in a simple way as subset variables provide. Considering the and-constructor and the or-constructor, a logical *and* or a logical *or* between two events, A and B, means four states as follows:

- none has happened
- A happened but not B
- B happened but not A
- A and B have happened

Using non-disjoint set variables (subset approach) allows us to model these combinations using two set variables, but disjoint set variables would not allow this by using only two set variables. Using disjoint set variables to model these combination would requires four state variables expilicitly. As a result the Event-B models of the and-constructor and the or-constructor corresponding to the disjoint set approach are larger and more complex comparing to the subset approach models.

Since the all-replicator and the some-replicator are generalisations of the and-constructor and the or-constructor respectively, having disjoint set variables make the same complexity in the corresponding Event-B models.

Considering the Event-B model of the and-constructor pattern presented in Table 4.3, using disjoint sets results in subtracting the parameter p from the set control variable *Event1*, in *AndEvent1*. Consequently, the ordering between *Event1* and the other child of and-constructor, *AndEvent2*, is not possible, since we can not track the execution of *Event1*. This is true for the or-constructor pattern presented in Table 4.4. Also considering the Event-B model of the all-replicator pattern in Table 4.6, using disjoint sets results in subtracting the parameter *p1* from set control variable *Event1* in the first execution of *AllEvent*. As a result, guard $(p1 \in Event1)$ does not hold for further executions of *AllEvent*. It is true for the some-replicator pattern presented in Table 4.7.

Using the function approach presented in Figure 4.13, can result in complex guards. For example in the and-constructor pattern presented in Table 4.3, guard $(p1 \in Event1)$ in *AndEvent1* is changed as follows:

$stateFunc(p) = Event1 \vee stateFunc(p) = AndEvent2$

The more constructor children there are, the more complex the guards.

4.5 Related Works and Comparison

The desire to explicitly model control flow is not restricted to Event-B. To address this issue usually a combination of two formal methods are suggested. A good example of such an approach is Circus [62, 63] combining CSP [37] and Z [64]. The combination of CSP and Classical B [9] has also been investigated in [65, 66].

To provide explicit control flow for an Event-B model a combination of two formal methods is presented in [67] which is based on using CSP alongside Event-B. Event-B is a state-based formalism, and as presented in Section 3.3, the control flow can only be implicitly modelled in state variables and event guards. On the other hand CSP is a process-based formalism (Section 2.3.4), which supports explicitly specifying control flow via processes. [67] presents an integrated formal method, a combination of Event-B as a state-based formalism and CSP as a control-based formalism, to explicitly model control flow in Event-B.

UML-B [61, 68] provides a “UML-like” graphical front-end for Event-B. It adds support for class-oriented and state machine modelling. State machines provide us with a graphical notation to explicitly define event sequencing. Events are represented by transitions on a state machine, and control flow is specified by defining the source and target state of each transition.

Another method to explicitly define control flow properties of an Event-B model is suggested in [69, 70]. This method extends Event-B models with expressions, called flows, defining event ordering. Flows are written in a language resembling those in process algebra.

A comparison between the atomicity decomposition approach and other techniques outlined above, is provided as follows:

- All outlined techniques only deal with explicit event sequencing; they do not support the explicit refinement relationship, provided by atomicity decomposition diagrams. The atomicity decomposition approach provides a graphical front-end to Event-B along with other features such as supporting explicit event sequencing and expressing refinement relationships between abstract and concrete events. Also it can be combined effectively with other techniques such as model decomposition [2]. The graphical front-end of the atomicity decomposition approach can provide an overall visualisation of the refinement structure, which is not supported by any of techniques outlined above.
- In integrated formal methods, the control flow constructs rely on the constructs in the process-based formalism of the integration. CSP constructs are used to model control flow in integrations of CSP and Z/B/Event-B. CSP constructs, which are outlined in Section 2.3.4, include prefix, deterministic choice, nondeterministic choice, parallel, interleaving, hiding and recursion.

  Atomicity decomposition control flow constructs are addressed in Chapter 4. Atomicity decomposition constructs contain the sequence construct, the loop construct, logical constructs, e.g. *and/or/xor*, and *all/some/one* constructs as generalisation of the *and/or/xor* constructs.

  The CSP constructs and the atomicity decomposition constructs can be compared as follows:

  – The prefix operator in CSP is used to describe the sequence of events and is equivalent to the sequence construct in the atomicity decomposition approach.
  – The choice operators in CSP are equivalent to the *xor* construct in the atomicity decomposition approach. We do not distinguish between deterministic and nondeterministic choice in the atomicity decomposition approach. The *one* construct in the atomicity decomposition approach is generalisation of the *xor* construct; the *one* construct is also supported in CSP.
  – The parallel operator is CSP is equivalent to the *all* construct in the atomicity decomposition approach. In the atomicity decomposition approach, the *all* construct is generalisation of the *and* construct; the *and* construct is also supported by parallel operator in CSP.
  – The interleaving operator is supported in CSP. Also in atomicity decomposition approach, different diagrams can be interleaved based on the Event-B interleaving.

  - CSP includes an event hiding operator. In the Event-B refinement, a new event introduced in a refining machine, may be considered as a hidden event in the abstract machine. In the atomicity decomposition approach, we decomposed the atomicity of an abstract event to new concrete events and a refining concrete event. The new events connected with dashed lines to the abstract event, are considered as hidden events in the abstract machine.
  - CSP supports recursion (which makes it possible to model loops). Atomicity decomposition supports loops but not recursion.
  - There is no equivalences for the *or* construct and the *some* construct (as generalisation of *or*) of the atomicity decomposition approach, in CSP. Recalling *or* construct in Section 4.2.5, in (A *or* B), one or both may occur which is different to choice and different to interleaving.

  The flow language presented in [69, 70] is based on process algebra. The flow language constructs contain sequential composition, parallel composition, choice and loop.

  Control flow in Event-B can be modelled in state machine supported by UML-B [61, 68]. Sequencing, choice and loop can be encoded in state machines, state machines do not have explicit constructs for these. State machines have explicit constructs for parallel regions. The *or* construct and the *some* construct (as generalisation of *or*) of the atomicity decomposition approach, are not supported in UML-B state machine.

- As explained in Section 2.4.6, a Classical B operation can be called by other operations. It is the responsibility of the caller to ensure that the called operation pre-conditions are hold. While in Event-B, an event contain guards and the enabled events are continually executed in a nondeterministic manner.

  In the integration of CSP and classical B presented in [65], classical B operations are called with CSP description. CSP description allows us to make sure that pre-conditions of called operations hold. In the integration of CSP and Event-B presented in [67], the authors do not need to deal with pre-conditions, as Event-B events contain guards rather than preconditions.

- In the integration of CSP and Event-B technique presented in [67], the authors need to tackle the verification of combined specifications. While in the atomicity decomposition approach and UML-B state machines the graphical representation is directly transformable to the Event-B formalism. This in turn means that verification effort can be carried out in the existing Event-B tool-set, Rodin, which is already familiar to the Event-B users. Also in the combined CSP with classical B approach presented in [66], CSP specifications are converted into standard B specifications.

- As [67] suggests, in combining formal method descriptions we may not be able to express all invariants as state predicates; because the control flow requirements are separated in a process-based description. While in the atomicity decomposition approach, control flow requirements are translated into Event-B; and Event-B invariants have access to all state variables in one place, the Event-B model.

4.6 Conclusion

Several atomicity decomposition constructors, which were discovered during case study developments, have been presented in this chapter. A pattern-based style was used to present the atomicity decomposition constructors. Each pattern is defined to satisfy a particular intention in decomposing the atomicity of an abstract event, and contains one constructor in a single level of refinement. Each pattern is encoded in terms of Event-B using some variables, invariants, events, guards and actions. The diagrammatic notation of a constructor and corresponding encoded Event-B model are presented both for single instance (SI) execution of an event and multiple instance (MI) execution.

In total eight constructors were presented as follows:

- The intention to model a sequential execution of two or more events is represented by the Sequence pattern.
- The Loop pattern represents zero or more execution of an event.
- The logical constructor patterns (and-constructor, or-constructor and xor-constructor) model a logical execution between two or more events.
- The replicator patterns, all-replicator, some-replicator and one-replicator, are generalisations of the logical constructor patterns, and-constructor, or-constructor and xor-constructor, respectively.

Each pattern contains three children in decomposition of an abstract event in one refinement level. In all patterns, except the sequence pattern, the middle sub-event is a loop or a logical constructor or a replicator. From a more general and formal point of view, the combination of constructors in one or more refinement levels is presented in Chapter 5. The patterns presented in this chapter help to aid understanding of the contents of Chapter 5.

Chapter 5

Atomicity Decomposition Part 3 - Language Description and Translation Rules

5.1 Introduction

In Chapter 4, several atomicity decomposition patterns have been outlined. The atomicity decomposition language needs to be described in a more general and formal way. This chapter addresses this; instead of the patterns described in Chapter 4 in one level of refinement, we consider all possible combination of patterns in one or more refinement level(s). In other words, different patterns can be applied in one refinement level.

In this chapter we begin by presenting an example of an atomicity decomposition diagram in several refinement levels including different types of atomicity decomposition constructors. Later this example is used to help explain the language description and translation rules. Section 5.3 presents a formal description of the syntax of the atomicity decomposition language. Then Section 5.4 is dedicated to translation rules which describe the transformation from the atomicity decomposition language to the Event-B notation. In this chapter, we use the abbreviation "ADL" to stand for the Atomicity Decomposition Language.

5.2 An Example

In Section 4.2, we presented each atomicity decomposition constructor in one pattern and in one refinement level. In this section we present an instance of an atomicity decomposition diagram combining different constructors and including an abstract level and two refinement levels, in Figure 5.1.

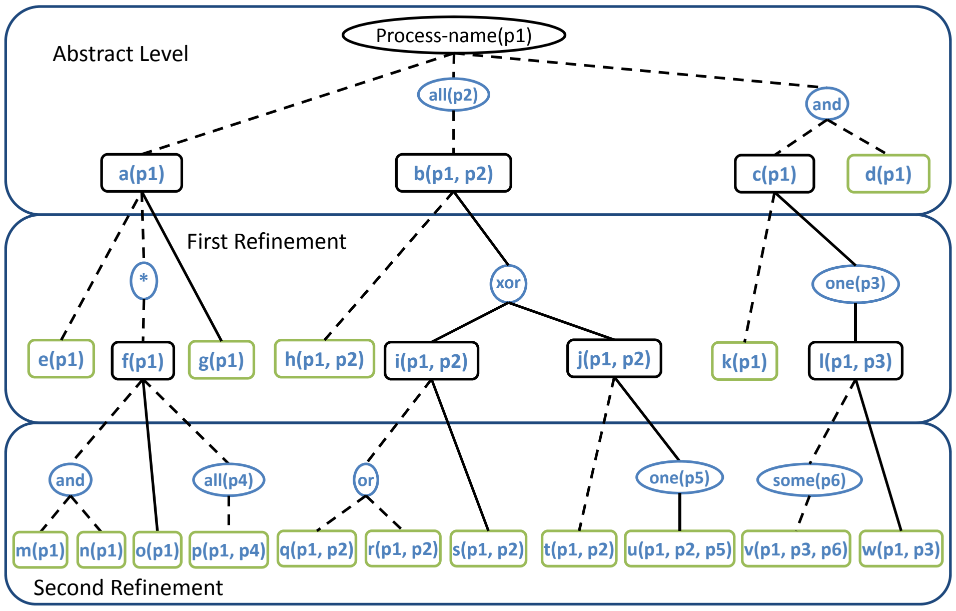


Figure 5.1: An Example of Atomicity Decomposition Diagram

In the most abstract level, there are four abstract events, *a*, *b*, *c* and *d*. The diagram indicates the sequencing between these events. First event *a(p1)* executes, then event *b(p1, p2)* for all instances of parameter *p2*, finally event *c(p1)* and *d(p1)* executes in any order. In first refinement level three events, *a*, *b* and *c*, are decomposed to some sub-events. And in the second refinement level there are four further atomicity decomposition. The green leaves present the events in the final refinement level (second refinement level). These events are leaf nodes (nodes that does not have any children).

In the later sections this example will be followed to explain the language description and translation rules to Event-B. The selection of constructors and their combination in this example is chosen in a way that it covers all cases of transformation to the Event-B.

5.3 Atomicity Decomposition Language Specification

To describe the language syntax, we adopt Augmented Backus-Naur Form (ABNF) [71]. ABNF is a metalanguage based on Backus-Naur Form (BNF). BNF is a notation for context-free grammars, often used to describe the syntax of languages. It is applied wherever exact descriptions of languages are needed. The differences between standard BNF and ABNF involve naming rules, repetition, alternatives, order-independence, and value ranges. In describing ADL, the repetition syntax in ABNF seems more suitable than in standard BNF.

An ABNF specification is a set of derivation rules, written as

$rule = definition$

The ABNF rules for ADL is shown in Figure 5.2. The following ABNF operators are used in describing ADL:

- Terminal values:
  Terminal values are placed between two apostrophes (“Terminal”).
- Alternative: (Rule1 / Rule2)
  A rule may be defined by a list of alternative rules separated by a solidus (“/”).
- Variable repetition: (n*m element)
  To indicate repetition of an element the form *(n*m element)* is used. The optional n gives the minimum number of elements to be included with the default of 0. The optional m gives the maximum number of elements to be included with the default of infinity.

  We use *element for zero or more elements, 1*element for one or more elements and 2*element for two or more elements.

```
flow          =  "flow" (name, *par,  sw) ( 1*child (ref) )

child         =  "leaf" (name)  /  constructor  /  1* flow
cons-child    =  "leaf" (name)  /  1* flow

constructor   =   ("and" / "or" / "xor")  ( 2* cons-child )
              /   ("all" / "some" / "one") (par) ( cons-child )
              /   "loop" ( cons-child )
```

Figure 5.2: Syntax of Atomicity Decomposition Language (ADL)

A flow refers to a single atomicity decomposition for a root node. To describe the refining and non-refining sub-events, we consider a boolean property, called “ref”. The refining and non-refining sub-events in an atomicity decomposition diagram are presented by type of lines, solid lines and dashed lines respectively. When a sub-event refines the abstract event (solid line) , “ref” is one; otherwise “ref” is zero. Also to distinguish strong sequencing flow from a weak sequencing flow, another boolean property, called “sw”, is used. When a flow has strong sequencing, “sw” is one, otherwise “sw” is zero.

Considering Figure 5.2, the ABNF for ADL may be described informally as follows:

- A flow consists of a name, zero or more parameters, an “*sw*” property, followed by one or more children. Each child of a flow has a “*ref*” property.
- A child is either a “*leaf*” with a name, or a constructor or one or more flow(s).
- A constructor is either an “*and*” or an “*or*” or a “*xor*”, with two or more constructor children (cons-child) or an “*all*” or a “*some*” or an “*one*” with a parameter,

followed by one constructor child (cons-child) or a "*loop*" with one constructor child (cons-child).

- A cons-child is either a "*leaf*" with a name or one or more flow(s).

There are some properties of the syntax of the ADL which reflect some features of atomicity decomposition diagrams, which have been discussed in Chapter 4. These properties are listed below:

- Since a most abstract flow has always strong sequencing, Section 4.3.1, the "sw" property is always one for an abstract flow. Also since the children of a most abstract flow are always non-refining, come with dashed lines, Section 4.3.4, so the "ref" property for its children is always zero.
- A cons-child inherits the value of the "ref" property from its constructor parent.
- Since some constructors including "*and*", "*or*", "*all*", "*some*" and "*loop*" always come with dashed lines, Section 4.2, the value of the "ref" property for these constructors is always zero. Whereas other constructors including '*xor*" and "*one*" can come with dashed or solid lines, therefore the "ref" property for them can be zero or one.
- One and only one of the children of each flow can refine the root event, as explained in Section 4.2, therefore in ABNF one and only one of the "ref" property of children of a flow is allowed to be one.
- Each flow inherits its parameters from its parent flow plus its constructor parent if exists.
- There can be more than one atomicity decomposition for a single event, as explained in Section 4.3.3. This feature is specified by (1*flow). All flows in a collection of (1*flow) should have same name, since they all show decomposition of the same event.

Considering example in Figure 5.1, the ABNF for each refinement level is presented separately in Figure 5.3. Although the diagram in Figure 5.1 does not indicate if sequencing of each flow is strong or weak, the ABNFs in Figure 5.3 presents this as a property of each flow.

The syntax definition of ADL prevents us from combining constructors at a single refinement level, e.g., the diagram presented in Figure 5.4, is not allowed. There are some reasons for this limitation. First, baseed on our experience during the case study developments, we have not seen the need to support a combination of constructors at single refinement level. Moreover, the atomicity decomposition approach is considered as a technique to partly solve the complexities of the Event-B modelling of large systems; therefore we try to keep the syntax definition as simple as it solves our requirements.

Abstract Level:

flow(process-name, p_1, 1) (leaf (a) (0), all (p2) (leaf (b)) (0), and (leaf (c), leaf (d)) (0))

1st Refinement Level:

flow(process-name, p_1, 1) (
flow(a, p_1, 1) (leaf (e) (0), loop (leaf (f)) (0), leaf (g) (1)) (0),
all (p_2) (flow(b, (p_1, p_2), 1) (leaf (h) (0), xor (leaf (i), leaf (j)) (1))) (0),
and (flow(c, p_1, 0) (leaf (k) (0), one (p_3) (leaf (l)) (1)), leaf (d)) (0))

2nd Refinement Level:

flow (process-name, p_1, 1) (
flow (a, p_1, 1) (leaf (e) (0), loop (flow(f, p_1, 1) (and (leaf (m), leaf (n)) (0), leaf (o) (1), all (p4) (leaf (p)) (0))) (0), leaf (g) (1)) (0),

all (p_2) (flow(b, (p_1, p_2), 1) (leaf (h) (0), xor (flow(i, (p_1, p_2), 1) (or (leaf (q), leaf (r)) (0) , leaf (s) (1)), flow(j, (p_1, p_2), 0) (leaf (t), one (p_5) (leaf(u)) (1))) (1))) (0),

and (flow(c, p_1, 1) (leaf (k) (0), one (p_3) (flow(l, (p_1, p_3), 1) (some (p_6) (leaf (v)) (0), leaf(w) (1))) (1)), leaf (d)) (0))

Figure 5.3: ABNF of the Diagram in Figure 5.1

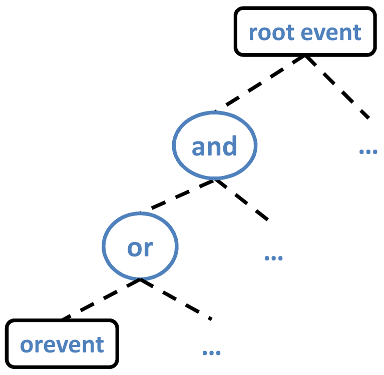


Figure 5.4: Invalid Combination of the Constructors

5.4 Atomicity Decomposition Translation Rules (TRs) to Event-B

5.4.1 Introduction

This section describes the translation rules formally. We outline how the ABNF of ADL is encoded in the Event-B language. In total 23 rules are presented. Most of these rules have been introduced informally in Section 4.2.

In the figure of each rule, the first row shows the signature of the rule, the second row presents the source element(s) of the rule, the ABNF element(s), and the last row(s) present the destination element(s) of the rule, the Event-B element(s). Each rule signature is of the form (ABNF element(s) $\rightarrow$ Event-B element(s)). There are some auxiliary functions which are presented in Section 5.4.2. The aim of defining these functions is to help describing some of the translation rules.

Considering atomicity decomposition patterns in Section 4.2, patterns are encoded in the Event-B modelling using control variables, invariants, events, guards and actions. These Event-B elements are transformed from four sources in the atomicity decomposition diagram: a leaf, the xor-constructor, the one-constructor and the loop constructor. A leaf is transformed to a variable, an invariant, an event, guard(s) and an action in order to manage the sequencing between events and to show the relationship between the abstract event and the refining sub-event. The xor-constructor is transformed to an invariant and guards to specify the mutual exclusive property of its children. The one-replicator is transformed to an invariant and a guard to limit the number of executions of its child to one. The loop constructor is transformed to a guard and a resetting event. Moreover as presented in Section 4.3.4, a weak sequencing flow is managed with sequencing invariant(s) and sequencing guard(s) in the Event-B model.

The translation rules are categorised according to their source element. The rules whose source is a leaf are presented in Section 5.4.3. The rules whose source is the xor-constructor are presented in Section 5.4.4. The rules whose source is the one-replicator are presented in Section 5.4.5. The rules whose source is the loop constructor are presented in Section 5.4.6. Finally the rules whose source is a weak flow is presented in Section 5.4.7. It is helpful to mention that the and-constructor, the or-constructor, the all-replicator and some-replicator properties are specified in sequencing invariants and sequencing guards which are generated in TR_leaf4 (Section 5.4.3.4) and TR_leaf8 (Section 5.4.3.8) respectively.

In the atomicity decomposition patterns (Section 4.2), the invariants and guards transformed from the xor-constructor, the one-constructor and the loop constructor are labelled with "_xor" suffix, "_one" suffix and "_loop" suffix respectively. Sequencing invariants and the sequencing guards are labelled with "_seq" suffix. And typing invariants and gluing invariants are labelled with "_type" suffix and "_gluing" suffix respectively. This labelling protocol helps to determine the aim of each encoded invariant or guard. The labelling protocol is followed in the translation rules as well.

Translation rules are presented per ABNF element. For each ABNF element, we present the resulting variables, events, guards, actions and invariants. We assume that we access to each ABNF element in an ABNF description of an atomicity decomposition diagram.

The translation rules are presented in a *modular* way to be encoded in the Event-B model. For example the events are generated in TR_leaf6 (Section 5.4.3.6) and TR_leaf7 (Section 5.4.3.7), and later other translation rules, e.g. TR_leaf8 (Section 5.4.3.8), TR_leaf9 (Section 5.4.3.9), TR_leaf10 (Section 5.4.3.10), TR_leaf11 (Section 5.4.3.11) and TR_leaf12 (Section 5.4.3.12), add the guards and actions to the generated events.

In some similar rules, the translations for a replicator (all-replicator, some-replicator, one-replicator) leaf and a non-replicator leaf are distinguished. This difference is applied because of the extra parameter that the replicator adds to the parameter list of its leaf. This replicator parameter changes the type of the replicator leaf variable. For instance, a typing invariant is generated for a non-replicator leaf in TR_leaf2 (Section 5.4.3.2), and for a replicator leaf in TR_leaf3 (Section 5.4.3.3).

The example that has been presented in Figure 5.1 and Figure 5.3, will be used to show the application of each translation rule.

5.4.2 Auxiliary Functions Definitions

5.4.2.1 Traversing Functions

Some of the translation rules are applied to an ABNF element placed in the the **final level of refinement** in a combined atomicity decomposition diagram. Some other of the translation rules cover translations from an ABNF element in the **earlier refinement level(s)** in a combined atomicity decomposition diagram. In the later translation rules we need to traverse down the subtree of a child in order to find leaves in the final refinement level. Some functions are defined in order to traverse the sub-trees in a combined atomicity decomposition diagram. We use the outputs of these functions to create invariants and guards as the destination element of the translation rules.

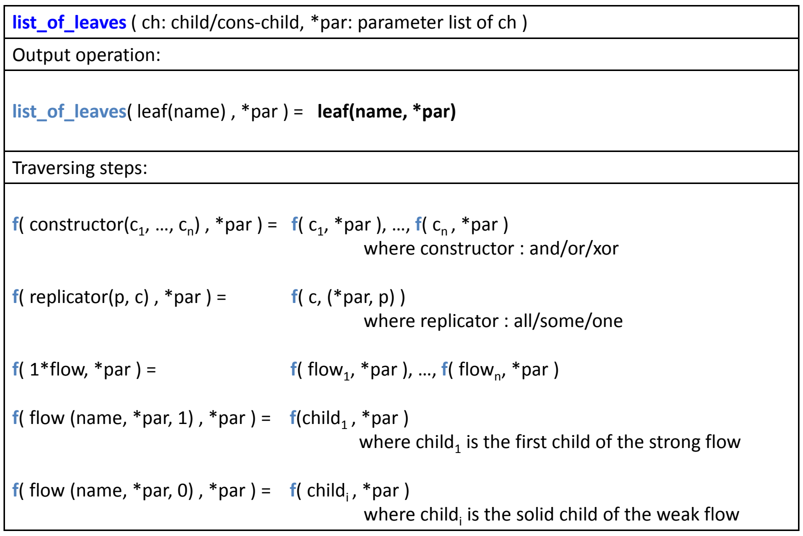
In total six functions are defined. The functions are summarised as follows:

- *list_of_leaves* function is presented in Figure 5.5. The function name, *list_of_leaves*, in the traversing steps is abbreviated to *f*. It is a recursive function that outputs a list of the leaf events, including their names and parameters.
- *disjunction_of_leaves* function is presented in Figure 5.6. The function name, *disjunction_of_leaves*, in the traversing steps is abbreviated to *f*. It is a recursive function that computes a predicate representing the disjunction of the invariants of the leaf events.
- *conjunction_of_leaves* function is presented in Figure 5.7. The function name, *conjunction_of_leaves*, in the traversing steps is abbreviated to *f*. It is a recursive function that computes a predicate representing the conjunction of the guards of the leaf events.
- *union_of_leaves* function is presented in Figure 5.8. The function name, *union_of_leaves*, in the traversing steps is abbreviated to *f*. It is a recursive function that computes a predicate representing the union of the leaf events. *domain* function may be applied to each output leaf event, for n times; where n is the number of existing replicators in the traversing steps.

- *build_seq_inv* function is presented in Figure 5.9. The function name, *build_seq_inv*, in the traversing steps is abbreviated to *f*. It is a recursive function that computes an invariant predicate specifying the sequencing between two leaf events. This function calls another function for the leaf events, to compute the invariant. The inner function is presented in the next section.

- *build_seq_grd* function is presented in Figure 5.10. The function name, *build_seq_grd*, in the traversing steps is abbreviated to *f*. It is a recursive function that computes a guard predicate specifying the sequencing between two leaf events. This function calls another function for the leaf events, to compute the guard. The inner function is presented in the next section.

In the traversing functions, the first or the last child of an input flow is selected; and the selected child name is acted as a variable name. Since we do not consider a variable for a loop (Loop Pattern 4.2.3), we assume that a loop is never placed as the first or the last child of a flow.



list_of_leaves (ch: child/cons-child, *par: parameter list of ch)

Output operation:

list_of_leaves(leaf(name) , *par) = **leaf(name, *par)**

Traversing steps:

f(constructor(c_1, …, c_n) , *par) = f(c_1, *par), …, f(c_n , *par)
where constructor : and/or/xor

f(replicator(p, c) , *par) = f(c, (*par, p))
where replicator : all/some/one

f(1*flow, *par) = f($flow_1$, *par), …, f($flow_n$, *par)

f(flow (name, *par, 1) , *par) = f($child_1$, *par)
where $child_1$ is the first child of the strong flow

f(flow (name, *par, 0) , *par) = f($child_i$, *par)
where $child_i$ is the solid child of the weak flow

Figure 5.5: *list_of_leaves* Function

```
disjunction_of_leaves (ch: child/cons-child, parnum: int)
Output operations:

disjunction_of_leaves( leaf(name), parnum ) =  name          where parnum = 0
disjunction_of_leaves( leaf(name), parnum ) =  name ≠ $\emptyset$      where parnum > 0

Traversing Steps:

f( constructor($c_1$, ..., $c_n$), parnum ) =  f( $c_1$ , parnum ) $\vee$ ... $\vee$ f( $c_n$ , parnum )
                                                  where constructor : and/or/xor

f( replicator(par, c), parnum ) =         f( c, parnum+1 )    where replicator : all/some/one

f( 1*flow, parnum )=                      f( $flow_1$ , parnum ) $\vee$ ... $\vee$ f( $flow_n$ , parnum )

f( flow(name, *par, 1), parnum ) =        f( $child_1$ , parnum )
                                              where $child_1$ is the first child of the strong flow

f( flow(name, *par, 0), parnum ) =        f( $child_i$ , parnum )
                                              where $child_i$ is the solid child of the weak flow
```

Figure 5.6: *disjunction_of_leaves* Function

```
conjunction_of_leaves ( ch: child/cons-child, parnum: int )
Output operations:

conjunction_of_leaves( leaf(name), parnum ) =  name  = FALSE        where parnum=0
conjunction_of_leaves( leaf(name) , parnum ) =  name = $\emptyset$              where parnum>0

Traversing steps:

f( constructor($c_1$, ..., $c_n$) , parnum ) =  f( $c_1$, parnum ) $\wedge$ ... $\wedge$ f( $c_n$, parnum )
                                                  where constructor : and/or/xor

f( replicator(par, c) , parnum ) =        f( c, parnum+1 )    where replicator : all/some/one

f( 1*flow, parnum ) =                     f( $flow_1$ , parnum ) $\wedge$ ... $\wedge$ f( $flow_n$ , parnum )

f( flow(name, *par, 1), parnum ) =        f( $child_1$, parnum )
                                              where $child_1$ is the first child of the strong flow

f( flow(name, *par, 0), parnum ) =        f( $child_i$, parnum )
                                              where $child_i$ is the solid child of the weak flow
```

Figure 5.7: *conjunction_of_leaves* Function

```
union_of_leaves (ch: child/cons-child, n: int)
Output operation:

union_of_leaves( leaf(name), n) =  $dom_1$( ... $dom_n$ (name) ...)

Traversing steps:

f( constructor($c_1$, ..., $c_n$), n) =   f( $c_1$ , n) $\cup$ ... $\cup$ f( $c_n$ , n)    where constructor : and/or/xor

f( replicator(par, c), n) =           f( c, n+1)                 where replicator : all/some/one

f( 1*flow, n) =                       f( $flow_1$ , n) $\cup$ ... $\cup$ f( $flow_n$ , n)

f( flow(name, *par, 1), n) =          f( $child_1$ , n)        where $child_1$ is the first child of the strong flow

f( flow(name, *par, 0), n) =          f( $child_i$ , n)        where $child_i$ is the solid child of the weak flow
```

Figure 5.8: *union_of_leaves* Function

build_seq_inv (predecessor: child/cons-child, *par1: parameter list of predecessor, l: leaf, *par2: parameter list of l)	
Output operation:	
build_seq_inv(leaf, *par1, l, *par2) =	seq_inv**(leaf, *par1, l, *par2)**
Traversing steps:	
f(and(c_1, …, c_n), *par1, l, *par2) =	f(c_1, *par1, l, *par2) ∧ … ∧ f(c_n, *par1, l, *par2)
f(or/xor(c_1, …, c_n), *par1, l, *par2) =	f(c_1, *par1, l, *par2) ∨ … ∨ f(c_n, *par1, l, *par2)
f(replicator(p, c), *par1, l, *par2) =	f(c, (*par1, p) , l, *par2) where replicator : all/some/one
f(1*flow, *par1, l, *par2) =	f($flow_1$, *par1, l, *par2) ∨ … ∨ f($flow_n$, *par1, l, *par2)
f(flow (… , 1), *par1, l, *par2) =	f($child_i$, *par1, l, *par2) where $child_i$ is the last child of the flow
f(flow (… , 0), *par1, l, *par2) =	f($child_i$, *par1, l, *par2) where $child_i$ is the solid child of the flow

Figure 5.9: *build_seq_inv* Function

build_seq_grd (predecessor: child/cons-child, *par1: parameter list of predecessor, l: leaf, *par2: parameter list of l)	
Output operation:	
build_seq_grd(leaf, *par1, l, *par2) =	seq_grd**(leaf, *par1, l, *par2)**
Traversing steps:	
f(and(c_1, …, c_n), *par1, l, *par2) =	f(c_1, *par1, l, *par2) ∧ … ∧ f(c_n, *par1, l, *par2)
f(or/xor(c_1, …, c_n), *par1, l, *par2) =	f(c_1, *par1, l, *par2) ∨ … ∨ f(c_n, *par1, l, *par2)
f(replicator(p, c), *par1, l, *par2) =	f(c, (*par1, p) , l, *par2) where replicator : all/some/one
f(1*flow, *par1, l, *par2) =	f($flow_1$, *par1, l, *par2) ∨ … ∨ f($flow_n$, *par1, l, *par2)
f(flow (… , 1), *par1, l, *par2) =	f($child_i$, *par1, l, *par2) where $child_i$ is the last child of the flow
f(flow (… , 0), *par1, l, *par2) =	f($child_i$, *par1, l, *par2) where $child_i$ is the solid child of the flow

Figure 5.10: *build_seq_grd* Function

5.4.2.2 Functions to Build Sequencing Invariants/Guards

Recall from Section 4.2 that the sequencing between events is managed with guards and the sequencing properties are specified with invariants. In Section 4.2, sequencing is defined between events with the same parent. Therefore the parameters of two sequential events were always the same (inherits from their parent). In the case of replicators, the replicator event had one more replicator parameter. Having the same parameters has made building of the sequencing invariants and guards easy.

Whereas in a combined atomicity decomposition diagram, two sequential events can be from a different parent, illustrated in Figure 5.11. A leaf from the $(i+1)^{th}$ child, *e2*, may execute only after execution of a leaf from the i^{th} child, *e1*. The leaves parameters can be different due to different possible replicators in each child. Assume leaf $e1$ parameter list contains $(p^1_1, ..., p^1_n)$, and leaf $e2$ parameter list contains $(p^2_1, ..., p^2_m)$. Some of their parameters which come from their common parent flow may be same, $(p_1, ..., p_i)$. The same parameters are always the first parameters in the parameter list, since each replicator parameter is added to the end of the parameter list. Two functions are defined to build the sequencing guard and invariants. Definitions of X, Y, Z, W and K in Figure 5.11, are used in defining the functions.

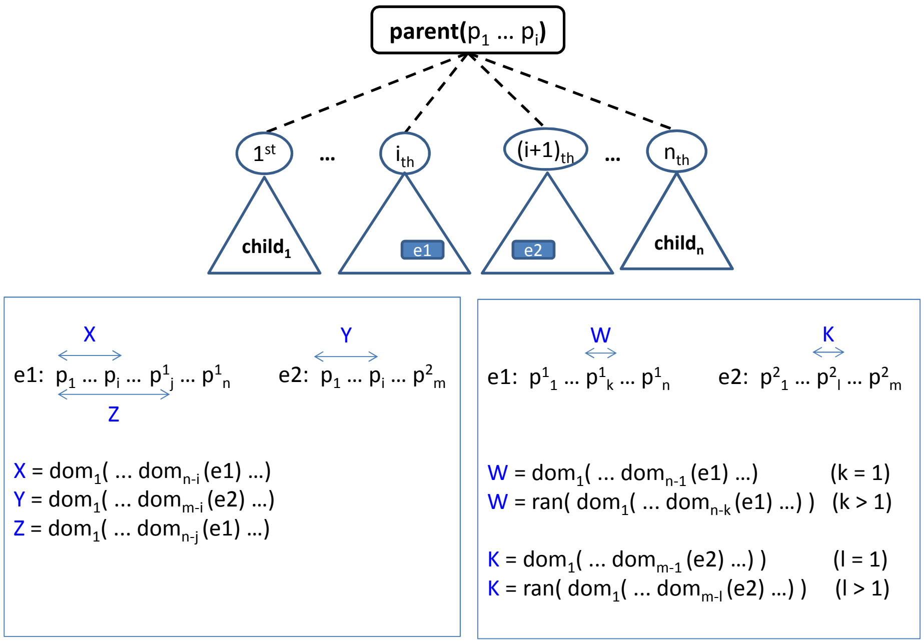

parent(p_1 … p_i)
1^{st}
…
i_{th}
$(i+1)_{th}$
…
n_{th}
$child_1$
e1
e2
$child_n$
X
Y
e1: p_1 … p_i … p^1_j … p^1_n
e2: p_1 … p_i … p^2_m
Z
X = dom_1(… dom_{n-i}(e1) …)
Y = dom_1(… dom_{m-i}(e2) …)
Z = dom_1(… dom_{n-j}(e1) …)
W
K
e1: p^1_1 … p^1_k … p^1_n
e2: p^2_1 … p^2_l … p^2_m
W = dom_1(… dom_{n-1}(e1) …) (k = 1)
W = ran(dom_1(… dom_{n-k}(e1) …)) (k > 1)
K = dom_1(… dom_{m-1}(e2) …)) (l = 1)
K = ran(dom_1(… dom_{m-l}(e2) …)) (l > 1)

Figure 5.11: Sequencing Between Two Leaf Events

The *seq_inv* and *seq_grd* functions are presented in Figure 5.12 and Figure 5.13 respectively. To generate the sequencing invariants and guards, we need to determine the possible same parameters from the common parent flow. The possible all-replicator parameters of *e1* have to be determined, since the all-replicator affects the guard of the next event, *e2*, and the sequencing invariants (all-replicator Pattern 4.2.7).

seq_inv (e1: leaf, $p_1 \dots p_n$: parameter list of e1,
e2: leaf, $p_1 \dots p_m$: parameter list of e2) =

- $e2 = TRUE \Rightarrow e1 = TRUE$ where $(n = 0)$ and $(m = 0)$
- $e2 \neq \emptyset \Rightarrow e1 = TRUE$ where $(n = 0)$ and $(m \neq 0)$
- $e2 = TRUE \Rightarrow e1 \neq \emptyset$ where $(n \neq 0)$ and $(m = 0)$ and
  (there is no all-replicator parameter in $(p_1 \dots p_n)$)
- $e2 = TRUE \Rightarrow W = TYPE(p_k)$ where $(n \neq 0)$ and $(m = 0)$ and
  (p_k is an all-replicator parameter $(1 \leq k \leq n)$)
- $e2 \neq \emptyset \Rightarrow e1 \neq \emptyset$ where $(n \neq 0)$ and $(m \neq 0)$ and
  (there is no common parent parameter) and
  (there is no all-replicator parameter in $(p_1 \dots p_n)$)
- $e2 \neq \emptyset \Rightarrow W = TYPE(p_k)$ where $(n \neq 0)$ and $(m \neq 0)$ and
  (there is no common parent parameter) and
  (p_k is an all-replicator parameter $(1 \leq k \leq n)$) and
  (there is no parameter in $(p_1 \dots p_m)$ with same type as p_k)
- $K \subseteq W$ where $(n \neq 0)$ and $(m \neq 0)$ and
  (there is no common parent parameter) and
  (p_k is an all-replicator parameter $(1 \leq k \leq n)$) and
  ($type(p_k) = type(p_l)$ $(1 \leq l \leq m)$)
- $Y \subseteq X$ where $(n \neq 0)$ and $(m \neq 0)$ and
  ($p_1 \dots p_i$ is list of common parent parameter) and
  (there is no all-replicator parameter in$(p_{i+1} \dots p_n)$)
- $p_1 \mapsto \dots \mapsto p_i \in e2 \Rightarrow Z[\{p_1 \mapsto \dots \mapsto p_{j-1}\}] = TYPE(p_j)$

  where $(n \neq 0)$ and $(m \neq 0)$ and
  ($p_1 \dots p_i$ is list of common parent parameter) and
  (p_j is an all-replicator parameter $(i+1 \leq j \leq n)$)

Figure 5.12: *seq_inv* Function

5.4.2.3 Predecessor/Successor Functions

In some of the translation rules we need to find the predecessor or successor of a subtree. Considering Figure 5.14, the predecessor of a subtree which is the i^{th} child of a flow, is its left subtree which is the $(i-1)^{th}$ child of that flow. If the i^{th} child is the first child of a flow then the predecessor of the i^{th} child is the predecessor of its parent flow.

The predecessor and successor functions are presented in Figure 5.15 and Figure 5.16 respectively. *predecessor* function is used to find the previous node of a leaf to create the sequencing invariants and guards, in TR_leaf4 (Section 5.4.3.4), TR_leaf8 (Section 5.4.3.8), TR_weak1 (Section 5.4.7.1) and TR_weak2 (Section 5.4.7.2). *successor* function is used to find the next node of a loop to create the loop guard, in TR_loop1 (Section 5.4.6.1) and TR_loop2 (Section 5.4.6.2).

Since we do not consider a variable for a loop (Loop Pattern 4.2.3), we move over the loop in both functions.

seq_grd (e1: leaf, $p_1 \ldots p_n$: parameter list of e1,
e2: leaf, $p_1 \ldots p_m$: parameter list of e2) =

• $e1 = \text{TRUE}$	where $(n = 0)$
• $e1 \neq \emptyset$	where $(n \neq 0)$ and $(m = 0)$ and (there is no all-replicator parameter in $(p_1 \ldots p_n)$)
• $W = \text{TYPE}(p_k)$	where $(n \neq 0)$ and $(m = 0)$ and (p_k is an all-replicator parameter $(1 \leq k \leq n)$)
• $e1 \neq \emptyset$	where $(n \neq 0)$ and $(m \neq 0)$ and (there is no common parent parameter) (there is no all-replicator parameter in $(p_1 \ldots p_n)$)
• $W = \text{TYPE}(p_k)$	where $(n \neq 0)$ and $(m \neq 0)$ and (there is no common parent parameter) and (p_k is an all-replicator parameter $(1 \leq k \leq n)$) and (there is no parameter in $(p_1 \ldots p_m)$ with same type as p_k)
• $p_l \subseteq K$	where $(n \neq 0)$ and $(m \neq 0)$ and (there is no common parent parameter) and (p_k is an all-replicator parameter $(1 \leq k \leq n)$) and ($\text{type}(p_k) = \text{type}(p_l)$ $(1 \leq l \leq m)$)
• $p_1 \mapsto \ldots \mapsto p_i \subseteq X$	where $(n \neq 0)$ and $(m \neq 0)$ and ($p_1 \ldots p_i$ is list of common parent parameter) and (there is no all-replicator parameter in$(p_{i+1} \ldots p_n)$)
• $Z[\{p_1 \mapsto \ldots \mapsto p_{j-1}\}] = \text{TYPE}(p_j)$	where $(n \neq 0)$ and $(m \neq 0)$ and ($p_1 \ldots p_i$ is list of common parent parameter) and (p_j is an all-replicator parameter $(i+1 \leq j \leq n)$)

Figure 5.13: *seq_grd* Function

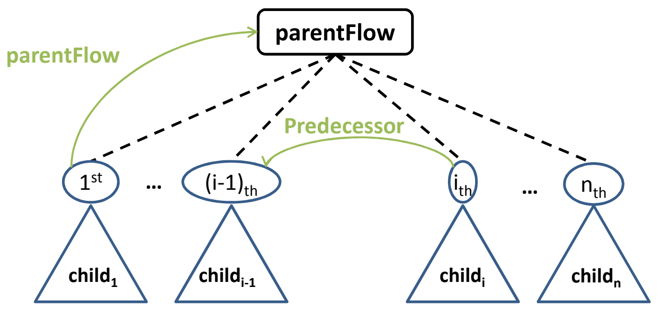


Figure 5.14: Predecessor of a Subtree

predecessor ($child_i$: child/cons-child, *par: list of parameter(s), sw: boolean) =

• **($child_{i-1}$, *par)**	where (i > 1) and ($child_{i-1} \neq$ loop)
• predecessor($child_{i-1}$, *par, sw)	where (i > 1) and ($child_{i-1}$ = loop)
• **"no predecessor"**	where (i = 1) and (sw = 0)
• **"no predecessor"**	where (i = 1) and (sw = 1) and (parentFlow($child_i$) is an abstract flow)
• predecessor(parent($child_i$), *par, sw)	where (i = 1) and (sw = 1) and (parentFlow($child_i$) is not a (all/some/one) child)
• predecessor(parent($child_i$), *par / p, sw)	where (i = 1) and (sw = 1) and (parentFlow($child_i$) is a (all/some/one)(p) child)

Figure 5.15: *predecessor* Function

Considering Figure 5.15, if the i^{th} child is the first child of an abstract flow (the most abstract level), then there is no predecessor of that child. An abstract flow in ABNF is indicated with $(sw = 1)$ and for all of its children $(ref = 0)$; whereas in a non abstract flow, there is always one child with $(ref = 1)$. If a child is the first child of a weak flow $(sw = 0)$, then we consider *no predecessor* for that child. Because there is no sequence constraint between the first child of a weak flow and the predecessor of it (Section 4.3.4).

successor ($child_i$: child/cons-child, parnum: int) =

• **($child_{i+1}$, parnum)**	where (i < n) and ($child_{i+1} \neq$ loop)
• successor($child_{i+1}$, parnum)	where (i < n) and ($child_{i+1}$ = loop)

Figure 5.16: *successor* Function

successor, presented in Figure 5.16, is used to find the next node of a loop to create the loop guard, in TR_loop1 (Section 5.53). n is the number of parent flow children (number of siblings of the input child). A loop is never placed as the first or the last child of a flow. Therefore in Figure 5.16, we do not consider the i^{th} child as the last child in *successor* function, (always $i < n$).

5.4.3 Translating a Leaf

5.4.3.1 TR_leaf1: mapping a leaf to a variable

A leaf is transformed to a variable with the same name, *leaf-name*, (Sequence Pattern 4.2.2). No variable is generated for a loop leaf, (Loop Pattern 4.2.3).

This translation rule is called TR_leaf1, presented in Figure 5.17. The first row in the figure is the signature of the rule; the second row presents the source element of the rule (ABNF element); and the last row shows the target element of the rule (Event-B element). In TR_leaf1, the source element is a leaf (not a loop leaf), and the destination element is a variable with the same name as the leaf name. The rules are applied to each matching sub term on the source element and each application of a rule adds a new element (e.g., variable) to the target model.

The *flow* that is presented as a part of the source element is the parent flow of the leaf. In this rule we do not need the properties of the parent flow, but in some of the other rules, which are described later, we use the parent flow properties for transformation. We aim to define the translation rules in a consistent way; therefore the parent flow is shown in all of the translation rules.

The invariant which defines the type of the generated variable is generated later in TR_leaf2 (Section 5.4.3.2), TR_leaf3 (Section 5.4.3.3) and TR_leaf4 (Section 5.4.3.4). All generated control variables are initialised to either false or to the empty set depending on the type of the control variable. The initialisation translation rule is omitted here since it is a trivial rule.

TR_leaf1: leaf $\longrightarrow$ variable

flow(parent-name, (p_1, ..., p_n), sw)(..., leaf (leaf-name)(ref), ...)

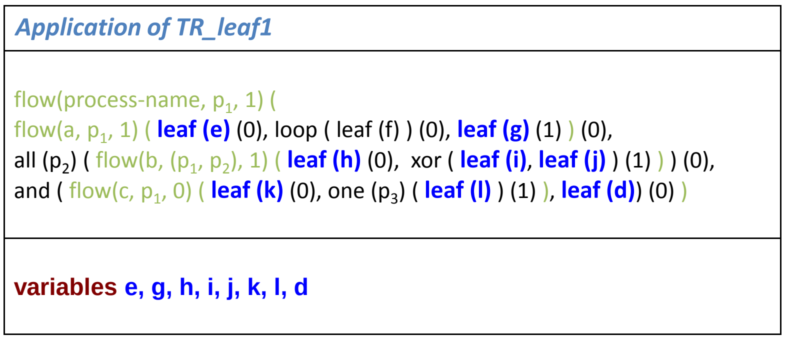
flow(parent-name, (p_1, ..., p_n), sw)(..., and (..., leaf(leaf-name), ...) (0), ...)
flow(parent-name, (p_1, ..., p_n), sw)(..., or (..., leaf(leaf-name), ...) (0), ...)
flow(parent-name, (p_1, ..., p_n), sw)(..., xor (..., leaf(leaf-name), ...) (ref), ...)

flow(parent-name, (p_1, ..., p_n), sw)(..., all (p_i, leaf(leaf-name)) (0), ...)
flow(parent-name, (p_1, ..., p_n), sw)(..., some (p_i, leaf(leaf-name))(0), ...)
flow(parent-name, (p_1, ..., p_n), sw)(..., one (p_i, leaf(leaf-name)) (ref), ...)

variables leaf-name

Figure 5.17: TR_leaf1: mapping a leaf to a variable

Figure 5.18 presents multiple applications of the rule in Figure 5.17 in the first refinement level of the example in Figure 5.1. There are eight leaves in the first refinement level, that each of them is transformed to a variable.



Application of TR_leaf1

flow(process-name, p_1, 1) (
flow(a, p_1, 1) (**leaf (e)** (0), loop (leaf (f)) (0), **leaf (g)** (1)) (0),
all (p_2) (flow(b, (p_1, p_2), 1) (**leaf (h)** (0), xor (**leaf (i)**, **leaf (j)**) (1))) (0),
and (flow(c, p_1, 0) (**leaf (k)** (0), one (p_3) (**leaf (l)**) (1)), **leaf (d)**) (0))

variables e, g, h, i, j, k, l, d

Figure 5.18: Application of TR_leaf1 in the Example of Figure 5.1, First Refinement Level

5.4.3.2 TR_leaf2: mapping a non-replicator leaf to a typing invariant

A leaf is transformed to an invariant which defines the type of the corresponding variable generated for the leaf in TR_leaf1.

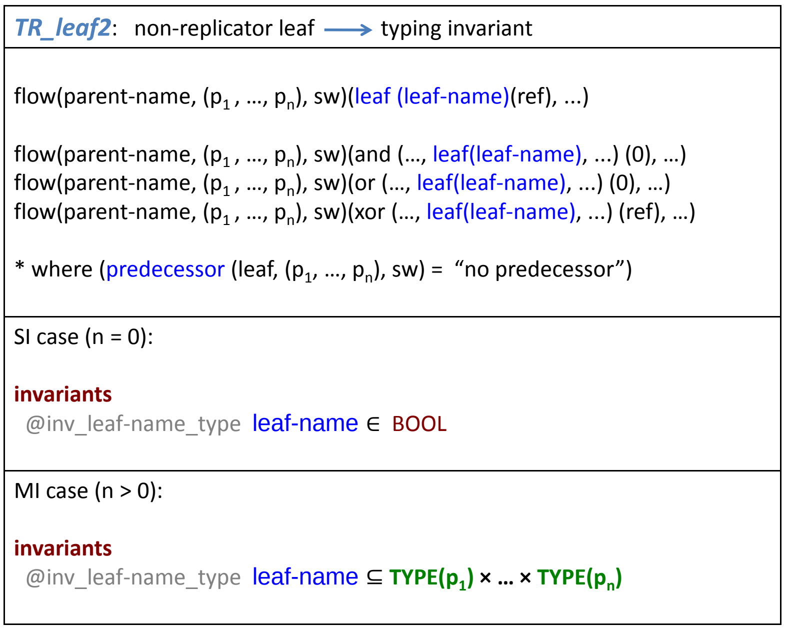
TR_leaf2, TR_leaf3 and TR_leaf4 are about this transformation. TR_leaf2 generates a typing invariant for a non-replicator leaf, (Sequence Pattern 4.2.2, and-constructor Pattern 4.2.4, or-constructor Pattern 4.2.5, xor-constructor Pattern 4.2.6), which has not got a predecessor node. In this case *predecessor* function (Section 5.4.2.3) outputs *no predecessor* for the leaf.

Type of a replicator leaf, (all-replicator Pattern 4.2.7, some-replicator Pattern 4.2.8), is defined in a typing invariant generated in TR_leaf3 (Section 5.4.3.3).

Finally, if a leaf has got a predecessor as the output of *predecessor* function, then its type is defined in a sequencing invariant which is generated in TR_leaf4 (Section 5.4.3.4).

TR_leaf2 is presented in Figure 5.19. If a leaf has not got any parameter ($n = 0$), then its type is *boolean*. Otherwise ($n > 0$), its type is the cartesian product of the type of its parameters.

Figure 5.20 presents the application of this rule in the first refinement level of the example in Figure 5.1. Leaf e is first node and there is no predecessor for it, $predecessor(leaf(e), p_1) = nopredecessor$).



TR_leaf2: non-replicator leaf $\longrightarrow$ typing invariant

flow(parent-name, (p_1 , …, p_n), sw)(leaf (leaf-name)(ref), …)

flow(parent-name, (p_1 , …, p_n), sw)(and (…, leaf(leaf-name), …) (0), …)
flow(parent-name, (p_1 , …, p_n), sw)(or (…, leaf(leaf-name), …) (0), …)
flow(parent-name, (p_1 , …, p_n), sw)(xor (…, leaf(leaf-name), …) (ref), …)

* where (predecessor (leaf, (p_1, …, p_n), sw) = "no predecessor")

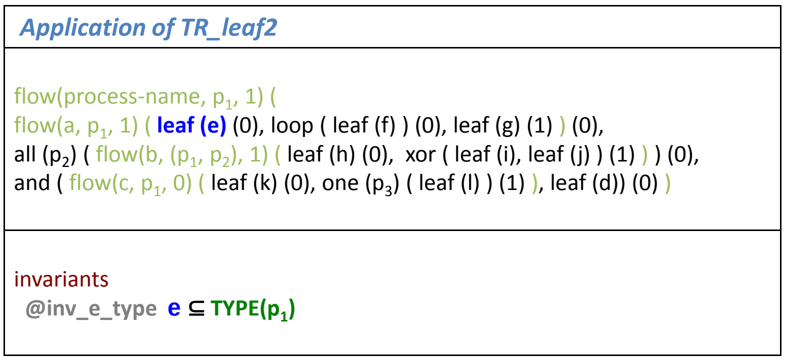
SI case (n = 0):

invariants
@inv_leaf-name_type leaf-name ∈ BOOL

MI case (n > 0):

invariants
@inv_leaf-name_type leaf-name ⊆ **TYPE(p_1) × … × TYPE(p_n)**

Figure 5.19: TR_leaf2: mapping a non-replicator leaf to a typing invariant



Application of TR_leaf2

flow(process-name, p_1, 1) (
flow(a, p_1, 1) (**leaf (e)** (0), loop (leaf (f)) (0), leaf (g) (1)) (0),
all (p_2) (flow(b, (p_1, p_2), 1) (leaf (h) (0), xor (leaf (i), leaf (j)) (1))) (0),
and (flow(c, p_1, 0) (leaf (k) (0), one (p_3) (leaf (l)) (1)), leaf (d)) (0))

invariants
@inv_e_type e ⊆ TYPE(p_1)

Figure 5.20: Application of TR_leaf2 in the Example of Figure 5.1, First Refinement Level

5.4.3.3 TR_leaf3: mapping a replicator leaf to a typing invariant

TR_leaf3, Figure 5.21, outlines the typing invariant translation in the case of a replicator leaf, (all-replicator Pattern 4.2.7, some-replicator Pattern 4.2.8).

Figure 5.22 presents the application of this rule in the first refinement level of the example in Figure 5.1.

TR_leaf3: replicator leaf $\longrightarrow$ typing invariant

flow(parent-name, (p_1 , ..., p_n), sw)(..., all (p_i , leaf(leaf-name)) (0), ...)
flow(parent-name, (p_1 , ..., p_n), sw)(..., some (p_i , leaf(leaf-name))(0), ...)
flow(parent-name, (p_1 , ..., p_n), sw)(..., one (p_i , leaf(leaf-name)) (ref), ...)

invariants
@inv_leaf-name_type leaf-name ⊆ **TYPE(p_1) × ... × TYPE(p_n) × TYPE(p_i)**

Figure 5.21: TR_leaf3: mapping a replicator leaf to a typing invariant

Application of TR_leaf3

flow(process-name, p_1, 1) (
flow(a, p_1, 1) (leaf (e) (0), loop (leaf (f)) (0), leaf (g) (1)) (0),
all (p_2) (flow(b, (p_1, p_2), 1) (leaf (h) (0), xor (leaf (i), leaf (j)) (1))) (0),
and (flow(c, p_1, 0) (leaf (k) (0), one (p_3) (**leaf (l)**) (1)), leaf (d)) (0))

invariants
@inv_l_type l ⊆ TYPE(p_1) × TYPE(p_3)

Figure 5.22: Application of TR_leaf3 in the Example of Figure 5.1, First Refinement Level

5.4.3.4 TR_leaf4: mapping a leaf to a sequencing invariant

As described in Section 4.2, ordering between events is managed with some guards and is specified with some invariants.

TR_leaf4 presented in Figure 5.23, transforms a leaf to a sequencing invariant. Sequencing guard is generated in TR_leaf8 (Section 5.4.3.8). Considering Figure 5.23, first *predecessor* function is applied to the leaf to find the previous child. Then *build_seq_inv* function is applied to the previous child. In *build_seq_inv* function first the leaf/leaves of the final refinement level are found via traversing steps, then *seq_inv* is called inside *build_seq_inv* function for each final refinement level leaf, to generated the appropriate invariant.

Figure 5.24 presents the application of this rule for leaf k in the second refinement level of the example in Figure 5.1. Considering leaf k, the previous child is $all(p2)(flow(b,(p_1,p_2),1)(...))(0)$. *build_seq_inv* function is applied to this child. The output leaves of *build_seq_inv* function are $q(p_1,p_2)$, $r(p_1,p_2)$ and $u(p_1,p_2,p_5)$. For each of them *seq_inv* is called as follows:

$seq\_inv(q,(p_1,p_2),k,(p_1))$

$seq\_inv(r,(p_1,p_2),k,(p_1))$

TR_leaf4: leaf $\longrightarrow$ sequencing invariant

flow(parent-name, (p_1 , …, p_n), sw)(…, leaf (leaf-name)(ref), …)

flow(parent-name, (p_1 , …, p_n), sw)(…, and (…, leaf(leaf-name), …) (0), …)
flow(parent-name, (p_1 , …, p_n), sw)(…, or (…, leaf(leaf-name), …) (0), …)
flow(parent-name, (p_1 , …, p_n), sw)(…, xor (…, leaf(leaf-name), …) (ref), …)

flow(parent-name, (p_1 , …, p_n), sw)(…, all (p_i , leaf(leaf-name)) (0), …)
flow(parent-name, (p_1 , …, p_n), sw)(…, some (p_i , leaf(leaf-name))(0), …)
flow(parent-name, (p_1 , …, p_n), sw)(…, one (p_i , leaf(leaf-name)) (ref), …)

flow(parent-name, (p_1 , …, p_n), sw)(…, loop (leaf(leaf-name)) (0), …)

* where (predecessor (leaf, (p_1, …, p_n), sw) ≠ "no predecessor")

predecessor (leaf, (p_1, …, p_n), sw) = (child, *par)

invariants
@inv_leaf-name_seq build_seq_inv (child, *par, leaf, (p_1, …, p_n))

Figure 5.23: TR_leaf4: mapping a leaf to a sequencing invariant

$seq\_inv(u, (p_1, p_2, p_5), k, (p_1))$

Considering the *seq_inv* function presented in Section 5.4.2.2, p_1 is a common parameter between k and the other three leaves, q, r and u; and p_2 is an all-replicator parameter in q, r and u leaves; therefore the invariant is build in the last case of the *seq_inv* function.

Application of TR_leaf4

flow (process-name, p_1, 1) (
flow (a, p_1, 1) (leaf (e) (0), loop (flow(f, p_1, 1) (and (leaf (m), leaf (n)) (0), leaf (o) (1), all (p4) (leaf (p)) (0))) (0), leaf (g) (1)) (0),

all (p_2) (flow(b, (p_1, p_2), 1) (leaf (h) (0), xor (flow(i, (p_1, p_2), 1) (or (leaf (q), leaf (r)) (0) , leaf (s) (1)), flow(j, (p_1, p_2), 0) (leaf (t), one (p_5) (leaf(u)) (1))) (1))) (0),

and (flow(c, p_1, 1) (**leaf (k)** (0), one (p_3) (flow(l, (p_1, p_3), 1) (some (p_6) (leaf (v)) (0), leaf(w) (1))) (1)), leaf (d)) (0))

invariants
@inv_leaf-name_seq $p_1 \in$ **k** $\Rightarrow$ **q [{p_1}] = TYPE(p_2) ∨**
r [{p_1}] = TYPE(p_2) ∨
dom(u) [{p_1}] = TYPE(p_2)

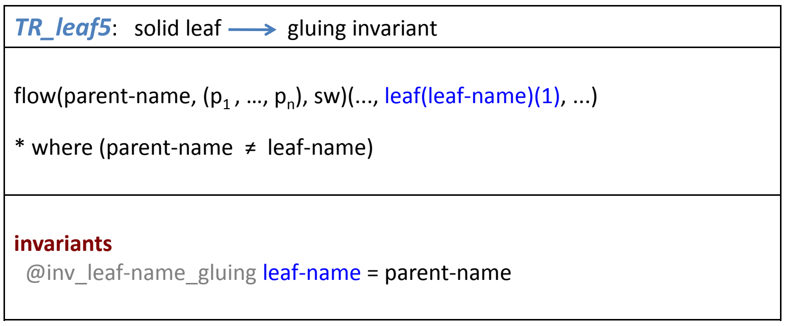
Figure 5.24: Application of TR_leaf4 in the Example of Figure 5.1, Second Refinement Level

5.4.3.5 TR_leaf5: mapping a solid leaf to a gluing invariant

Each leaf with a solid line, (*refining = 1*), is transformed to a gluing invariant.

This leaf can be a simple leaf, TR_leaf5, or a leaf of a refining xor-constructor, TR_xor1, or a refining one-replicator, TR_one1. It is good to recall that other constructors are always non-refining, come with dashed lines (*refining = 0*).

TR_leaf5 outlines this rule for a simple leaf in Figure 5.25, (Sequence Pattern 4.2.2). Since the corresponding event of the leaf refines the parent event, an invariant describes the relation between the concrete variable, *leaf-name* and the abstract variable, *parent-name*. It is important to mention that we need this invariant only when the *leaf-name* and the *parent-name* are different.



TR_leaf5: solid leaf $\longrightarrow$ gluing invariant

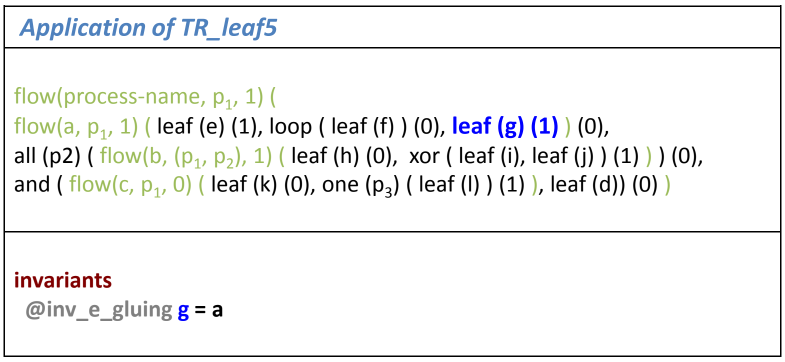
flow(parent-name, (p_1 , …, p_n), sw)(…, leaf(leaf-name)(1), …)

* where (parent-name ≠ leaf-name)

invariants
@inv_leaf-name_gluing leaf-name = parent-name

Figure 5.25: TR_leaf5: mapping a solid leaf to a gluing invariant

Figure 5.26 presents the application of this rule in the first refinement level of the example in Figure 5.1.



Application of TR_leaf5

flow(process-name, p_1, 1) (
flow(a, p_1, 1) (leaf (e) (1), loop (leaf (f)) (0), **leaf (g) (1)**) (0),
all (p2) (flow(b, (p_1, p_2), 1) (leaf (h) (0), xor (leaf (i), leaf (j)) (1))) (0),
and (flow(c, p_1, 0) (leaf (k) (0), one (p_3) (leaf (l)) (1)), leaf (d)) (0))

invariants
@inv_e_gluing g = a

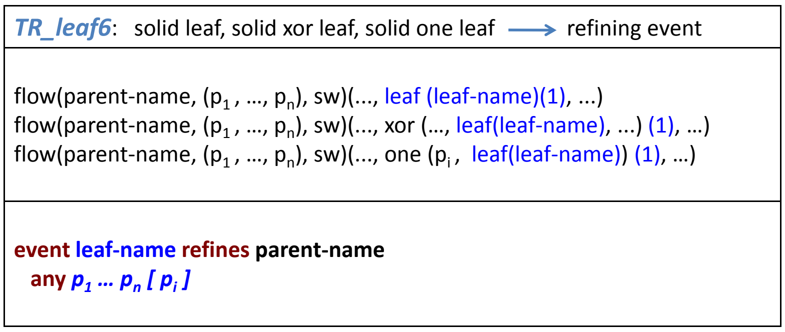
Figure 5.26: Application of TR_leaf5 in the Example of Figure 5.1, First Refinement Level

5.4.3.6 TR_leaf6: mapping a solid leaf to a refining event

A leaf which is connected to its parent with a solid line is transformed to an event which refines the parent event, (Sequence Pattern 4.2.2).

In TR_leaf6, Figure 5.27, each leaf with a solid line, (*refining* = *1*), is transformed to an event which refines the parent event, *parent-name*. As described in Section 4.2, between the logical constructors and replicators, just the xor-constructor and the one-replicator can refine the parent event, (*refining* = *1*). The generated event's name is the same as leaf's name, *leaf-name*.

The list of parameters of a leaf appears in the parameters of the generated event. These parameters include the parent flow parameters followed by any possible replicator's parameter, the one-replicator in this case.



TR_leaf6: solid leaf, solid xor leaf, solid one leaf ⟶ refining event

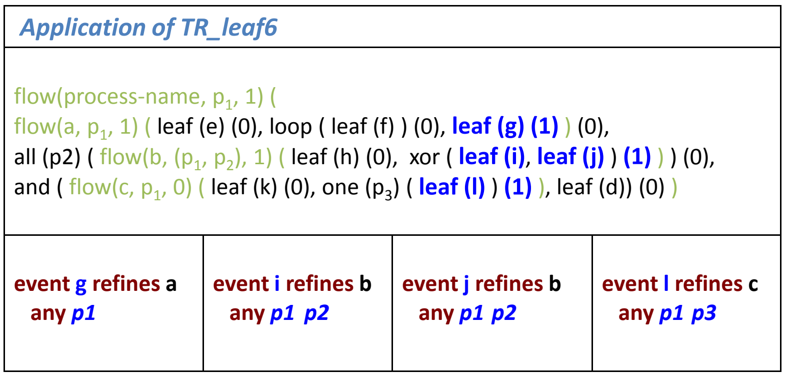
flow(parent-name, (p_1, ..., p_n), sw)(..., leaf (leaf-name)(1), ...)
flow(parent-name, (p_1, ..., p_n), sw)(..., xor (..., leaf(leaf-name), ...) (1), ...)
flow(parent-name, (p_1, ..., p_n), sw)(..., one (p_i, leaf(leaf-name)) (1), ...)

event leaf-name refines parent-name
any *p_1 ... p_n [p_i]*

Figure 5.27: TR_leaf6: mapping a solid leaf to a refining event

Figure 5.28 presents multiple applications of this rule in the first refinement level of the example in Figure 5.1.

- The parameter for *leaf l*, includes its parent, *c*, parameter: *p1*, followed by the one-replicator parameter: *p3*.
- leaf *i* and leaf *j* inherit their refining value from their parent constructor, xor-constructor.



Application of TR_leaf6

flow(process-name, p_1, 1) (
flow(a, p_1, 1) (leaf (e) (0), loop (leaf (f)) (0), **leaf (g) (1)**) (0),
all (p2) (flow(b, (p_1, p_2), 1) (leaf (h) (0), xor (**leaf (i)**, **leaf (j)**) **(1)**)) (0),
and (flow(c, p_1, 0) (leaf (k) (0), one (p_3) (**leaf (l)**) **(1)**), leaf (d)) (0))

event g refines a **any *p1***	**event i refines b** **any *p1 p2***	**event j refines b** **any *p1 p2***	**event l refines c** **any *p1 p3***

Figure 5.28: Application of TR_leaf6 in the Example of Figure 5.1, First Refinement Level

5.4.3.7 TR_leaf7: mapping a dashed leaf to a non-refining event

A leaf which is connected to its parent with a dashed line is transformed to a non-refining event, (Sequence Pattern 4.2.2).

TR_leaf7 is almost same as TR_leaf6. It transforms a leaf with (*refining = 0*) to an event. The difference is that the generated event does not refine the parent event. As described in Section 4.2, all of the constructors are allowed to use dashed line, therefore all of them appear in TR_leaf7. The rule is presented in Figure 5.29.

TR_leaf7: dashed leaf ⟶ non-refining event

flow(parent-name, (p_1 , …, p_n), sw)(…, leaf (leaf-name)(0), …)

flow(parent-name, (p_1 , …, p_n), sw)(…, and (…, leaf(leaf-name) , …) (0), …)
flow(parent-name, (p_1 , …, p_n), sw)(…, or (…, leaf(leaf-name), …) (0), …)
flow(parent-name, (p_1 , …, p_n), sw)(…, xor (…, leaf(leaf-name), …) (0), …)

flow(parent-name, (p_1 , …, p_n), sw)(…, all (p_i , leaf(leaf-name)) (0), …)
flow(parent-name, (p_1 , …, p_n), sw)(…, some (p_i , leaf(leaf-name)) (0), …)
flow(parent-name, (p_1 , …, p_n), sw)(…, one (p_i , leaf(leaf-name)) (0), …)

flow(parent-name, (p_1 , …, p_n), sw)(…, loop (leaf(leaf-name)) (0), …)

event leaf-name
  any *p_1 … p_n [p_i]*

Figure 5.29: TR_leaf7: mapping a dashed leaf to a new event

Figure 5.30 presents multiple applications of this rule in the first refinement level of the example in Figure 5.1.

Application of TR_leaf7

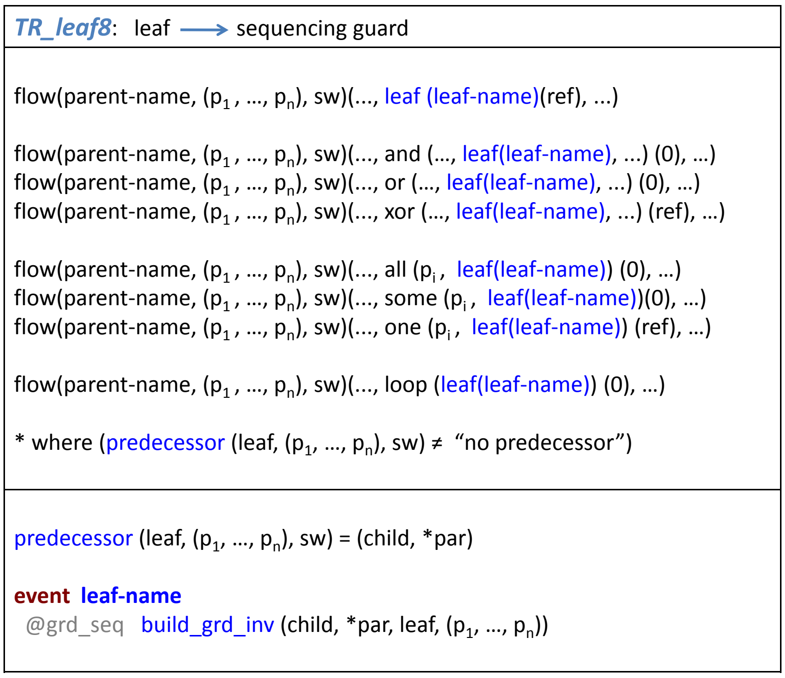
flow(process-name, p_1, 1) (
flow(a, p_1, 1) (**leaf (e) (0)**, loop (**leaf (f)**) **(0)**, loop (leaf (g)) (0)) (0),
all (p2) (flow(b, (p_1, p_2), 1) (**leaf (h) (0)**, xor (leaf (i), leaf (j)) (1))) (0),
and (flow(c, p_1, 0) (**leaf (k) (0)**, one (p_3) (leaf (l)) (1)), **leaf (d)**) **(0)**)

event e **any *p_1***	**event f** **any *p_1***	**event h** **any *p_1, p_2***	**event k** **any *p_1***	**event d** **any *p_1***

Figure 5.30: Application of TR_leaf7 in the Example of Figure 5.1, First Refinement Level

5.4.3.8 TR_leaf8: mapping a leaf to a sequencing guard

TR_leaf8 presented in Figure 5.31, transforms a leaf to a sequencing guard in the corresponding event. In a same way as TR_leaf4, *build_seq_grd* function first outputs the leaf/leaves of the final refinement level of the predecessor child,. Then *seq_grd* is called inside *build_seq_grd* function for each final refinement leaf to generated the appropriate guard.



TR_leaf8: leaf $\longrightarrow$ sequencing guard

flow(parent-name, (p_1, ..., p_n), sw)(..., leaf (leaf-name)(ref), ...)

flow(parent-name, (p_1, ..., p_n), sw)(..., and (..., leaf(leaf-name), ...) (0), ...)
flow(parent-name, (p_1, ..., p_n), sw)(..., or (..., leaf(leaf-name), ...) (0), ...)
flow(parent-name, (p_1, ..., p_n), sw)(..., xor (..., leaf(leaf-name), ...) (ref), ...)

flow(parent-name, (p_1, ..., p_n), sw)(..., all (p_i, leaf(leaf-name)) (0), ...)
flow(parent-name, (p_1, ..., p_n), sw)(..., some (p_i, leaf(leaf-name))(0), ...)
flow(parent-name, (p_1, ..., p_n), sw)(..., one (p_i, leaf(leaf-name)) (ref), ...)

flow(parent-name, (p_1, ..., p_n), sw)(..., loop (leaf(leaf-name)) (0), ...)

* where (predecessor (leaf, (p_1, ..., p_n), sw) ≠ "no predecessor")

predecessor (leaf, (p_1, ..., p_n), sw) = (child, *par)

event leaf-name
@grd_seq build_grd_inv (child, *par, leaf, (p_1, ..., p_n))

Figure 5.31: TR_leaf8: mapping a leaf to a sequencing guard

Figure 5.32 presents the application of this rule for leaf k in the second refinement level of the example in Figure 5.1.

5.4.3.9 TR_leaf9: mapping a non-replicator leaf to a guard

Each leaf is transformed to a guard in the corresponding event of the leaf, generated in TR_leaf2 or TR_leaf3. This guard ensures that the event has not executed before (for the same instance of the event parameter(s)).

TR_leaf9 and TR_leaf10 are about this translation. TR_leaf9 outlines this translation in the case of a non-replicator leaf (Sequence Pattern 4.2.2, and-constructor Pattern 4.2.4, or-constructor Pattern 4.2.5, xor-constructor Pattern 4.2.6). Considering TR_leaf9, Figure 5.33, if leaf has not got any parameter, $(n = 0)$, then the guard is like "*leaf-name* $=$ *FALSE*"; Otherwise $(n > 0)$ the guard ensures that the event has not executed before for the same instance of the parameter(s).

Application of TR_leaf8

flow (process-name, p_1, 1) (
flow (a, p_1, 1) (leaf (e) (0), loop (flow(f, p_1, 1) (and (leaf (m), leaf (n)) (0), leaf (o) (1), all (p4) (leaf (p)) (0))) (0), leaf (g) (1)) (0),

all (p_2) (flow(b, (p_1, p_2), 1) (leaf (h) (0), xor (flow(i, (p_1, p_2), 1) (or (leaf (q), leaf (r)) (0) , leaf (s) (1)), flow(j, (p_1, p_2), 0) (leaf (t), one (p_5) (leaf(u)) (1))) (1))) (0),

and (flow(c, p_1, 1) (**leaf (k)** (0), one (p_3) (flow(l, (p_1, p_3), 1) (some (p_6) (leaf (v)) (0), leaf(w) (1))) (1)), leaf (d)) (0))

event k
any *p1* where
@grd_k_seq **q [{p_1}] = TYPE(p_2) ∨**
r [{p_1}] = TYPE(p_2) ∨
dom(u) [{p_1}] = TYPE(p_2)

Figure 5.32: Application of TR_leaf8 in the Example of Figure 5.1, Second Refinement Level

TR_leaf9: non-replicator leaf $\longrightarrow$ guard

flow(parent-name, (p_1 , ..., p_n), sw)(..., leaf(leaf-name)(ref), ...)

flow(parent-name, (p_1 , ..., p_n), sw)(..., and (..., leaf(leaf-name), ...) (0), ...)
flow(parent-name, (p_1 , ..., p_n), sw)(..., or (..., leaf(leaf-name), ...) (0), ...)
flow(parent-name, (p_1 , ..., p_n), sw)(..., xor (..., leaf(leaf-name), ...) (ref), ...)

SI case (n = 0):

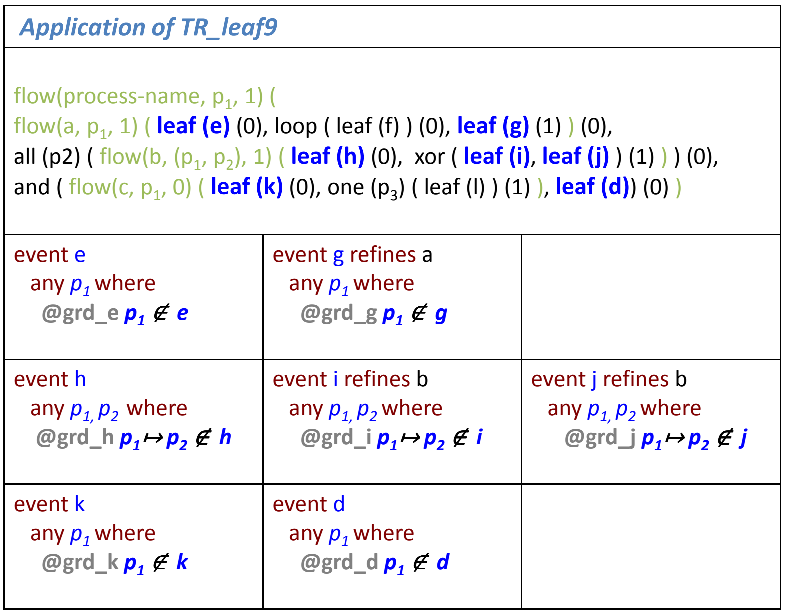
event leaf-name
@grd leaf-name = FALSE

MI case (n > 0):

event leaf-name
@grd $p_1 \mapsto ... \mapsto p_n \notin$ *leaf-name*

Figure 5.33: TR_leaf9: mapping a non-replicator leaf to a guard

Figure 5.34 presents multiple applications of this rule in the first refinement level of the example in Figure 5.1.



Application of TR_leaf9

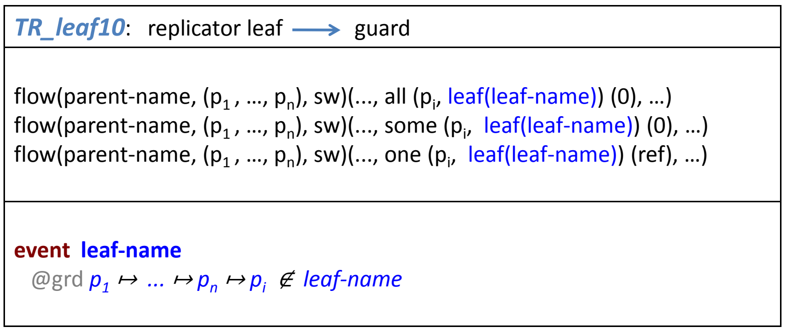
flow(process-name, p_1, 1) (
flow(a, p_1, 1) (**leaf (e)** (0), loop (leaf (f)) (0), **leaf (g)** (1)) (0),
all (p2) (flow(b, (p_1, p_2), 1) (**leaf (h)** (0), xor (**leaf (i)**, **leaf (j)**) (1))) (0),
and (flow(c, p_1, 0) (**leaf (k)** (0), one (p_3) (leaf (l)) (1)), **leaf (d)**) (0))

event e any p_1 where @grd_e $p_1 \notin e$	event g refines a any p_1 where @grd_g $p_1 \notin g$	
event h any p_1, p_2 where @grd_h $p_1 \mapsto p_2 \notin h$	event i refines b any p_1, p_2 where @grd_i $p_1 \mapsto p_2 \notin i$	event j refines b any p_1, p_2 where @grd_j $p_1 \mapsto p_2 \notin j$
event k any p_1 where @grd_k $p_1 \notin k$	event d any p_1 where @grd_d $p_1 \notin d$	

Figure 5.34: Application of TR_leaf9 in the Example of Figure 5.1, First Refinement Level

5.4.3.10 TR_leaf10: mapping a replicator leaf to a guard

TR_leaf10 outlines the guard translation in the case of a replicator leaf (all-replicator Pattern 4.2.7, some-replicator Pattern 4.2.8, one-replicator Pattern 4.2.9). Therefore as least one parameter, the replicator parameter, exists. TR_leaf10 is presented in Figure 5.35.



TR_leaf10: replicator leaf $\longrightarrow$ guard

flow(parent-name, (p_1 , …, p_n), sw)(…, all (p_i, leaf(leaf-name)) (0), …)
flow(parent-name, (p_1 , …, p_n), sw)(…, some (p_i, leaf(leaf-name)) (0), …)
flow(parent-name, (p_1 , …, p_n), sw)(…, one (p_i, leaf(leaf-name)) (ref), …)

event leaf-name
@grd $p_1 \mapsto \dots \mapsto p_n \mapsto p_i \notin$ *leaf-name*

Figure 5.35: TR_leaf10: mapping a replicator leaf to a guard

Figure 5.36 presents the application of this rule in the first refinement level of the example in Figure 5.1.

```
Application of TR_leaf10

flow(process-name, p1, 1) (
flow(a, p1, 1) ( leaf (e) (1), loop ( leaf (f) ) (0), loop ( leaf (g) ) (0) ) (0),
all (p2) ( flow(b, (p1, p2), 1) ( leaf (h) (0),  xor ( leaf (i), leaf (j) ) (1) ) ) (0),
and ( flow(c, p1, 0) ( leaf (k) (0), one (p3) ( leaf (l) ) (1) ), leaf (d)) (0) )

event l
  any p1, p3 where
   @grd_l p1↦p3 ∉ l
```

Figure 5.36: Application of TR_leaf10 in the Example of Figure 5.1, First Refinement Level

5.4.3.11 TR_leaf11: mapping a non-replicator leaf to an action

Each leaf is transformed to an action in the corresponding event of the leaf, generated in TR_leaf2 and TR_leaf3. This action indicates that the event executes (for an instance of the event parameter(s)).

TR_leaf11 and TR_leaf12 are about this translation. TR_leaf11 outlines this translation in the case of in the case of a non-replicator leaf (Sequence Pattern 4.2.2, and-constructor Pattern 4.2.4, or-constructor Pattern 4.2.5, xor-constructor Pattern 4.2.6). In TR_leaf6, Figure 5.37, if leaf has not got any parameter ($n = 0$) then the action is like "*leaf-name := TRUE*"; Otherwise ($n > 0$), the action indicates that the event executes for an instances of the parameter(s).

```
TR_leaf11:  non-replicator leaf ⟶ action

flow(parent-name, (p1 , …, pn), sw)(…, leaf (leaf-name)(0), …)

flow(parent-name, (p1 , …, pn), sw)(…, and (…, leaf(leaf-name) , …) (0), …)
flow(parent-name, (p1 , …, pn), sw)(…, or (…, leaf(leaf-name), …) (0), …)
flow(parent-name, (p1 , …, pn), sw)(…, xor (…, leaf(leaf-name) (ref), …), …)

SI case (n = 0):

event  leaf-name
  @act leaf-name ≔ TRUE

MI case (n > 0):

event  leaf-name
  @act leaf-name ≔ leaf-name ∪ {p1 ↦ … ↦ pn }
```

Figure 5.37: TR_leaf11: mapping a non-replicator leaf to an action

Figure 5.38 presents multiple applications of this rule for in the first refinement level of the example in Figure 5.1.

Application of TR_leaf11

flow(process-name, p_1, 1) (
flow(a, p_1, 1) (**leaf (e)** (0), loop (leaf (f)) (0), **leaf (g)** (1)) (0),
all (p2) (flow(b, (p_1, p_2), 1) (**leaf (h)** (0), xor (**leaf (i)**, **leaf (j)**) (1))) (0),
and (flow(c, p_1, 0) (**leaf (k)** (0), one (p_3) (leaf (l)) (1)), **leaf (d)**) (0))

event e any p_1 where @grd_e $p_1 \notin e$ then @act_e **e ≔ e ∪ {p_1}**	event g refines a any p_1 where @grd_g $p_1 \notin g$ then @act_g **g ≔ g ∪ {p_1}**	
event h any p_1, p_2 where @grd_h $p_1 \mapsto p_2 \notin h$ then @act_h **h ≔ h ∪ {$p_1 \mapsto p_2$}**	event i refines b any p_1, p_2 where @grd_i $p_1 \mapsto p_2 \notin I$ then @act_i **i ≔ i ∪ {$p_1 \mapsto p_2$}**	event j refines b any p_1, p_2 where @grd_j $p_1 \mapsto p_2 \notin j$ then @act_j **j ≔ j ∪ {$p_1 \mapsto p_2$}**
event k any p_1 where @grd_k $p_1 \notin k$ then @act_k **k ≔ k ∪ {p_1}**	event d any p_1 where @grd_d $p_1 \notin d$ then @act_d **d ≔ d ∪ {p_1}**	

Figure 5.38: Application of TR_leaf11 in the Example of Figure 5.1, First Refinement Level

5.4.3.12 TR_leaf12: mapping a replicator leaf to an action

TR_leaf12 outlines the action translation in the case of a replicator leaf (all-replicator Pattern 4.2.7, some-replicator Pattern 4.2.8, one-replicator Pattern 4.2.9). Therefore as least one parameter, the replicator parameter, exists. TR_leaf12 is presented in Figure 5.39.

TR_leaf12: replicator leaf ⟶ action

flow(parent-name, (p_1 , …, p_n), sw)(…, all (p_i, leaf(leaf-name)) (0), …)
flow(parent-name, (p_1 , …, p_n), sw)(…, some (p_i, leaf(leaf-name)) (0), …)
flow(parent-name, (p_1 , …, p_n), sw)(…, one (p_i, leaf(leaf-name)) (ref), …)

event leaf-name
@act leaf-name ≔ leaf-name ∪ {$p_1 \mapsto \ldots \mapsto p_n \mapsto p_i$}

Figure 5.39: TR_leaf12: mapping a replicator leaf to an action

Figure 5.40 presents the application of this rule in the first refinement level of the example in Figure 5.1.

Application of TR_leaf12

flow(process-name, p_1, 1) (
flow(a, p_1, 1) (leaf (e) (0), loop (leaf (f)) (0), leaf (g) (1)) (0),
all (p2) (flow(b, (p_1, p_2), 1) (leaf (h) (0), xor (leaf (i), leaf (j)) (1))) (0),
and (flow(c, p_1, 0) (leaf (k) (0), one (p_3) (**leaf (l)**) (1)), leaf (d)) (0))

event l
  any p_1, p_3 where
    @grd_l $p_1 \mapsto p_3 \notin l$
  then
    @act_l $l \coloneqq l \cup \{p_1 \mapsto p_3\}$

Figure 5.40: Application of TR_leaf12 in the Example of Figure 5.1, First Refinement Level

5.4.4 Translating the xor-constructor

5.4.4.1 TR_xor1: mapping a solid xor-constructor to a gluing invariant

TR_xor1 describes the gluing invariant translation in the case of a solid xor-constructor, (xor-constructor Pattern 4.2.6). In this case all leaves of the solid xor-constructor refine the parent event, as generated in TR_leaf6. The gluing invariant describes the relation between concrete variables of xor-constructor leaves and the abstract variable. Also it ensures that just one of the xor-constructor leaves is allowed to execute. Figure 5.41 presents TR_xor1.

TR_xor1: solid xor-constructor $\longrightarrow$ gluing invariant

flow(parent-name, (p_1 , …, p_n), sw)(…, xor (leaf($name_1$), …, leaf($name_n$)) (1), …)

SI case (n = 0):

invariants
  @inv_xor_gluing
    partition({parent-name} ∩ {TRUE},
        {$name_1$} ∩ {TRUE}, …, {$name_n$} ∩ {TRUE})

MI case (n > 0):

invariants
  @inv_xor_gluing partition(parent-name, $name_1$, …, $name_n$)

Figure 5.41: TR_xor1: mapping a solid xor-constructor to a gluing invariant

Figure 5.42 presents the application of this rule in the first refinement level of the example in Figure 5.1.

Application of TR_xor1

flow(process-name, p_1, 1) (
flow(a, p_1, 1) (leaf (e) (0), loop (leaf (f)) (0), leaf (g) (1)) (0),
all (p2) (flow(b, (p_1, p_2), 1) (leaf (h) (0), **xor (leaf (i), leaf (j)) (1)**)) (0),
and (flow(c, p_1, 0) (leaf (k) (0), one (p_3) (leaf (l)) (1)), leaf (d)) (0))

invariants
@inv_xor_gluing **partition(b, i, j)**

Figure 5.42: Application of TR_xor1 in the Example of Figure 5.1, First Refinement Level

5.4.4.2 TR_xor2: mapping a xor-constructor to an invariant

For each xor-constructor we need to ensure that just one of its children is allowed to execute. This constraint is modelled in Event-B with an invariant and a guard in each generated event of each xor-constructor children, in TR_xor2 and TR_xor3 respectively, (xor-constructor Pattern 4.2.6).

TR_xor2 transforms the xor-constructor to an invariant, Figure 5.43. In the generated invariant, we need to specify that the variables corresponding to leaf/leaves of each xor-constructor child, are mutually exclusive. If the parent flow of the xor-constructor has no parameter ($n = 0$), we use *disjunction_of_leaves* function to get the proper expression for each xor-constructor child. Then the invariant specifies a mutual exclusive relation between the outputs of *disjunction_of_leaves* function. In the Event-B language, the xor operator is not implemented. In the case that there are some parent flow parameter(s) ($n > 0$), we can use partition operator to describe the mutual exclusive relationship.

TR_xor2: xor-constructor $\longrightarrow$ invariant

flow(parent-name, (p_1, …, p_n), sw)(…, xor ($child_1$, …, $child_m$) (ref), …)

SI case (n = 0):

invariants
@inv_xor disjunction_of_leaves ($child_1$, 0) **xor** … **xor** disjunction_of_leaves ($child_m$, 0)

MI case (n > 0):

invariants
@inv_xor partition((union_of_leaves ($child_1$, 0) ∪ …∪ union_of_leaves ($child_m$, 0)),
union_of_leaves ($child_1$, 0), …, union_of_leaves ($child_m$, 0))

Figure 5.43: TR_xor2: mapping a xor-constructor to an invariant

Figure 5.44 presents the application of this rule in the second refinement level of the example in Figure 5.1.

Application of TR_xor2

flow (process-name, p_1, 1) (
flow (a, p_1, 1) (leaf (e) (0), loop (flow(f, p_1, 1) (and (leaf (m), leaf (n)) (0), leaf (o) (1), all (p_4) (leaf (p)) (0))) (0), leaf (g) (1)) (0),

all (p2) (flow(b, (p_1, p_2), 1) (leaf (h) (0), **xor (flow(i, p_1, p_2, 1) (or (leaf (q), leaf (r)) (0) , leaf (s) (1)), flow(j, p_1, p_2, 0) (leaf (t), one (p_5) (leaf(u)) (1))) (1)**)) (0),

and (flow(c, p_1, 1) (leaf (k) (0), one (p_5) (flow(l, (p_1, p_3), 1) (some (p_6) (leaf (v)) (0), leaf(w) (1))) (1)), leaf (d)) (0))

MI case (n > 0):

invariants
@inv_xor **partition((q ∪ r ∪ dom(u)) , q ∪ r, dom(u))**

SI case (n = 0):

invariants
@inv_xor **(q ∨ r) xor (u ≠ ∅)**

Figure 5.44: Application of TR_xor2 in the Example of Figure 5.1, Second Refinement Level

In the last row of the figure we assume that the xor-constructor is included in a parent flow without parameter.

TR_xor1 (Section 5.4.4.1) was about a solid xor-constructor when all of its children are leaves. Whereas TR_xor2 is transformed a dashed xor-constructor, or a (solid or dashed) xor-constructor witch as least one of its children is a flow, not a leaf.

5.4.4.3 TR_xor3: mapping a xor-constructor to guards

TR_xor3 in Figure 5.45, presents generation of guards for the xor-constructor. At least two guards are generated for each xor-constructor since there are at least two children for each xor-constructor (xor-constructor Pattern 4.2.6). First *list_of_leaves* function is applied to each xor-constructor child. The result would be a list of leaves. Then for the corresponding event of each leaf in the list (same name as leaf name), one guard is added. In the guard we aim to check that other xor-constructor children have not executed before.

Since for each xor-constructor child, we need to check that none of other xor-constructor children has executed before, *conjunction_of_leaves* function, in case of ($n = 0$), and *union_of_leaves* function, in case of ($n > 0$), are called for the other child of the xor-constructor.

TR_xor3: xor-constructor $\longrightarrow$ 2*guard

flow(parent-name, (p_1 , …, p_n), sw)(…, xor ($child_1$, …, $child_m$) (ref), …)

list_of_leaves ($child_i$, (p_1 , …, p_n)) =
leaf $^i{}_1$ (leaf-name$^i{}_1$, *par$^i{}_1$), …, leaf $^i{}_k$ (leaf-name$^i{}_k$, *par$^i{}_k$)   (1 <= i <= m)

SI case (n = 0):
event leaf-name$^i{}_j$ (1 <= j <= k)
@grd_xor conjunction_of_leaves ($child_1$, 0) ∧
… ∧
conjunction_of_leaves ($child_{i-1}$, 0) ∧
conjunction_of_leaves ($child_{i+1}$, 0) ∧
… ∧
conjunction_of_leaves ($child_m$, 0)

MI case (n > 0):
event leaf-name$^i{}_j$ (1 <= j <= k)
any p_1 … p_n … where
@grd_xor **p_1↦…↦p_n** ∉ union_of_leaves ($child_1$, 0) ∪
… ∪
union_of_leaves ($child_{i-1}$, 0) ∪
union_of_leaves ($child_{i+1}$, 0) ∪
… ∪
union_of_leaves ($child_m$, 0)

Figure 5.45: TR_xor3: mapping a xor-constructor to guards

Figure 5.46 presents multiple applications of this rule in the second refinement level of the example in Figure 5.1. In the last row of the figure we assume that xor-constructor is included in a parent flow without parameter.

Application of TR_xor3

flow (process-name, p_1, 1) (
flow (a, p_1, 1) (leaf (e) (0), loop (flow(f, p_1, 1) (and (leaf (m), leaf (n)) (0), leaf (o) (1), all (p_4) (leaf (p)) (0))) (0), leaf (g) (1)) (0),

all (p2) (flow(b, (p_1, p_2), 1) (leaf (h) (0), **xor (flow(i, p_1, p_2, 1) (or (leaf (q), leaf (r)) (0) , leaf (s) (1)), flow(j, p_1, p_2, 0) (leaf (t), one (p_5) (leaf(u)) (1))) (1)**)) (0),

and (flow(c, p_1, 1) (leaf (k) (0), one (p_5) (flow(l, (p_1, p_3), 1) (some (p_6) (leaf (v)) (0), leaf(w) (1))) (1)), leaf (d)) (0))

MI case (n > 0):

event q any p_1, p_2 where @grd_q_xor **p_1↦p_2 ∉ dom(u)**	event r any p_1, p_2 where @grd_r_xor **p_1↦p_2 ∉ dom(u)**	event u refines j any p_1, p_2 p_5 where @grd_u_xor **p_1↦p_2 ∉ q ∪ r**

SI case (n = 0):

event q where @grd_q_xor **u = ∅**	event r where @grd_r_xor **u = ∅**	event u refines j any p_5 where @grd_u_xor **q = FALSE ∧ r = FALSE**

Figure 5.46: Application of TR_xor3 in the Example of Figure 5.1, Second Refinement Level

5.4.5 Translating the one-replicator

5.4.5.1 TR_one1: mapping a solid one-replicator to a gluing invariant

TR_one1 describes the gluing invariant translation in the case of a solid one-replicator, (one-replicator Pattern 4.2.9). In this case the solid one-replicator leaf event refines the parent event, as generated in TR_leaf6. The gluing invariant describes the relation between concrete variable of the one-replicator leaf and the abstract variable. Figure 5.47 presents TR_one1.

TR_one1: solid one-replicator $\longrightarrow$ gluing invariant

flow(parent-name, (p_1 , …, p_n), sw)(…, one (p_i, leaf(leaf-name)) (1), …)

* where (parent-name ≠ leaf-name)

SI case (n = 0):

invariants
@inv_one_gluing leaf-name ≠ ∅ ⇔ parent-name = TRUE

MI case (n > 0):

invariants
@inv_one_gluing dom(leaf-name) = parent-name

Figure 5.47: TR_one1: mapping a solid one-replicator to a gluing invariant

Figure 5.48 presents the application of this rule in the first refinement level of the example in Figure 5.1.

Application of TR_one1

flow(process-name, p_1, 1) (
flow(a, p_1, 1) (leaf (e) (0), loop (leaf (f)) (0), leaf (g) (1)) (0),
all (p2) (flow(b, (p_1, p_2), 1) (leaf (h) (0), xor (leaf (i), leaf (j)) (1))) (0),
and (flow(c, p_1, 0) (leaf (k) (0), **one (p_3) (leaf (l)) (1)**), leaf (d)) (0))

invariants
@inv_one_gluing dom(l) = c

Figure 5.48: Application of TR_one1 in the Example of Figure 5.1, First Refinement Level

5.4.5.2 TR_one2: mapping an one-replicator to (an) invariant(s)

The one-replicator child can execute only for one instance of the one-replicator parameter. This constraint is modelled in Event-B with (an) invariant(s) and (a) guard(s), in

TR_one2 and TR_one3 respectively, (one-replicator Pattern 4.2.9).

In TR_one2, presented in Figure 5.49, one or more invariant(s) is generated for an one-replicator. First *list_of_leaves* function is applied to the one-replicator child to find the list of leaves in the last refinement level of an one-replicator child. Then for each leaf in the list, one invariant is generated depending on the leaf's parameter list.

TR_one2 : one-replicator $\longrightarrow$ 1*invariant

flow(parent-name, (p_1 , ..., p_n), sw)(..., one (par-one, child)(ref), ...)

list_of_leaves (child, 0) = leaf$_1$(leaf-name$_1$, *par$_1$), ..., leaf$_k$(leaf-name$_k$, *par$_k$)

leaf-name$_i$: $\overset{X}{\overleftrightarrow{p_1, \ldots, p_n, \text{par-one}}}$, $p^i_1, \ldots, p^i_m$

Where X = dom$_1$(... dom$_m$ (leaf-name$_i$) ...)

invariants

@inv_one $n \neq 0, m_i \neq 0 : \forall p_1, \ldots, p_n \cdot \text{card}(X [\{p_1 \mapsto \ldots \mapsto p_n\}]) \leq 1$

$n = 0, m_i = 0 : \text{card}(\text{leaf-name}_i) \leq 1$

$n = 0, m_i \neq 0 : \text{card}(X) \leq 1$

$n \neq 0, m_i = 0 : \forall p_1, \ldots, p_n \cdot \text{card}(\text{leaf-name}_i [\{p_1 \mapsto \ldots \mapsto p_n\}]) \leq 1$

Figure 5.49: TR_one2: mapping an one-replicator to (an) invariant(s)

Each leaf inherits its parameter from its parent flow and the possible parent replicator. As presented in Figure 5.49, in TR_one2 each leaf's parameter list is divided to three parts. First is the parameters which are the same as the one-replicator parameters, $p_1 ... p_n$. Second is the one-replicator parameter, *par-one*. Finally the possible parameters which can be added from other replicators below the one-replicator, $p^i_1 ... p^i_m$.

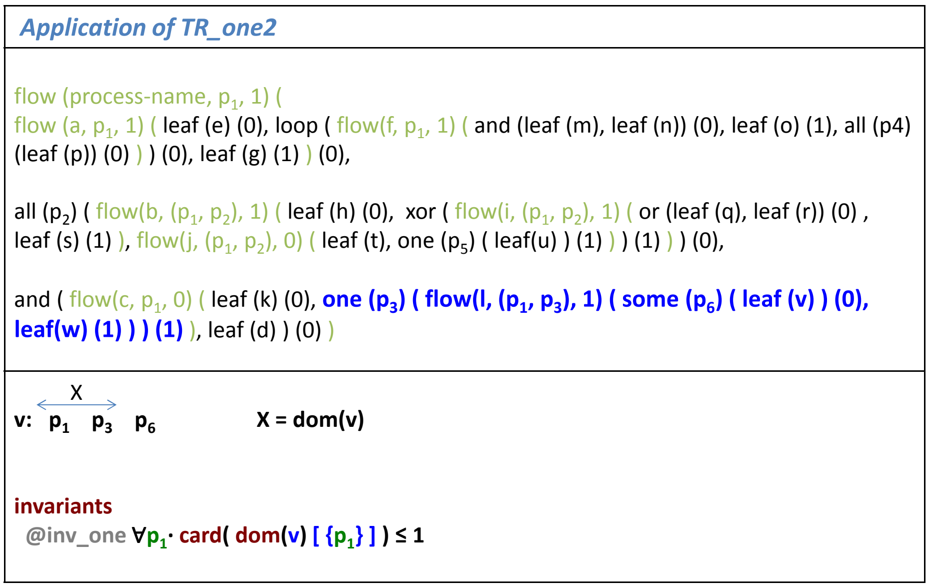
The invariant restricts the value of the one-replicator parameter in the different executions of event *leaf-name*. In all executions of event *leaf-name*, *par-one* can take only one value per each instance of $(p_1 \mapsto ... \mapsto p_n)$.

So in the most general case ($n \neq 0$, $m \neq 0$), the cardinality of image of X on $(p_1 \mapsto ... \mapsto p_n)$ shows the number of *par-one*'s value per $(p_1 \rightarrow ... \rightarrow p_n)$, which should be at most one. It is helpful to represent the definition of relational image operator here:

$$relation[S] = \{y | \exists x.x \in S \wedge x \mapsto y \in relation\}$$

Figure 5.50 presents the application of this rule in the second refinement level of the example in Figure 5.1.

list_of_leaves function returns only one leaf, *v*. Leaf *v* has three parameters, one before the one-replicator, *p1*, the one-replicator parameter, *p3*, and one after the one-replicator, *p6*, added with the some-replicator. The one-parameter *p3* can take only one value for



Application of TR_one2

flow (process-name, p_1, 1) (
flow (a, p_1, 1) (leaf (e) (0), loop (flow(f, p_1, 1) (and (leaf (m), leaf (n)) (0), leaf (o) (1), all (p4) (leaf (p)) (0))) (0), leaf (g) (1)) (0),

all (p_2) (flow(b, (p_1, p_2), 1) (leaf (h) (0), xor (flow(i, (p_1, p_2), 1) (or (leaf (q), leaf (r)) (0) , leaf (s) (1)), flow(j, (p_1, p_2), 0) (leaf (t), one (p_5) (leaf(u)) (1))) (1))) (0),

and (flow(c, p_1, 0) (leaf (k) (0), **one (p_3) (flow(l, (p_1, p_3), 1) (some (p_6) (leaf (v)) (0), leaf(w) (1))) (1)**), leaf (d)) (0))

v: $\overset{X}{\overleftrightarrow{p_1 \quad p_3}} \quad p_6$ **X = dom(v)**

invariants
@inv_one ∀p_1· card(dom(v) [{p_1}]) ≤ 1

Figure 5.50: Application of TR_one2 in the Example of Figure 5.1, Second Refinement Level

all executions of event v per each instance of *p1*. Whereas for each instance of *p6*, event v can execute with more than one value for one-parameter *p3*. To make this point clear assume :

$TYPE(p_1) = \{a\}$
$TYPE(p_3) = \{c, d\}$
$TYPE(p_6) = \{e, f\}$

Then these two executions of event $v(p_1, p_3, p_6)$ is allowed:

< *v(a, c, e), v(a, c, f)* >

Whereas after those two execution, $v(a, d, e)$ or $v(a, d, f)$ violates the invariant $(card(dom(v)[\{a\}]) \leq 1)$. Because one-parameter *p3*, can not take more than one value per any instance of *p1*, value *a* here.

5.4.5.3 TR_one3: mapping an one-replicator to (a) guard(s)

In TR_one3, presented in Figure 5.51, one or more guard(s) is generated for the one-replicator. What we do in TR_one3 is like TR_one2. In the guard of the one-replicator leaf/leaves, we need to ensure that the one-replicator parameter's value per $(p_1 \mapsto ... \mapsto p_n)$ is unique.

Figure 5.52 presents the application of this rule in the second refinement level of the example in Figure 5.1. Considering the assumption in the example of previous translation rule, when $v = \{(a, c, e), (a, c, f)\}$ then the generated guard is false for $v(a, d, e)$ or $v(a, d, f)$, since $d \notin dom(v)[\{a\}]$, where $dom(v)[\{a\}] = \{c\}$.

TR_one3: one-replicator $\longrightarrow$ 1*guard

flow(parent-name, $(p_1, \ldots, p_n)$, sw)(..., one (par-one, child)(ref), ...)

list_of_leaves (child, $(p_1, \ldots, p_n$, par-one) = $leaf_1$($leaf\text{-}name_1$, $*par_1$), ..., $leaf_k$($leaf\text{-}name_k$, $*par_k$)

$leaf\text{-}name_i : \overbrace{p_1, \ldots, p_n, \text{ par-one}}^{X}, p^i_1, \ldots, p^i_m$

Where X = dom_1(... dom_m ($leaf\text{-}name_i$) ...)

$(1 \leq i \leq k)$
event $leaf\text{-}name_i$
any $p_1 \ldots p_n$ *par-one* $p^i_1 \ldots p^i_m$ where

@grd_one $n \neq 0, m_i \neq 0 : X[\{p_1 \mapsto \ldots \mapsto p_n\}] \neq \emptyset \Rightarrow \text{par-one} \in X[\{p_1 \mapsto \ldots \mapsto p_n\}]$
$n = 0, m = 0 : leaf\text{-}name_i = \emptyset$
$n = 0, m_i \neq 0 : leaf\text{-}name_i \neq \emptyset \Rightarrow \text{par-one} \in dom(leaf\text{-}name_i)$
$n \neq 0, m_i = 0 : p_1 \mapsto \ldots \mapsto p_n \notin dom(leaf\text{-}name_i)$

Figure 5.51: TR_one3: mapping an one-replicator to (a) guard(s)

Application of TR_one3

flow (process-name, p_1, 1) (
flow (a, p_1, 1) (leaf (e) (0), loop (flow(f, p_1, 1) (and (leaf (m), leaf (n)) (0), leaf (o) (1), all (p4) (leaf (p)) (0))) (0), leaf (g) (1)) (0),

all (p_2) (flow(b, (p_1, p_2), 1) (leaf (h) (0), xor (flow(i, (p_1, p_2), 1) (or (leaf (q), leaf (r)) (0) , leaf (s) (1)), flow(j, (p_1, p_2), 0) (leaf (t), one (p_5) (leaf(u)) (1))) (1))) (0),

and (flow(c, p_1, 0) (leaf (k) (0), **one (p_3) (flow(l, (p_1, p_3), 1) (some (p_6) (leaf (v)) (0), leaf(w) (1))) (1)**), leaf (d)) (0))

event v
any p_1 p_3 p_6 where
@grd_q_one **dom(v) [{p_1}] $\neq \emptyset \Rightarrow p_3 \in$ dom(v) [{p_1}]**

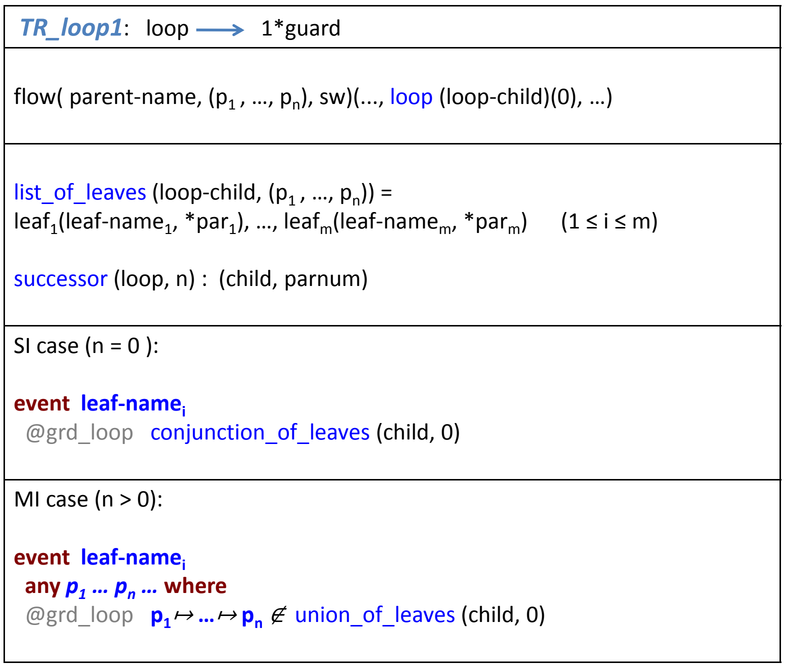
Figure 5.52: Application of TR_one3 in the Example of Figure 5.1, Second Refinement Level

5.4.6 Translating the Loop Constructor

5.4.6.1 TR_loop1: mapping a loop to (a) guard(s)

The loop child can execute zero or more time(s) before execution of next child, (Loop Pattern 4.2.3). This constraint is modelled in Event-B with a guard added to the loop child. The guard ensures that next child has not executed yet.

TR_loop1 presented in Figure 5.53, transforms a loop to one or more guard(s) in the loop child event(s).



TR_loop1: loop $\longrightarrow$ 1*guard

flow(parent-name, (p_1 , …, p_n), sw)(…, loop (loop-child)(0), …)

list_of_leaves (loop-child, (p_1 , …, p_n)) =
$leaf_1$($leaf\text{-}name_1$, *par_1), …, $leaf_m$($leaf\text{-}name_m$, *par_m) $(1 \le i \le m)$

successor (loop, n) : (child, parnum)

SI case (n = 0):

event **$leaf\text{-}name_i$**
@grd_loop conjunction_of_leaves (child, 0)

MI case (n > 0):

event **$leaf\text{-}name_i$**
any ***p_1 … p_n …*** **where**
@grd_loop **$p_1 \mapsto … \mapsto p_n$** $\notin$ union_of_leaves (child, 0)

Figure 5.53: TR_loop1: mapping a loop to (a) guard(s)

First *list_of_leaves* function is applied to the loop child to find the loop leaf/leaves in the final refinement level. For each final refinement leaf, we need to generate a guard. We use *successor* function to find next child of the loop. Finally if the parent flow of the loop has no parameter ($n = 0$), then *conjunction_of_leaves* function is applied to the next child, and a guard is generated in the leaf event. Otherwise ($n > 0$), *union_of_leaves* function is used to generated the guard.

Figure 5.54 presents the application of this rule in the second refinement level of the example in Figure 5.1. The leaves of the first Loop child, m and n, can execute until event g executes. In the last row of the figure we assume that loop is included in a parent flow without parameter.

Application of TR_loop1	
flow (process-name, p_1, 1) (flow (a, p_1, 1) (leaf (e) (0), **loop (flow(f, p_1, 1) (and (leaf (m), leaf (n)) (0), leaf (o) (1), all (p4) (leaf (p)) (0)))** (0), leaf (g) (1)) (0), all (p_2) (flow(b, (p_1, p_2), 1) (leaf (h) (0), xor (flow(i, (p_1, p_2), 1) (or (leaf (q), leaf (r)) (0) , leaf (s) (1)), flow(j, (p_1, p_2), 0) (leaf (t), one (p_5) (leaf(u)) (1))) (1))) (0), and (flow(c, p_1, 1) (leaf (k) (0), one (p_3) (flow(l, (p_1, p_3), 1) (some (p_6) (leaf (v)) (0), leaf(w) (1))) (1)), leaf (d)) (0))	
MI case (n > 0):	
event m any p_1 where **@grd_m_loop** $\boldsymbol{p_1 \notin g}$	event n any p_1 where **@grd_n_loop** $\boldsymbol{p_1 \notin g}$
SI case (n = 0):	
event m where **@grd_m_loop** **g = ∅**	event n where **@grd_n_loop** **g = ∅**

Figure 5.54: Application of TR_loop1 in the Example of Figure 5.1, Second Refinement Level

5.4.6.2 TR_loop2: mapping a loop to (a) guard(s)

The event(s) after a loop can not execute in the middle of execution of the loop events. This ensures with a guard which is added to the next event(s), (Section 4.3.5).

TR_loop2 presented in Figure 5.55, transforms a loop to one or more guard(s) in the next event(s). This translation is applied to the loop only when the loop contains a flow, not a single leaf ($loop\text{-}child \neq leaf$). Because as described in Section 4.3.5, when a loop contain a single leaf we do not need to add an extra guard in the next event(s) after loop.

First *successor* function is used to find next child of the loop. Then *list_of_leaves* function is applied to the next child, to find the leaf/leaves of the final refinement level. Finally in the event of each final refinement leaf of the next child, a guard is generated, in the same way as TR_loop1 (Section 5.4.6.1).

Figure 5.56 presents the application of this rule in the second refinement level of the example in Figure 5.1. Here leaf g is the next child after loop. The generated guard ensures that event g does not execute in the middle of execution of loop events, as described in Section 4.3.5. In the last row of the figure we assume that loop is included in a parent flow without parameter.

TR_loop2: loop $\longrightarrow$ 1*guard

flow(parent-name, (p_1 , …, p_n), sw)(…, loop (loop-child)(0), …)

* where (loop-child ≠ leaf)

successor (loop, n) = (child, parnum)

list_of_leaves (child, (p_1 , …, p_n)) =
leaf_1(leaf-name_1, *par_1), …, leaf_m(leaf-name_m, *par_m) $(1 \le i \le m)$

SI case (n = 0):

event leaf-name$_i$
 @grd_loop conjunction_of_leaves (loop-child, 0)

MI case (n > 0):

event leaf-name$_i$
 any p_1 … p_n … where
 @grd_loop **$p_1 \mapsto … \mapsto p_n$** $\not\in$ union_of_levaes (loop-child, 0)

Figure 5.55: TR_loop2: mapping a loop to a resetting event

Application of TR_loop2

flow (process-name, p_1, 1) (
flow (a, p_1, 1) (leaf (e) (0), **loop (flow(f, p_1, 1) (and (leaf (m), leaf (n)) (0), leaf (o) (1), all (p4) (leaf (p)) (0)))** (0), leaf (g) (1)) (0),

all (p_2) (flow(b, (p_1, p_2), 1) (leaf (h) (0), xor (flow(i, (p_1, p_2), 1) (or (leaf (q), leaf (r)) (0) , leaf (s) (1)), flow(j, (p_1, p_2), 0) (leaf (t), one (p_5) (leaf(u)) (1))) (1))) (0),

and (flow(c, p_1, 1) (leaf (k) (0), one (p_3) (flow(l, (p_1, p_3), 1) (some (p_6) (leaf (v)) (0), leaf(w) (1))) (1)), leaf (d)) (0))

MI case (n > 0):

event g
 any p_1 where
 @grd_g_loop p_1 $\not\in$ m ∪ n

SI case (n = 0):

event g
 where
 @grd_g_loop m = FALSE ∧ n = FALSE

Figure 5.56: Application of TR_loop2 in the Example of Figure 5.1, Second Refinement Level

5.4.6.3 TR_loop3: mapping a loop to a resetting event

As described in 4.3.5, when a loop contains a flow rather than a single leaf ($loop-child \neq leaf$), we need a resetting event in order to reset the loop control variables to enable more than one execution of the loop events.

TR_loop3 presented in Figure 5.57, transforms a loop to a resetting event. First *list_of_leaves* function finds the loop leaves. Then for each output of the *list_of_leaves*, a resetting action is generated in an event. If the parent flow of the loop does not have any parameter, ($n = 0$), then the loop control variables are either boolean ($n_i = 0$) or a set ($n_i \neq 0$), since some parameter can be introduced with some possible replicators. Otherwise ($n > 0$), the loop control variables can have same parameter list as the parent flow of the loop ($n_i = n$), or a longer list of parameters ($n_i > n$) as a result of introducing some new parameters with possible replicators. In the case of ($n_i > n$), we use domain subtraction operators to reset the control variable. The domain subtraction operator is defined as below:

$S \lhd r = \{x, y | x \mapsto y \in r \wedge x \notin S\}$

The generated guard ensures that the last child of the loop has been executed. We use *build_seq_grd* function as a same way in TR_leaf8 (Section 5.4.3.8) to find the final refinement leaf/leaves of the loop and generate the proper guard.

TR_loop3: loop $\longrightarrow$ resetting event
flow(parent-name, (p_1, …, p_n), sw)(…, loop (loop-child)(0), …) loop-child ≠ leaf
list_of_leaves (loop-child, (p_1, …, p_n)) = leaf$_1$(name$_1$, (p_{11}, …, p_{n1})), …, leaf$_m$(name$_m$, (p_{1m}, …, p_{nm}))   ($1 \leq i \leq m$)

SI case (n = 0):
event reset_loop
  where
  @grd_reset build_seq_grd (loop-child, null, null, null)
  then
  @act$_i$_reset [ni = 0: name$_i$ ≔ FALSE
                     ni ≠ 0: name$_i$ ≔ ∅

MI case (n > 0):
event reset_loop
  any *p_1* … *p_n* where
  @grd_reset build_seq_grd (loop-child, (p_1, …, p_n), null, (p_1, …, p_n))
  then
  @act$_i$_reset [ni = n: name$_i$ ≔ name$_i$ \ { $p_1 \mapsto … \mapsto p_n$ }
                     ni > n: name$_i$ ≔ { $p_1 \mapsto … \mapsto p_n$ } ⩤ name$_i$

Figure 5.57: TR_loop3: mapping a loop to a resetting event

Figure 5.58 presents the application of this rule in the second refinement level of the example in Figure 5.1. The last child of the loop is an all-replicator. So the guard ensures that the all-replicator event, event p, has been executed for all of instances of the all-replicator parameter, p_4. Then all of the loop control variables are reset in the actions of the resetting event.

Application of TR_loop3	
flow (process-name, p_1, 1) (flow (a, p_1, 1) (leaf (e) (0), **loop (flow(f, p_1, 1) (and (leaf (m), leaf (n)) (0), leaf (o) (1), all (p4) (leaf (p)) (0)))** (0), leaf (g) (1)) (0), all (p_2) (flow(b, (p_1, p_2), 1) (leaf (h) (0), xor (flow(i, (p_1, p_2), 1) (or (leaf (q), leaf (r)) (0) , leaf (s) (1)), flow(j, (p_1, p_2), 0) (leaf (t), one (p_5) (leaf(u)) (1))) (1))) (0), and (flow(c, p_1, 1) (leaf (k) (0), one (p_3) (flow(l, (p_1, p_3), 1) (some (p_6) (leaf (v)) (0), leaf(w) (1))) (1)), leaf (d)) (0))	
MI case (n > 0):	SI case (n = 0):
event reset_loop **any** p_1 **where** @grd_reset **p [{p_1}] = TYPE(p_4)** **then** @act1_reset **m ≔ m / { p_1 }** @act2_reset **n ≔ n / { p_1 }** @act3_reset **o ≔ o / { p_1 }** @act4_reset **p ≔ { p_1 } ⩤ p** **end**	**event reset_loop** **where** @grd_reset **p = TYPE(p_4)** **then** @act1_reset **m ≔ FALSE** @act2_reset **n ≔ FALSE** @act3_reset **o ≔ FALSE** @act4_reset **p ≔ ∅** **end**

Figure 5.58: Application of TR_loop3 in the Example of Figure 5.1, Second Refinement Level

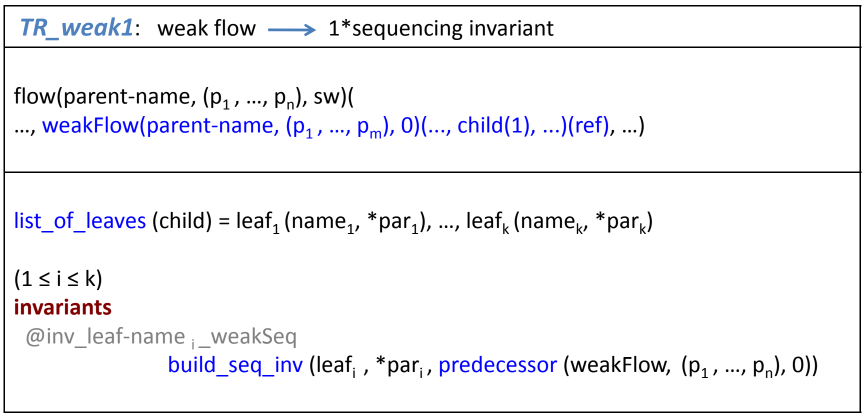
5.4.7 Translating a Weak Sequencing Flow

5.4.7.1 TR_weak1: mapping a weak sequencing flow to (a) invariant(s)

Recall from 4.3.4, considering a weak sequencing flow, the ordering between a weak flow children and the earlier refinement level children, is applied only to the solid child of the weak flow. Obviously there is a separate ordering between the children of a weak flow, managed with TR_leaf4 (Section 5.4.3.4) and TR_leaf8 (Section 5.4.3.8).

TR_weak1 illustrated in Figure 5.59, transforms a weak flow, ($sw = 0$), to one or more invariant(s) which specifies the ordering between the solid child of the weak flow and the previous child. *list_of_leaves* function outputs the final refinement leaf/leaves of the solid child, ($refining = 1$), of the weak flow. Then *predecessor* function is applied to the weak flow to find the previous child of the weak flow. Then in a same way as TR_leaf4, *build_seq_inv* function generate the sequencing invariant.

Figure 5.60 presents the application of this rule in the second refinement level of the example in Figure 5.1. Flow j is a weak flow ($sw = 0$). The only leaf found in *list_of_leaves* function is u. The previous child is leaf h.

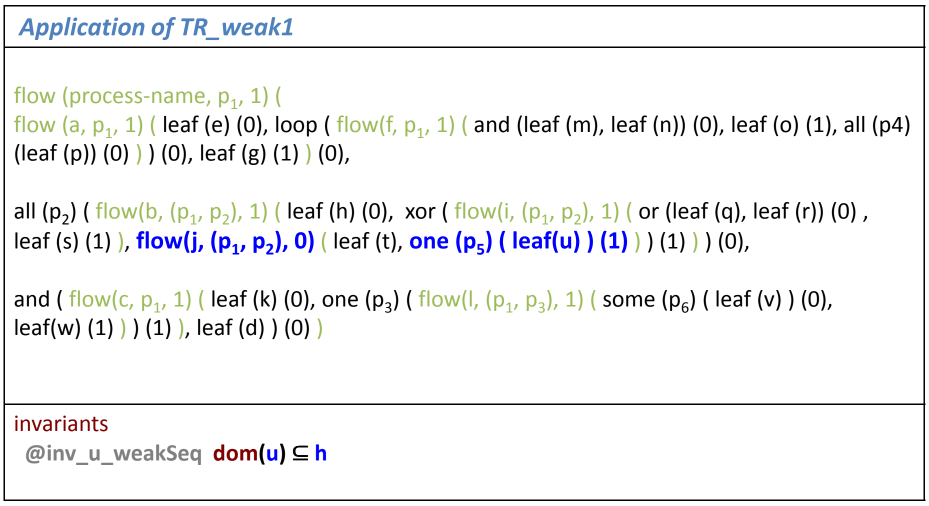

TR_weak1: weak flow ⟶ 1*sequencing invariant

flow(parent-name, (p_1, …, p_n), sw)(
…, weakFlow(parent-name, (p_1, …, p_m), 0)(…, child(1), …)(ref), …)

list_of_leaves (child) = leaf_1 (name_1, *par_1), …, leaf_k (name_k, *par_k)

(1 ≤ i ≤ k)
invariants
@inv_leaf-name $_i$_weakSeq
build_seq_inv (leaf_i , *par_i , predecessor (weakFlow, (p_1, …, p_n), 0))

Figure 5.59: TR_weak1: mapping a weak sequencing flow to (a) guard(s)


Application of TR_weak1

flow (process-name, p_1, 1) (
flow (a, p_1, 1) (leaf (e) (0), loop (flow(f, p_1, 1) (and (leaf (m), leaf (n)) (0), leaf (o) (1), all (p4) (leaf (p)) (0))) (0), leaf (g) (1)) (0),

all (p_2) (flow(b, (p_1, p_2), 1) (leaf (h) (0), xor (flow(i, (p_1, p_2), 1) (or (leaf (q), leaf (r)) (0) , leaf (s) (1)), **flow(j, (p_1, p_2), 0)** (leaf (t), **one (p_5) (leaf(u)) (1)**)) (1))) (0),

and (flow(c, p_1, 1) (leaf (k) (0), one (p_3) (flow(l, (p_1, p_3), 1) (some (p_6) (leaf (v)) (0), leaf(w) (1))) (1)), leaf (d)) (0))

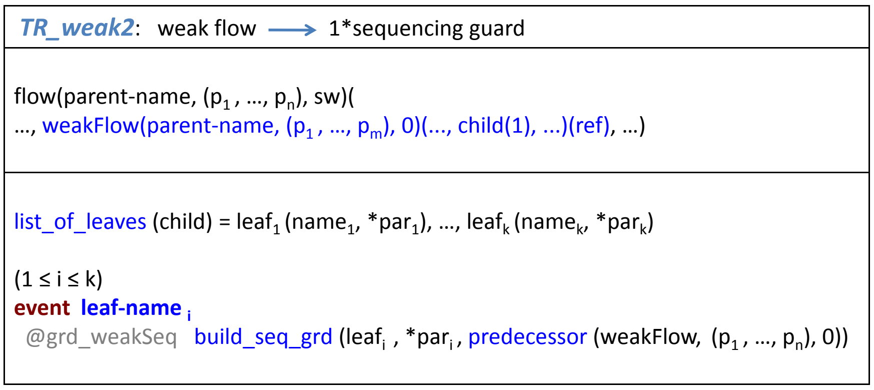
invariants
@inv_u_weakSeq dom(u) ⊆ h

Figure 5.60: Application of TR_weak1 in the Example of Figure 5.1, Second Refinement Level

5.4.7.2 TR_weak2: mapping a weak sequencing flow to (a) guard(s)

TR_weak2 illustrated in Figure 5.61, transforms a weak flow, ($sw = 0$), to one or more guard(s) in the solid child of the weak flow. *list_of_leaves* function outputs the final refinement leaf/leaves of the solid child, ($refining = 1$), of the weak flow. Then *predecessor* function is applied to the weak flow to find the previous child of the weak flow. Then in a same way as TR_leaf8, *build_seq_grd* function generate the sequencing guard. Obviously another guard(s) may be generated in TR_leaf8 (Section 5.4.3.8) to manage the ordering between the children of the weak flow. Also the next event after the weak flow is guarded with solid child of weak flow variable(s) in TR_leaf8.

Figure 5.62 presents the application of this rule in the second refinement level of the example in Figure 5.1.



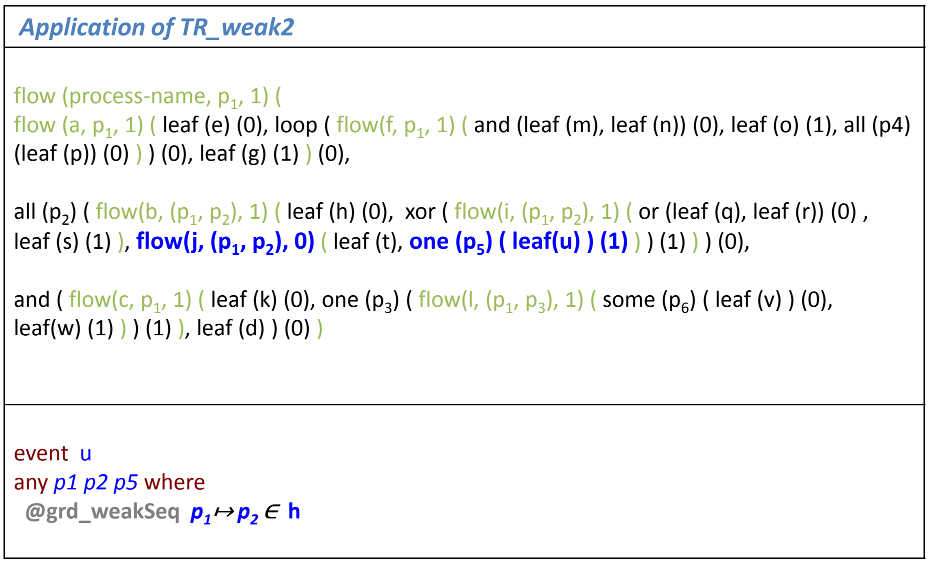
TR_weak2: weak flow ⟶ 1*sequencing guard

flow(parent-name, (p_1 , …, p_n), sw)(
…, weakFlow(parent-name, (p_1 , …, p_m), 0)(…, child(1), …)(ref), …)

list_of_leaves (child) = $leaf_1$ ($name_1$, *par_1), …, $leaf_k$ ($name_k$, *par_k)

($1 \le i \le k$)
event leaf-name $_i$
 @grd_weakSeq build_seq_grd ($leaf_i$, *par_i , predecessor (weakFlow, (p_1 , …, p_n), 0))

Figure 5.61: TR_weak2: mapping a weak sequencing flow to (a) guard(s)



Application of TR_weak2

flow (process-name, p_1, 1) (
flow (a, p_1, 1) (leaf (e) (0), loop (flow(f, p_1, 1) (and (leaf (m), leaf (n)) (0), leaf (o) (1), all (p4) (leaf (p)) (0))) (0), leaf (g) (1)) (0),

all (p_2) (flow(b, (p_1, p_2), 1) (leaf (h) (0), xor (flow(i, (p_1, p_2), 1) (or (leaf (q), leaf (r)) (0) , leaf (s) (1)), **flow(j, (p_1, p_2), 0)** (leaf (t), **one (p_5) (leaf(u)) (1)**)) (1))) (0),

and (flow(c, p_1, 1) (leaf (k) (0), one (p_3) (flow(l, (p_1, p_3), 1) (some (p_6) (leaf (v)) (0), leaf(w) (1))) (1)), leaf (d)) (0))

event u
any *p1 p2 p5* where
 @grd_weakSeq $p_1 \mapsto p_2 \in h$

Figure 5.62: Application of TR_weak2 in the Example of Figure 5.1, Second Refinement Level

5.5 Conclusion

In this chapter, first the language of the atomicity decomposition diagrams was described in a formal way using ABNF (Augmented Backus-Naur Form). Then using translation rules, we defined how an ABNF of an atomicity decomposition diagram can be encoded in terms of Event-B. The translation rules were categorised according to their source element.

Each leaf node in an atomicity decomposition diagram is encoded with a variable (TR_leaf1), and an event (TR_leaf6 and TR_leaf7). The variable corresponding to a leaf is disabled in the body of the corresponding event (TR_leaf11 and TR_leaf12). From a leaf node two guards are encoded in the corresponding event; one guard is to prevent occurrence of same instance of the event for the second time (TR_leaf9 and TR_leaf10); and the aim of the other guard is to control ordering between the corresponding event and the before event (TR_leaf8). To create the actions and guards of an event, we distinguish between a leaf which is a child of a replicator (all-replicator, some-replicator, one-replicator), and a leaf which is not is a child of a replicator. This difference is applied because of the extra replicator parameter which is added to the list of child replicator parameter.

Three types of invariants are encoded. First the typing invariant (TR_leaf2 and TR_leaf3), second the sequencing invariant (TR_leaf4) which specifies the ordering between events and finally the gluing invariant (TR_leaf5, TR_xor1 and TR_one1). A solid line in a diagram is encoded as a gluing invariant.

An xor-constructor causes encoding an invariant (TR_xor2) and a guard (TR_xor3) in each of its children events, to specify the mutual exclusive property between its children.

The one-replicator results in encoding an invariant (TR_one2) and a guard (TR_one3) in its child event, to specify the one execution property.

A loop is encoded as one or more guards (TR_loop1) to prevent the execution of loop event(s) after the execution of next event. Moreover another guard is encoded (TR_loop2) in the next event after loop to prevent its execution in the middle of executions of the loop events. Also a resetting event is encoded (TR_loop3) to reset the control variables of loop in order to enable the loop to execute for another time.

The child with solid line of a weak sequencing decomposition, is encoded as a sequencing invariant (TR_weak1) and a sequencing guard in the solid child event(s) (TR_weak2).

The ordering between the and-constructor, the or-constructor, the all-replicator and the some-replicator children and next child, is managed with sequencing invariant(s) and sequencing guard(s) which are encoded in TR_leaf4 and TR_leaf8 respectively.

The definitions of atomicity decomposition language (ADL) and translation rules helped us to develop tool support for the atomicity decomposition approach. The tool development is presented in Chapter 6. The atomicity decomposition tool makes the process of modelling in Event-B automatic in terms of controlling ordering and relations between events of different refinement levels.

Chapter 6

Tool Development: Atomicity Decomposition Plug-in in Rodin platform

6.1 Introduction

A tool for the atomicity decomposition approach was developed to support the refinement structuring in Event-B. By taking advantage of the extensibility feature of the Event-B toolkit (Rodin platform), we have developed a plug-in as tool support for the atomicity decomposition approach. The Rodin platform serves as a host for the atomicity decomposition plug-in. Developing the atomicity decomposition plug-in in the Rodin platform, helps developers to make Event-B models easier, since using the atomicity decomposition plug-in results in automatic generation of a part of the Event-B model related to the ordering and relationships between events of different refinement levels.

The atomicity decomposition plug-in allows users to structure refinement by using decomposition of an atomic event of an abstract model into some sub-events of a concrete model which execute in a sequential style. First the user can define the atomicity decomposition diagram, then the diagram is automatically transformed to an Event-B model. Currently the atomicity decomposition diagram is build as an instance of the atomicity decomposition meta-model, included in an Event-B machine. However we consider developing a graphical environment for the plug-in as future work. The atomicity decomposition *meta-model* defines all possible atomicity decomposition models; a *model* is a particular instance of the meta-model.

From Section 5.4, the translation rules which correspond to the elements in the last refinement level in a combined atomicity decomposition diagram have been developed in the plug-in. The translation which corresponds to the elements in the earlier refinement levels in a combined atomicity decomposition diagram, are partly developed and need to be examined more and improved as a future work. The atomicity decomposition plug-in has been examined via development of two case studies which are explained in Chapter 7. A perspective of the plug-in in the development of case studies are presented in Chapter 7.

6.2 Architecture and Technologies

Eclipse [72], is a multi-language software development environment comprising an integrated development environment (IDE) and an extensible plug-in system. The Rodin Platform is an Eclipse-based IDE for Event-B and is further extendable with plug-ins. The atomicity decomposition plug-in is developed in the Eclipse environment.

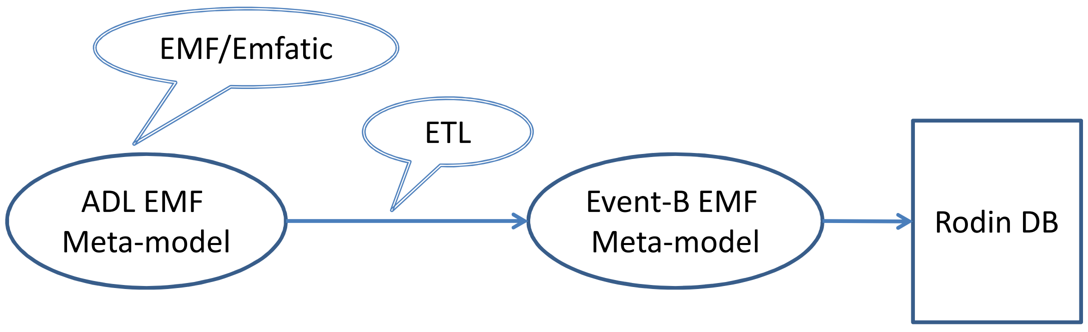


Figure 6.1: Atomicity Decomposition Plug-in Architecture

The development architecture is illustrated in Figure 6.1. The architecture is based on model-driven architecture. In this approach we define the Atomicity Decomposition Language (ADL) specification in an EMF (Eclipse Modelling Framework) [73] meta-model, called source meta-model, and then the source meta-model is transformed to the Event-B EMF meta-model as a target meta-model. The ADL meta-model defines all possible atomicity decomposition models.

The transformation is done using the Epsilon Transformation Language (ETL) [74]. Finally the destination Event-B model is transformed to the Rodin Data Base (DB). The Emfatic text editor is used for creating EMF meta-model. All mentioned technologies are briefly explained below. The explanations are from [72, 75]

Eclipse Modelling Framework (EMF) and Emfatic:

The meta-model describes the structure of the language. EMF [73] can be used to describe the meta-model of the atomicity decomposition language. We decided to use EMF technology since it has advantages in our plug-in development, some of them are listed here:

- Once the EMF meta-model is specified, we can generate the corresponding Java implementation classes from this model. EMF provides the possibility to safely extended the generated code by hand.
- With EMF we can make our model explicit which helps to provide clear visibility of the model.
- EMF also provides change notification functionality to the model in case of model changes.
- EMF will generate interfaces to create our own objects. Therefore it helps us to keep our application clean from the individual implementation classes.
- Another advantage is that we can regenerate the Java code from the model at any point in time.

Emfatic [73] is a text editor supporting navigation, editing, and conversion of EMF models, using a compact and human-readable syntax similar to Java.

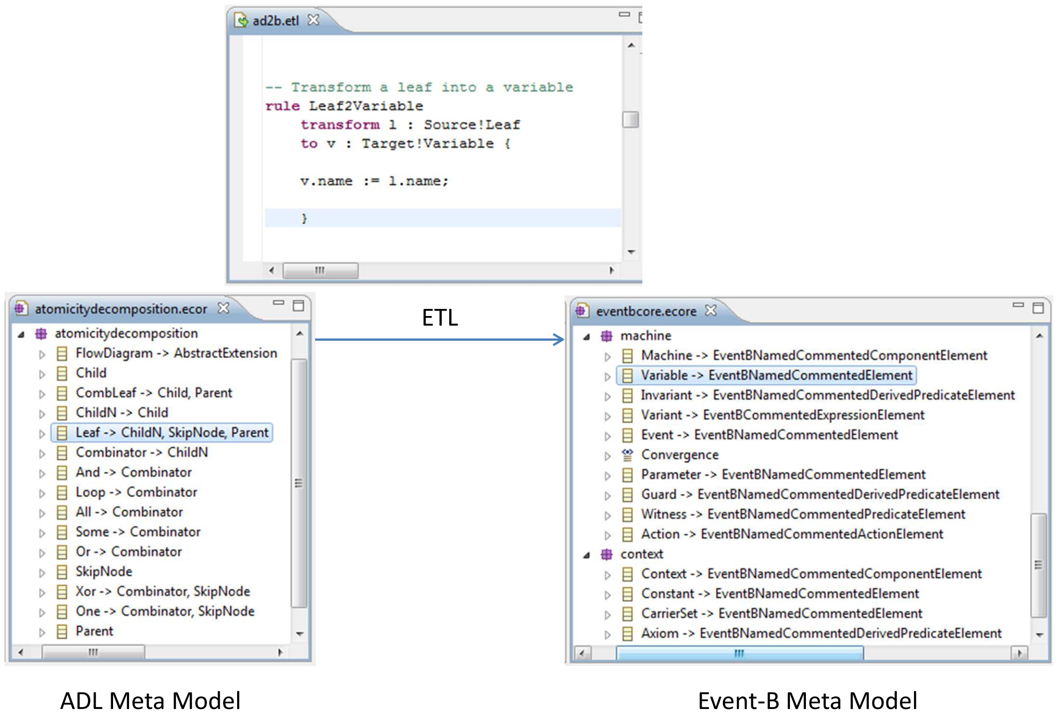
Epsilon Transformation Language (ETL):

ETL [74], is a rule-based model-to-model transformation language. We benefit from features of ETL. The prominent features are as follows:

- Transform many input to many output models
- Ability to query/navigate/modify both source and target models
- Automated rule execution
- Rule inheritance
- Guarded rules

Figure 6.2 presents a view of the ADL EMF meta-model on the left side. Using ETL rules, some components of this meta-model are transformed to some components of the Event-B EMF meta-model on the right. As an example, Figure 6.2 illustrates how the translation rule TR_leaf1 (Section 5.4.3.1) is encoded as an ETL rule. This rule transforms a *leaf* from the ADL meta-model (as the source meta-model) to a *variable* in the Event-B meta-model (as the target meta-model). In the body of rule the name of the target component (variable) is assigned to the name of the source component (leaf).

Another example of an ETL rule in presented in Figure 6.3. This rule is corresponded to the translation rule TR_xor1 (Section 5.4.4.1) in the MI case, which transforms a solid xor-constructor to a gluing invariant. The rule is guarded for a solid xor and the MI case. In the body of the rule, first the name of the invariant is assigned, then the predicate

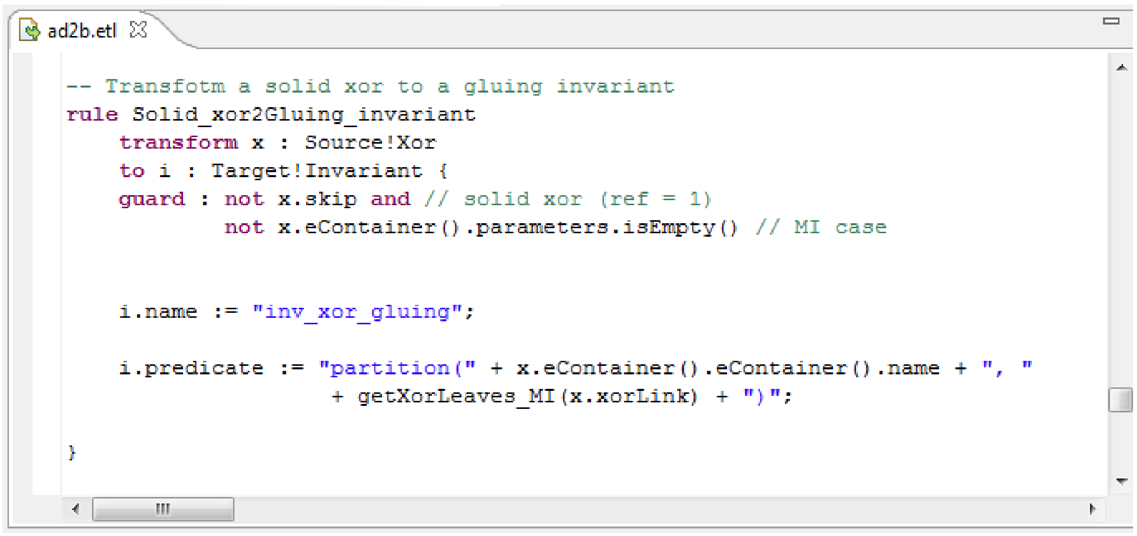


```
-- Transform a leaf into a variable
rule Leaf2Variable
    transform l : Source!Leaf
    to v : Target!Variable {

    v.name := l.name;

    }
```

ad2b.etl
atomicitydecomposition.ecor
atomicitydecomposition
FlowDiagram -> AbstractExtension
Child
CombLeaf -> Child, Parent
ChildN -> Child
Leaf -> ChildN, SkipNode, Parent
Combinator -> ChildN
And -> Combinator
Loop -> Combinator
All -> Combinator
Some -> Combinator
Or -> Combinator
SkipNode
Xor -> Combinator, SkipNode
One -> Combinator, SkipNode
Parent
ETL
eventbcore.ecore
machine
Machine -> EventBNamedCommentedComponentElement
Variable -> EventBNamedCommentedElement
Invariant -> EventBNamedCommentedDerivedPredicateElement
Variant -> EventBCommentedExpressionElement
Event -> EventBNamedCommentedElement
Convergence
Parameter -> EventBNamedCommentedElement
Guard -> EventBNamedCommentedDerivedPredicateElement
Witness -> EventBNamedCommentedPredicateElement
Action -> EventBNamedCommentedActionElement
context
Context -> EventBNamedCommentedComponentElement
Constant -> EventBNamedCommentedElement
CarrierSet -> EventBNamedCommentedElement
Axiom -> EventBNamedCommentedDerivedPredicateElement
ADL Meta Model
Event-B Meta Model

Figure 6.2: Atomicity Decomposition Language, EMF Meta-model

of the invariant is assigned to a partition operator. $x.econtainer().econtainer().name$ returns the parent name of the xor-constructor and $getXorLeaves\_MI(x.xorLink)$ outputs the list of xor-constructor leaves' names.



```
-- Transfotm a solid xor to a gluing invariant
rule Solid_xor2Gluing_invariant
    transform x : Source!Xor
    to i : Target!Invariant {
    guard : not x.skip and // solid xor (ref = 1)
            not x.eContainer().parameters.isEmpty() // MI case


    i.name := "inv_xor_gluing";

    i.predicate := "partition(" + x.eContainer().eContainer().name + ", "
                + getXorLeaves_MI(x.xorLink) + ")";

}
```

Figure 6.3: An ETL rule, Corresponded to TR_xor1 (Section 5.4.4.1)

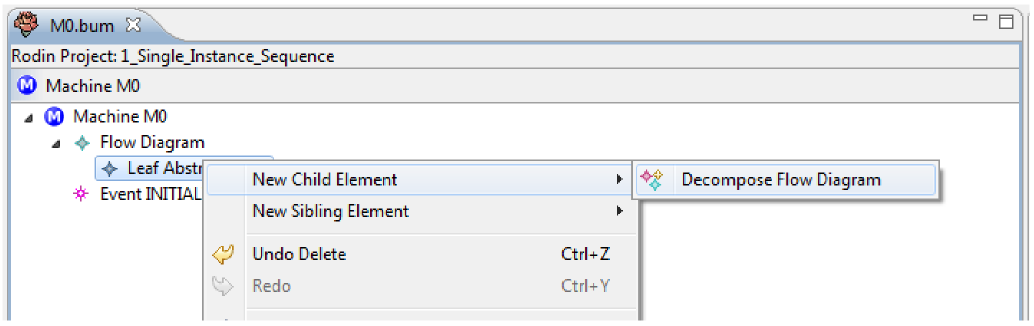
6.3 User Interface

This section briefly describes how the atomicity decomposition plug-in is used. As mentioned in Section 6.1, currently the atomicity decomposition diagram is built as an instance of the ADL meta-model included in a Event-B machine. The user can add each element of the atomicity decomposition diagram in the appropriate place when right clicking on an element. For example, in Figure 6.4(a), a new flow can be added to a leaf when right clicking on the leaf, in order to define a new decomposition flow of the leaf. After finishing the atomicity decomposition model, like the example in Figure 6.4(b), the atomicity decomposition model can be transformed to the Event-B model. The user accesses the transformation feature when right clicking on the machine, presented in Figure 6.4(c). Behind the "Transform to Event-B" submenu, the ETL transformation rules are applied and the Event-B model is generated. Figure 6.5 presents the generated Event-B model for the atomicity decomposition model of Figure 6.4(b).

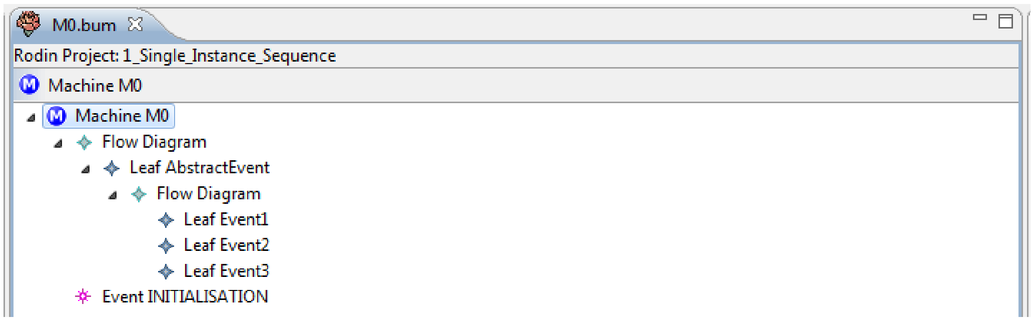
6.4 Conclusion

The Rodin platform, as an Event-B tool, serves as a host for the atomicity decomposition plug-in developed to give tool support to the atomicity decomposition approach. The theory behind the atomicity decomposition plug-in has been gradually presented in Chapter 3, Chapter 4 and Chapter 5; and the applications to case studies will be presented in Chapter 7. We benefit from some features of EMF (Eclipse Modelling Framework) and ETL (Epsilon Transformation Language) to create the ADL meta-model and transform it to the Event-B meta-model. We consider developing a graphical user interface to create the ADL meta-model in a diagrammatic view, as future work.

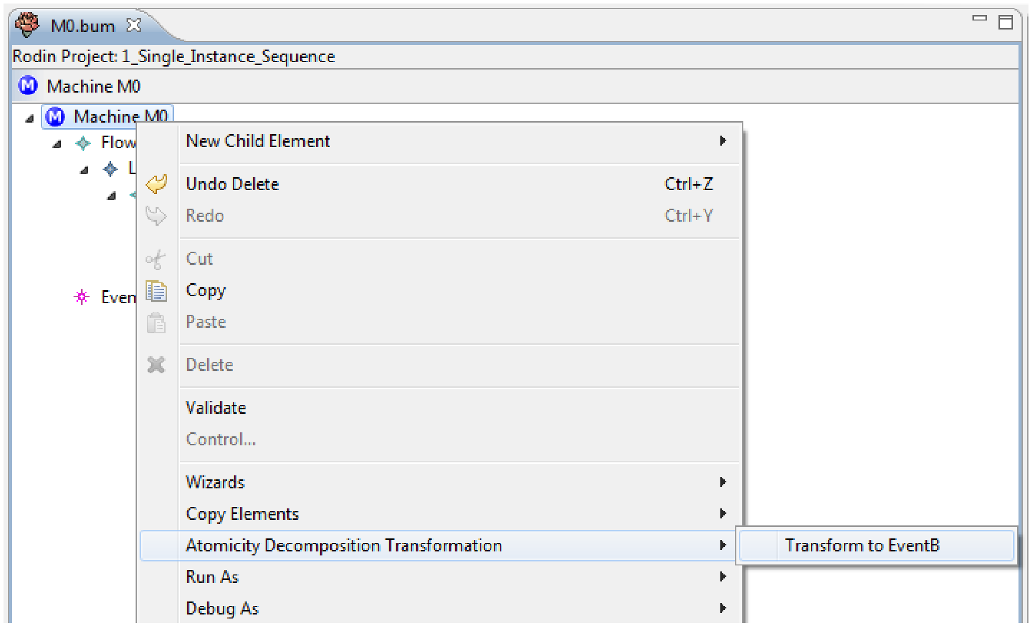
The atomicity decomposition plug-in supports automatic generation of Event-B models in terms of ordering between events and relationships between refinement levels. Extra requirements can be added manually to the automatic Event-B model. Automatic generation aims to decrease the effort of modelling complex systems in Event-B, and contributes to improve the development process of a complex system.



(a) Creating an Instance of the Atomicity Decomposition Meta-model

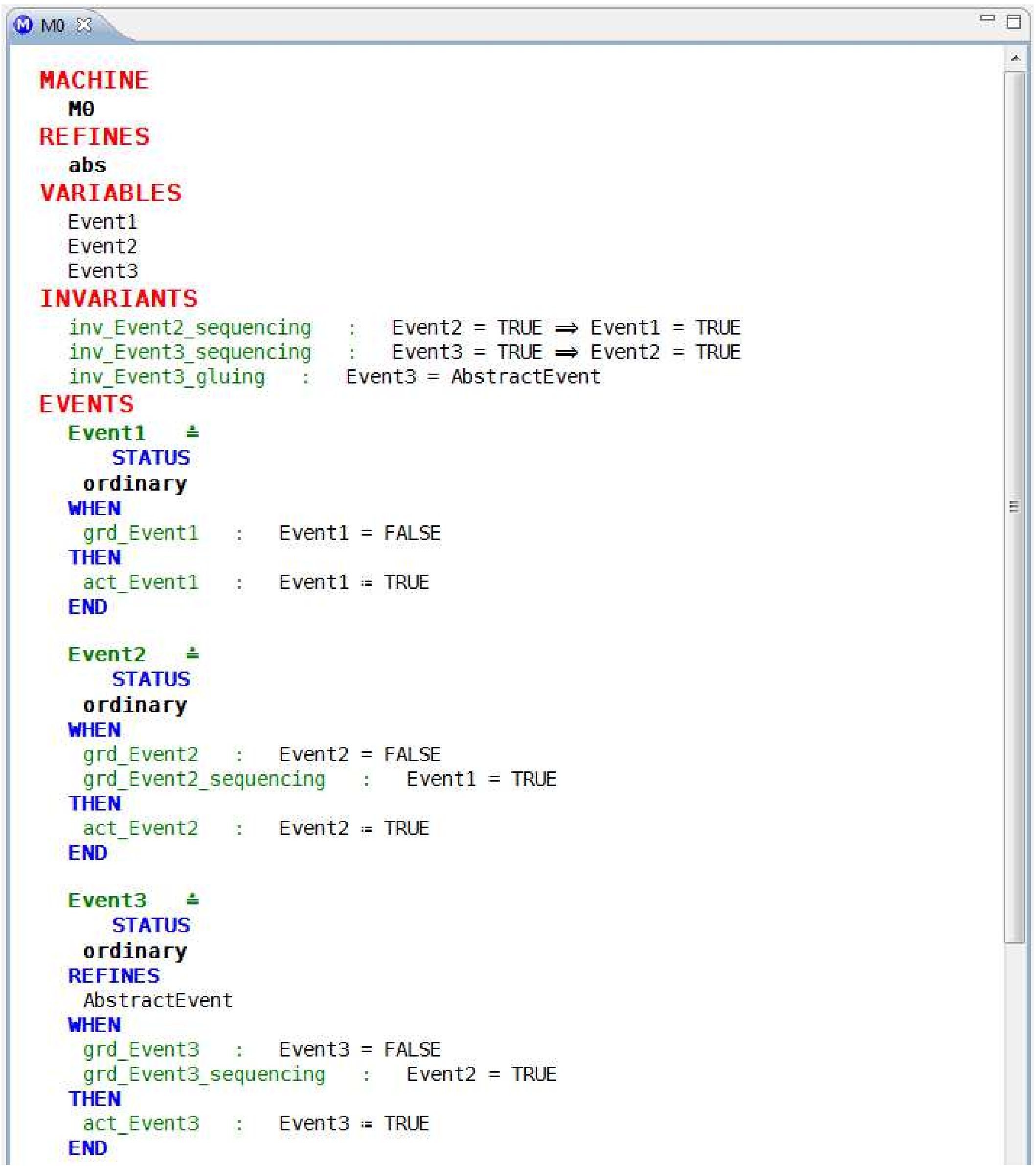


(b) The Atomicity Decomposition Model



(c) Transforming to the Event-B Model

Figure 6.4: User Interface



```
MACHINE
  M0
REFINES
  abs
VARIABLES
  Event1
  Event2
  Event3
INVARIANTS
  inv_Event2_sequencing   :   Event2 = TRUE ⇒ Event1 = TRUE
  inv_Event3_sequencing   :   Event3 = TRUE ⇒ Event2 = TRUE
  inv_Event3_gluing   :   Event3 = AbstractEvent
EVENTS
  Event1   ≙
      STATUS
   ordinary
  WHEN
   grd_Event1   :   Event1 = FALSE
  THEN
   act_Event1   :   Event1 ≔ TRUE
  END

  Event2   ≙
      STATUS
   ordinary
  WHEN
   grd_Event2   :   Event2 = FALSE
   grd_Event2_sequencing   :   Event1 = TRUE
  THEN
   act_Event2   :   Event2 ≔ TRUE
  END

  Event3   ≙
      STATUS
   ordinary
  REFINES
   AbstractEvent
  WHEN
   grd_Event3   :   Event3 = FALSE
   grd_Event3_sequencing   :   Event2 = TRUE
  THEN
   act_Event3   :   Event3 ≔ TRUE
  END
```

Figure 6.5: Event-B Model of the Instance of the Atomicity Decomposition Model in Figure 6.4

Chapter 7

Case Studies

7.1 Introduction

We have developed two case studies, using the atomicity decomposition approach, initially manually before the plug-in was developed; and later with the plug-in which has been outlined in the previous chapter. The existing atomicity decomposition approach, presented in [24], has been evaluated during manual development of the case studies. Manual development of these case studies helped us to improve the atomicity decomposition approach. As a result, some new patterns have been discovered which helped us to define the atomicity decomposition language and translation rules in a formal description, followed by developing tool support for the approach. The discovered patterns have been outlined in Chapter 4, the formal description of atomicity decomposition language and translation rules have been defined in Chapter 5 and the tool development has been described in Chapter 6. The evaluation and methodological results of these case studies are explained later in Chapter 8. Also Chapter 8 will outline how the manual development helped to improve the atomicity decomposition patterns and language. Through our experiment in these developments we found out that how the atomicity decomposition approach can help us to structure refinement and how much it is beneficial in modelling the requirements of different phases using the diagrammatic notation of the atomicity decomposition approach.

After defining the language and translation rules in a formal description and developing tool support for the atomicity decomposition approach, we modelled the case studies for second time using our plug-in in a semi-automatic approach. The reason we call it semi-automatic is that, part of the Event-B model which is related to the ordering requirements between events is generated automatically with the plug-in. The other requirements have been added manually to the generated model, commented with *manually* in the Event-B model.

This chapter presents the automatic developments of the case studies. The major differences between the manual Event-B model and the automatic Event-B model are presented for each case study.

First, Section 7.2 presents the development of the Media Channel system. The work presented in this section is published in “Formal Methods for Components and Objects” (FMCO) 2009 conference [1]. The complete version of the automatic Event-B model of the media channel system, which is developed using the atomicity decomposition plug-in, is presented in Appendix A. And the complete version of the manual Event-B model is available online [1].

The second case study, the BepiColombo system, is presented in Section 7.3, and published in the “Nasa Formal Methods” (NFM) 2011 symposium [2]. Applying both atomicity decomposition and model decomposition to a large system is one of the motivations for developing the second case study. Moreover the methodological results reached during the first case study development, have been evaluated in the development of second larger system. The complete version of the automatic Event-B model of the BepiColombo system, which is developed using the atomicity decomposition plug-in, is presented in Appendix B. And the complete version of the manual Event-B model is available online [2].

Recall from previous motivations, refinement in Event-B helps developers to do incremental modelling of complex systems. However refinement does not solve the problems of building the models of complicated and difficult systems completely. Event-B refinement is not able to illustrate explicit connections between abstract and concrete events through different levels of refinement. It motivates us to apply refinement and the atomicity decomposition approach to large case studies.

Our approach in developing the case studies is incremental. Developing a system in incremental steps means it starts with a very abstract model and more details are added to model gradually in the refinement levels. In other words, the gap between refinement levels is not too great. We add some intermediate model to reduce the abstraction gap between refinement levels.

The content of each case study section, begins with the review of the requirements of the system. Then the abstract specification of the system is introduced followed by five refinement levels for the media channel system, and three refinement levels followed by a model decomposition followed by two refinement levels for the BepiColombo system. Finally the major differences between manual Event-B model and the automatic Event-B model are presented, and it is followed by a review of the proof obligations.

[1]http://eprints.ecs.soton.ac.uk/21261/
[2]http://eprints.ecs.soton.ac.uk/22048/

7.2 Media Channel System (Published in FMCO 2009 Conference)

7.2.1 Overview of the Media Channel System

The Media Channel Properties: All properties described in this section are from [76]. Each media channel has one source, one sink, a codec type and a specific direction. A media channel is point-to-point and dynamic, established for transferring data, called medium. A media channel is established between two endpoints. An endpoint is any source or sink of a media stream. A point-to-point media channel is simply illustrated in Figure 7.1.

Figure 7.1: A Simple Image of the Media Channel between Two Endpoints

A Codec is a specific data format by which data is encoded. The codec choice in the media channel is dynamic. This means that each endpoint of the channel is allowed to change the codec at any time in the middle of data transfer. Although each endpoint can interpret more than one codec, the source and sink of a media channel have to know with which codec they are supposed to send or receive. So any two endpoints of a media channel should have at least one common codec.

The Important Protocol Rules: Either end of a channel, sender or receiver, can attempt to open a media channel by sending an open signal. The other end can respond affirmatively with *openAck* (open acknowledge) or negatively with close. A media flow can be established between two media endpoints if and only if both media endpoints agree.

Each open signal carries the medium being requested, and a descriptor. A descriptor is a record in which an endpoint describes itself as a receiver of media. A descriptor contains an IP address, port number, and a set of codecs that the endpoint can handle. If the endpoint does not wish to receive media, then the only offered codec is *noMedia*. Each *openAck* signal also carries a descriptor, describing the channel acceptor as a receiver of media.

A selector is a response to a descriptor. A selector is a record in which an endpoint describes itself as a sender of media. It contains the identification of the descriptor it is responding to, the IP address of the sender, and the port number of the sender. If the selecting endpoint does not wish to send media, then the selector contains *noMedia*. Otherwise, it contains a single codec selected from the set of codecs in the descriptor. The only legal response to a descriptor *noMedia* is a selector *noMedia*.

7.2.1.1 Requirements for Establishing a Media Channel

After sending an *open* signal with the initiator side of the channel, and sending an *openAck* signal with the other side, called the acceptor, both endpoints have to respond to descriptors carried by *open* and *openAck* signal, by sending a select signal carrying a selector. As said before, it is a rule of the system that a selector should be sent in response to receiving a descriptor. A media channel is established with the endpoint which receives a real codec in a select signal. Figure 7.2 shows the steps involved in establishing a media channel.

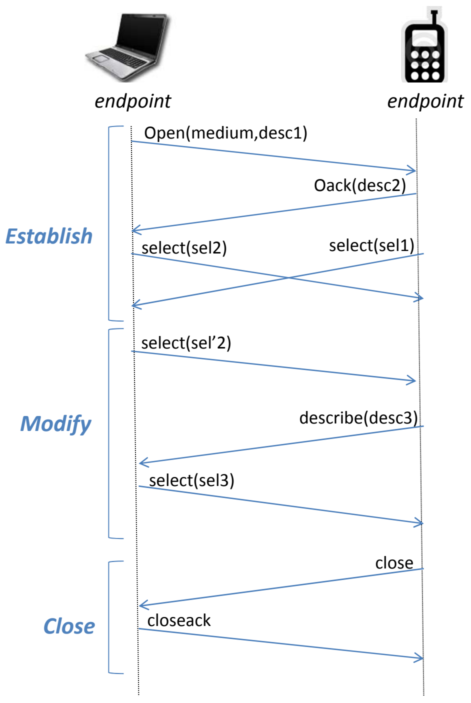


Figure 7.2: Protocol of the Media Channel System

7.2.1.2 Requirements for Modifying an Established Media Channel

Modifying an established media channel may involve changing of the codec used in transferring data or changing the port of each endpoint. At any time after sending the first selector in response to a descriptor, an endpoint can choose a new codec from the set of codecs in the descriptor, send it as a selector in a select signal, and begin to send media in the new codec. In Figure 7.2, select (sel'2) shows this possibility.

At any time after sending or receiving *oAck*, an endpoint can send a new descriptor in a describe signal. The endpoint that receives the new descriptor must begin to act according to the new descriptor. This might mean sending to a new address or choosing a new codec. In any case, the receiver of the descriptor must respond with a new selector in a select signal, if only to show that it has received the descriptor. In Figure 7.2, descriptor3 and selector3 illustrate this interaction. Finally at any time after sending or receiving *oAck*, an endpoint can send a new port and describe itself by a new port.

7.2.1.3 Requirements for Closing an Established Media Channel

As can be seen in Figure 7.2, either endpoint can close the media channel at any time by sending a close signal, which must be acknowledged by the other end with a *closeAck* (close Acknowledge).

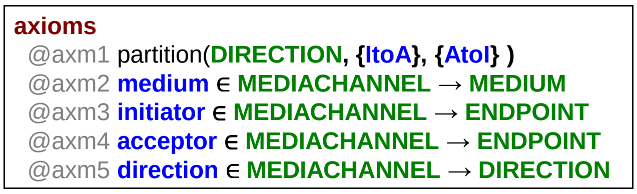
7.2.2 Abstract Specification

7.2.2.1 Static Part of the Specification

The abstract context, *C1*, consists of five sets and six constants. As mentioned in Section 2.4.2, the context contains the static part of the system.

ENDPOINT (set of endpoints of system which play the role of source and sink of a media channel), *MEDIUM* (set of media which can sent or received in the process of transferring data), *CODEC* (set of all existing codecs), *MEDIACHANNEL* (set of all potential media channels), *DIRECTION* (an enumerated set showing the direction of a media channel which can be form Initiator to Acceptor (*ItoA*), or from Acceptor to Initiator (*AtoI*)).

As mentioned in the previous section, each media channel has a specific endpoint as its initiator, a specific endpoint as its acceptor, a specific direction, and a specific medium. These properties are modelled as total functions, illustrated in Figure 7.3. A total function guarantees that each media channel has exactly one medium, one initiator, one acceptor and one direction. These functions are considered as constants because they do not change after establishing a media channel. Whereas the codec property of a media channel is considered as a variable in the model, since it is a dynamic part and can change after establishing a media channel.



```
axioms
  @axm1 partition(DIRECTION, {ItoA}, {AtoI} )
  @axm2 medium ∈ MEDIACHANNEL → MEDIUM
  @axm3 initiator ∈ MEDIACHANNEL → ENDPOINT
  @axm4 acceptor ∈ MEDIACHANNEL → ENDPOINT
  @axm5 direction ∈ MEDIACHANNEL → DIRECTION
```

Figure 7.3: Context C1, Media Channel System

7.2.2.2 Events and Dynamic Part of the Specification

In the abstract model, *M0*, the main goals of the system are modelled. The most abstract events are illustrated in Figure 7.4, using the diagram explained in Section 4.3.1. First a media channel is established by execution of *establishMediaChannel* event, then it can be modified for zero or more times by execution of *modify* event and then can be closed by execution of *close* event.

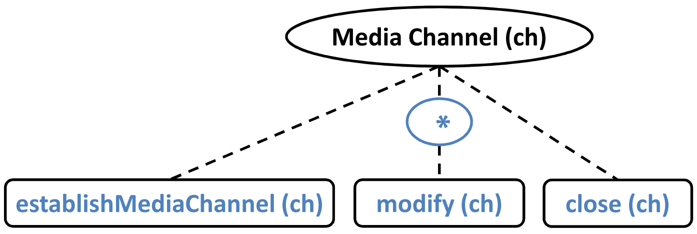
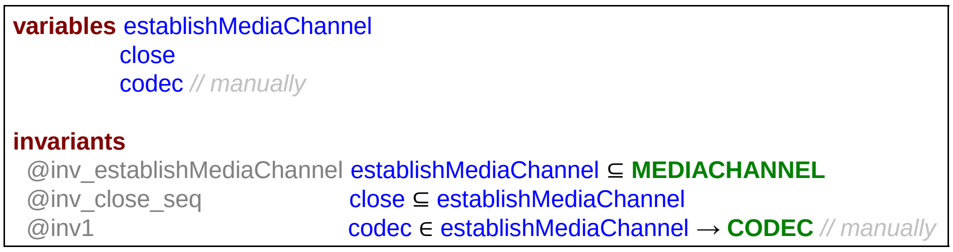


Figure 7.4: The Atomicity Decomposition Diagram, M0, Media Channel System

As explained in Section 4.2, the ordering between events is modelled using some control variables, invariants and guards. As described in Section 4.2.3 we do not consider a variable for a loop event. In machine *M0*, there are two control variables and one manual variable. Figure 7.5 presents the variables and invariants of *M0*. Control variables, invariants, etc, are added automatically by the tool; manual variables, invariants, etc, are added manually by the user and are commented with *manually*.

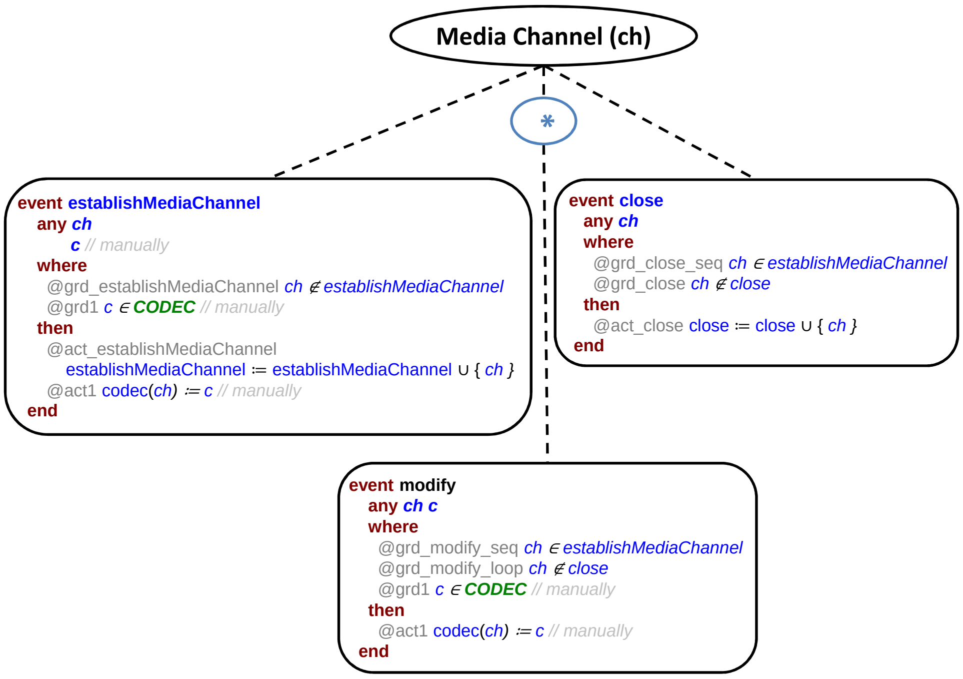
For each event there is a control variable with same name as the event, and if one event is executed after another one, the later variable is a subset of the former one. For example, the *close* event can be executed only after execution of *establishMediaChannel* event, so invariant *inv_close_seq* describes the *close* variable as a subset of the *establishMediaChannel* variable. Variable *codec* is modelled manually. It is a total function, specifying the codec property of an established media channel. Variable *codec* and the corresponding invariant are added manually, the other invariants and variables are generated with the tool.



```
variables establishMediaChannel
          close
          codec // manually

invariants
  @inv_establishMediaChannel establishMediaChannel ⊆ MEDIACHANNEL
  @inv_close_seq             close ⊆ establishMediaChannel
  @inv1                      codec ∈ establishMediaChannel → CODEC // manually
```

Figure 7.5: Variables and Invariants, M0, Media Channel System

The abstract events are illustrated in Figure 7.6. There are some guards for controlling the sequencing of events. In the first event, *establishMediaChannel*, one channel, *ch*, is added to the *establishMediaChannel* variable, and then *modify* and *close* events can executed only for a *ch* which belongs to *establishMediaChannel* variable, which is checked in *grd_modify_seq* and *grd_close_seq* guards in *modify* and *close* events. Guards and actions related to the codec property of the media channel are modelled manually. As can be seen in Figure 7.6, in the *establishMediaChannel* event, the codec property of the channel is initialized. Later in the *modify* event, the codec can be changed to a new value. The Event-B model of loop constructor follows the scheme explained in Section 4.2.3.



```
event establishMediaChannel
  any ch
      c // manually
  where
    @grd_establishMediaChannel ch ∉ establishMediaChannel
    @grd1 c ∈ CODEC // manually
  then
    @act_establishMediaChannel
      establishMediaChannel ≔ establishMediaChannel ∪ { ch }
    @act1 codec(ch) ≔ c // manually
end
```

```
event close
  any ch
  where
    @grd_close_seq ch ∈ establishMediaChannel
    @grd_close ch ∉ close
  then
    @act_close close ≔ close ∪ { ch }
end
```

```
event modify
  any ch c
  where
    @grd_modify_seq ch ∈ establishMediaChannel
    @grd_modify_loop ch ∉ close
    @grd1 c ∈ CODEC // manually
  then
    @act1 codec(ch) ≔ c // manually
end
```

Figure 7.6: Event-B Model, M0, Media Channel System

7.2.3 1st Refinement: Breaking the Atomicity of Establish Media Channel

In the abstract model, we saw that a media channel is established in a single atomic step. It provides simplicity in the abstract level. However, in the real protocol, explained in Section 7.2.1, establishing a media channel is not atomic. Instead, an *open* request should be sent by the initiator endpoint and should be responded to by an *openAck* signal from the acceptor endpoint. After receiving a select signal carrying a real codec, selected from the set of codecs, the media channel can be established. This scenario is illustrated in Figure 7.7. Two scenarios are possible. First, as illustrated on the left of the figure, the requester sends an *open* signal carrying a descriptor with a real set of codec. It means the requester is the receiver. The acceptor sends an *openAck* signal carrying a descriptor without real codecs, since the acceptor is the sender. In this point, the acceptor, which has received a real set of codecs, selects a codec from the set and establishes the channel. The second scenario on the right of the figure illustrates when the requester is the sender. In this case the requester sends an *open* signal carrying a descriptor without real codec, and the acceptor responds by sending an *openAck* signal carrying a descriptor with a real set of codecs. In this point, the requester, which has received a real set of codecs, selects a codec from the set and establishes the channel.

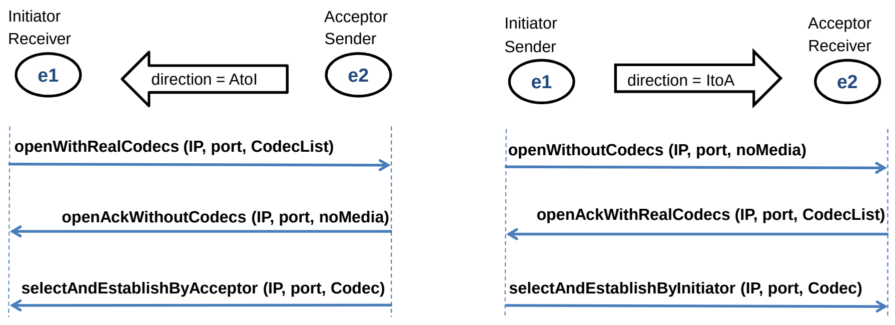

Initiator
Receiver
Acceptor
Sender
e1
direction = AtoI
e2
openWithRealCodecs (IP, port, CodecList)
openAckWithoutCodecs (IP, port, noMedia)
selectAndEstablishByAcceptor (IP, port, Codec)
Initiator
Sender
Acceptor
Receiver
e1
direction = ItoA
e2
openWithoutCodecs (IP, port, noMedia)
openAckWithRealCodecs (IP, port, CodecList)
selectAndEstablishByInitiator (IP, port, Codec)

Figure 7.7: Establish Media Channel Scenario

Breaking the atomicity of establishing a media channel is outlined in the atomicity decomposition diagrams in Figure 7.8.

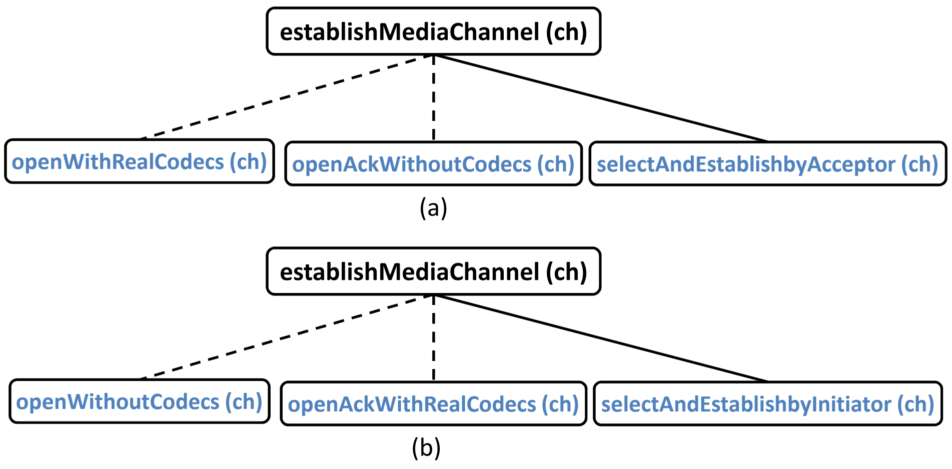
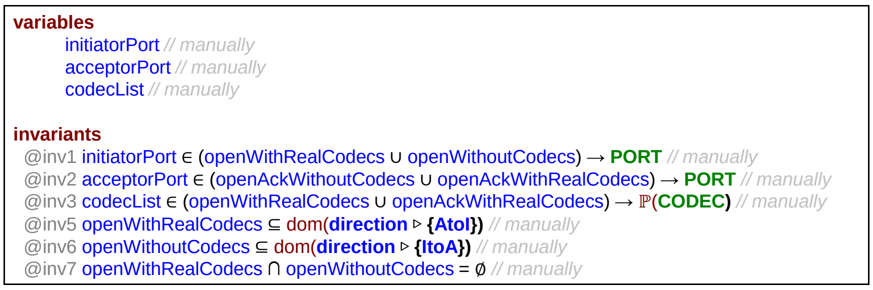

establishMediaChannel (ch)
openWithRealCodecs (ch)
openAckWithoutCodecs (ch)
selectAndEstablishbyAcceptor (ch)
(a)
establishMediaChannel (ch)
openWithoutCodecs (ch)
openAckWithRealCodecs (ch)
selectAndEstablishbyInitiator (ch)
(b)

Figure 7.8: Breaking the Atomicity of Establish a Media Channel, M1

Possible event traces of establishing a media channel are:

$< openWithRealCodecs,\ openAckWithoutCodecs,\ selectAndEstablishbyAcceptor >$
$< openWithoutCodecs,\ openAckWithRealCodecs,\ selectAndEstablishbyInitiator >$

The control variables and invariants which control the sequencing between events are generated automatically. There are three manual variables defined in machine *M1* in order to model the initiator port, acceptor port and codec set of a media channel. The manual variables and invariants are presented in Figure 7.9. *inv1*, *inv2* and *inv3* define the new properties of a media channel. *inv5* specifies that the channels which contain an *open* signal carrying a real set of codecs are always from acceptor to initiator, (direction = AtoI). Similarly, *inv6* specifies that the channels which contain an *open* signal without real codec are always from initiator to acceptor, (direction = ItoA). Finally *inv7* specifies that these two kinds of channels are disjoint.



```
variables
        initiatorPort // manually
        acceptorPort // manually
        codecList // manually

invariants
  @inv1 initiatorPort ∈ (openWithRealCodecs ∪ openWithoutCodecs) → PORT // manually
  @inv2 acceptorPort ∈ (openAckWithoutCodecs ∪ openAckWithRealCodecs) → PORT // manually
  @inv3 codecList ∈ (openWithRealCodecs ∪ openAckWithRealCodecs) → ℙ(CODEC) // manually
  @inv5 openWithRealCodecs ⊆ dom(direction ▷ {AtoI}) // manually
  @inv6 openWithoutCodecs ⊆ dom(direction ▷ {ItoA}) // manually
  @inv7 openWithRealCodecs ∩ openWithoutCodecs = ∅ // manually
```

Figure 7.9: Manual Variables and Invariants, M1, Media Channel System

There is a gluing invariant in machine M1 which define the relations between abstract variable and concrete variables. The gluing invariant which generated automatically is :

$@inv\_gluing \quad selectAndEstablishbyAcceptor \cup selectAndEstablishbyInitiator = establishMediaChannel$

Since two events refine the abstract event, *establishMediaChannel*, the union of the corresponding control variables is equal to the abstract variable.

The sequencing guards and actions are generated automatically. The Event-B model of the first diagram in Figure 7.8 is shown in Figure 7.10, Figure 7.11 and Figure 7.12. Obviously the sub-events with dashed lines, *openWithRealCodecs* and *openAckWithoutCodecs*, are new events which refine *skip* and the event with solid line, *selectAndEstablishbyAcceptor* event refines the abstract event, *establishMediaChannel*. The other properties of the media channel is assigned manually in each event. Codec set and initiator port of a media channel is initialized in *openWithRealCodecs* event; Acceptor port is initialized in *openAckWithoutCodecs* event and the selected codec is initialized in *selectAndEstablishbyAcceptor* event.

7.2.4 2nd Refinement: Breaking the Atomicity of Modify Media Channel

Up to this level, modify was considered as an atomic event which was done by one single event and simply changes the codec of an established media channel. In this refinement we break the atomicity of the *modify* event.

As presented in Figures 7.13, 7.14 and 7.15, there are three ways of modifying the properties of an established channel.

First, as it is presented in Figure 7.13, after establishing a media channel, the sender endpoint can select a new codec from the set of acceptable codecs of the other endpoint,

```
event openWithRealCodecs
  any ch
     cl // manually
     p // manually
     i // manually
  where
   @grd_openWithRealCodecs ch ∉ openWithRealCodecs
   @grd1 ch ∉ openWithoutCodecs // manually
   @grd2 cl ⊆ CODEC // manually
   @grd3 cl ≠ ∅ // manually
   @grd4 p ∈ PORT // manually
   @grd5 i ∈ IP // manually
   @grd6 i ∈ dom(endpointIp∼) // manually
   @grd7 initiator(ch) = endpointIp∼(i) // manually
   @grd8 direction(ch) = AtoI // manually
  then
   @act_openWithRealCodecs
     openWithRealCodecs ≔ openWithRealCodecs ∪ { ch }
   @act1 codecList(ch) ≔ cl // manually
   @act2 initiatorPort(ch) ≔ p // manually
 end
```

Figure 7.10: Event-B Model, M1, Media Channel System

```
event openAckWithoutCodecs
  any ch
      cl // manually
      p // manually
      i // manually
  where
   @grd_openAckWithoutCodecs_seq ch ∈ openWithRealCodecs
   @grd_openAckWithoutCodecs ch ∉ openAckWithoutCodecs
   @grd1 cl ⊆ CODEC // manually
   @grd2 cl = ∅ // manually
   @grd3 p ∈ PORT // manually
   @grd4 i ∈ IP // manually
   @grd5 i ∈ dom(endpointIp∼) // manually
   @grd6 acceptor(ch) = endpointIp∼(i) // manually
 then
   @act_openAckWithoutCodecs
     openAckWithoutCodecs ≔ openAckWithoutCodecs ∪ {ch}
   @act1 acceptorPort(ch) ≔ p // manually
 end
```

Figure 7.11: Event-B Model, M1, Media Channel System

```
event selectAndEstablishbyAcceptor refines establishMediaChannel
  any ch
      c // manually
  where
   @grd_selectAndEstablishbyAcceptor_seq ch ∈ openAckWithoutCodecs
   @grd_selectAndEstablishbyAcceptor ch ∉ selectAndEstablishbyAcceptor
   @grd1 c ∈ codecList(ch) // manually
  then
   @act_selectAndEstablishbyAcceptor
     selectAndEstablishbyAcceptor ≔ selectAndEstablishbyAcceptor ∪ { ch }
   @act1 codec(ch) ≔ c // manually
 end
```

Figure 7.12: Event-B Model, M1, Media Channel System

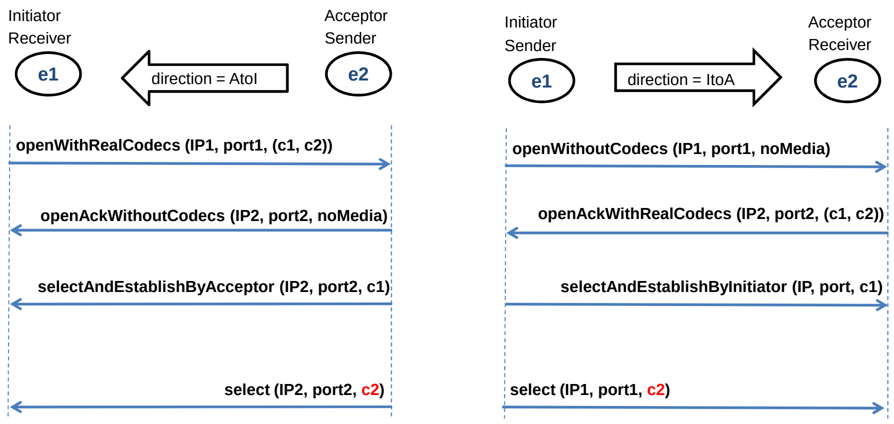

Initiator
Receiver
e1
direction = AtoI
Acceptor
Sender
e2
openWithRealCodecs (IP1, port1, (c1, c2))
openAckWithoutCodecs (IP2, port2, noMedia)
selectAndEstablishByAcceptor (IP2, port2, c1)
select (IP2, port2, c2)
Initiator
Sender
e1
direction = ItoA
Acceptor
Receiver
e2
openWithoutCodecs (IP1, port1, noMedia)
openAckWithRealCodecs (IP2, port2, (c1, c2))
selectAndEstablishByInitiator (IP, port, c1)
select (IP1, port1, c2)

Figure 7.13: Modify Set of Codecs of a Media Channel by Selector Scenario

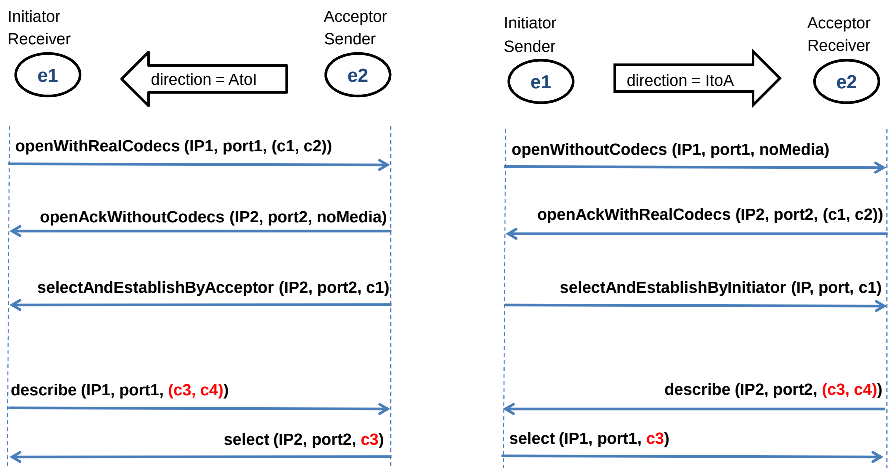

Initiator
Receiver
e1
direction = AtoI
Acceptor
Sender
e2
openWithRealCodecs (IP1, port1, (c1, c2))
openAckWithoutCodecs (IP2, port2, noMedia)
selectAndEstablishByAcceptor (IP2, port2, c1)
describe (IP1, port1, (c3, c4))
select (IP2, port2, c3)
Initiator
Sender
e1
direction = ItoA
Acceptor
Receiver
e2
openWithoutCodecs (IP1, port1, noMedia)
openAckWithRealCodecs (IP2, port2, (c1, c2))
selectAndEstablishByInitiator (IP, port, c1)
describe (IP2, port2, (c3, c4))
select (IP1, port1, c3)

Figure 7.14: Modify Codec of a Media Channel by Descriptor Scenario

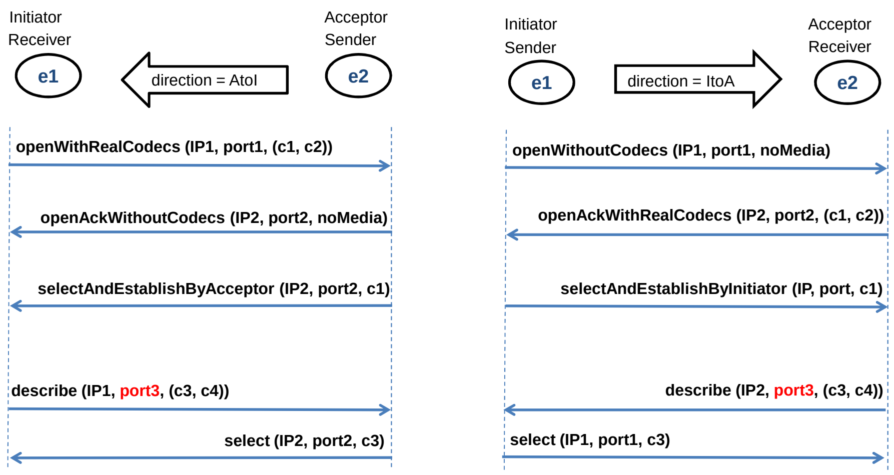

Initiator
Receiver
e1
direction = AtoI
Acceptor
Sender
e2
openWithRealCodecs (IP1, port1, (c1, c2))
openAckWithoutCodecs (IP2, port2, noMedia)
selectAndEstablishByAcceptor (IP2, port2, c1)
describe (IP1, port3, (c3, c4))
select (IP2, port2, c3)
Initiator
Sender
e1
direction = ItoA
Acceptor
Receiver
e2
openWithoutCodecs (IP1, port1, noMedia)
openAckWithRealCodecs (IP2, port2, (c1, c2))
selectAndEstablishByInitiator (IP, port, c1)
describe (IP2, port3, (c3, c4))
select (IP1, port1, c3)

Figure 7.15: Modify Port of a Media Channel by Descriptor Scenario

which has been received in the time of establishing the media channel, and start sending data by the new codec.

Second, considering Figure 7.14, the receiver endpoint, can send a new set of codecs in a describe signal. As described in Section 7.2.1, the other endpoint, has to respond to the descriptor by choosing a codec from the new set and sending it via a selector.

Finally, in Figure 7.15, it is shown that each endpoint can describe itself with a new port by sending a descriptor signal carrying the new port value.

Considering the three described modify scenarios, the *modify* event is decomposed in four atomicity diagrams, presented in Figure 7.16. In the first two scenarios in Figure 7.13 and Figure 7.14, one of the properties of the established channel is modified (set of codecs or selected codec). Whereas Figure 7.15 presents modifying the initiator port property in the left and modify the acceptor port property in the right. Therefore Figure 7.15 corresponds to two diagrams in Figure 7.16.

First diagram (a), is related to "modify codec of the media channel by selector" scenario in Figure 7.13. As described before, modifying codec can be done by initiator or acceptor of the media channel, both of them is done by *modifyBySelector* event in this level of refinement. More details are added in the 4th refinement level.

Diagram (b), is related to both types of "modify codec of the media channel by descriptor" scenario in Figure 7.14.

Diagram (c), contains decomposition related to "modify the initiator port" shown on the left hand side of Figure 7.15.

Finally diagram (d), shows the decomposition related to "modify the acceptor port" on the right hand side of Figure 7.15.

For instance, the Event-B model of part (b) and part (c) are presented in Figure 7.17 and Figure 7.18 respectively. Considering changing codec scenario in Figure 7.17, the refining event is a response, whereas in changing port scenario in Figure 7.18, the refining event is the *modify* event. As explained in Section 4.2.2, the refining event is the event which simulates the main behaviour of the abstract event. Here the event which changes one of the properties of the channel, is considered as the refining event. In Figure 7.17, the action of changing the codec is done in the *respond* event, whereas in Figure 7.18, the *modify* event changes one of the properties (initiator port) of the channel.

As described in Section 4.3.5, since the *modify* event in the first refinement is a loop event, in this level of refinement a loop resetting event is needed for each atomicity decomposition of the *modify* event. For instance, the resetting event for part (b) of Figure 7.16, is presented in Figure 7.19.

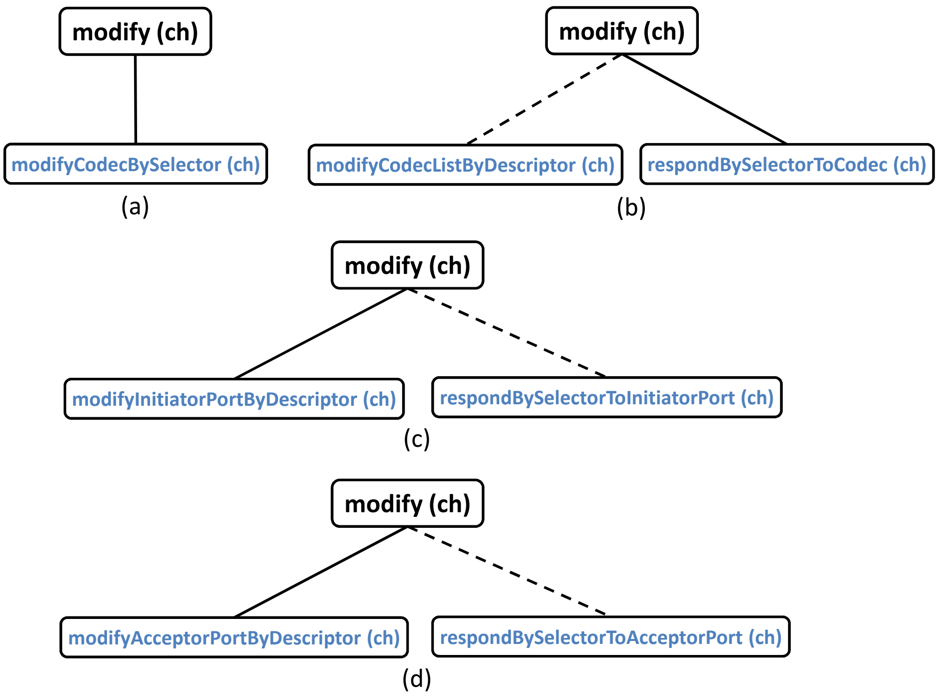
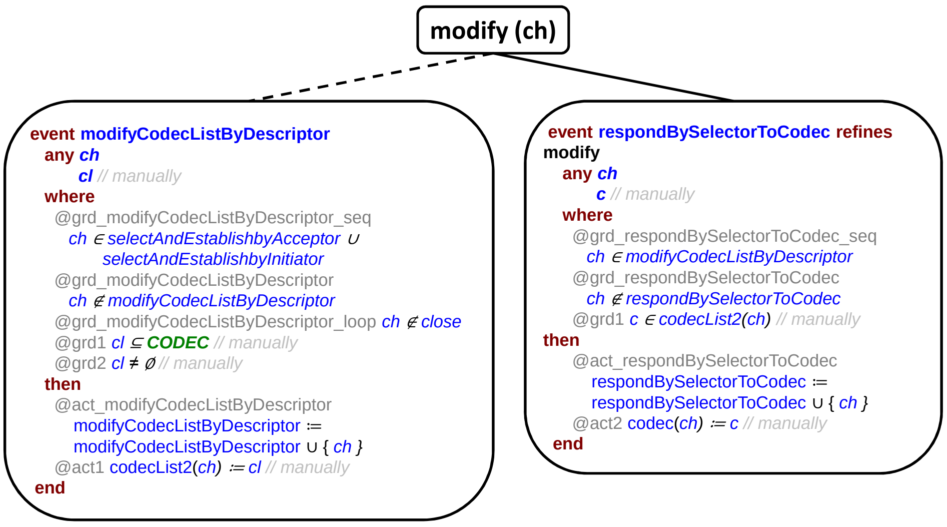

modify (ch)
modifyCodecBySelector (ch)
(a)
modify (ch)
modifyCodecListByDescriptor (ch)
respondBySelectorToCodec (ch)
(b)
modify (ch)
modifyInitiatorPortByDescriptor (ch)
respondBySelectorToInitiatorPort (ch)
(c)
modify (ch)
modifyAcceptorPortByDescriptor (ch)
respondBySelectorToAcceptorPort (ch)
(d)

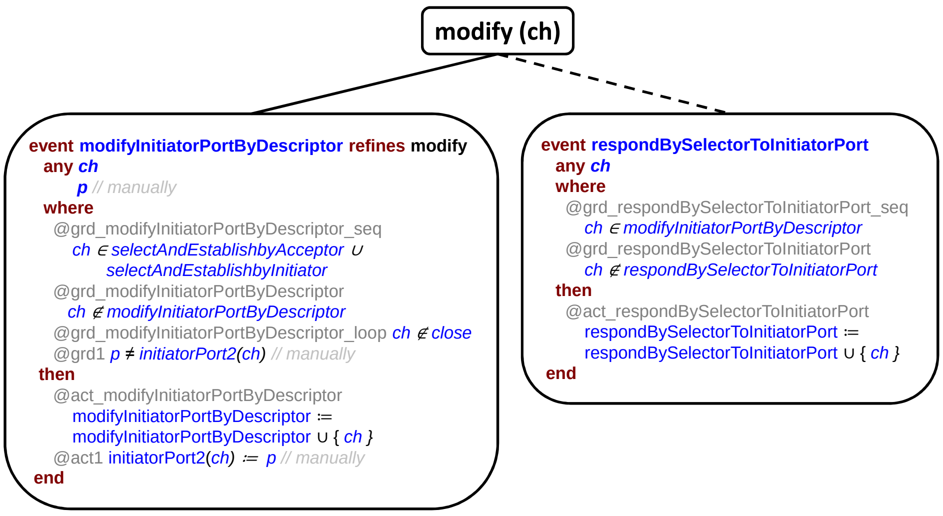
Figure 7.16: Breaking the Atomicity of Modify a Media Channel, M2



```
event modifyCodecListByDescriptor
  any ch
      cl // manually
  where
   @grd_modifyCodecListByDescriptor_seq
     ch ∈ selectAndEstablishbyAcceptor ∪
          selectAndEstablishbyInitiator
   @grd_modifyCodecListByDescriptor
     ch ∉ modifyCodecListByDescriptor
   @grd_modifyCodecListByDescriptor_loop ch ∉ close
   @grd1 cl ⊆ CODEC // manually
   @grd2 cl ≠ ∅ // manually
  then
   @act_modifyCodecListByDescriptor
      modifyCodecListByDescriptor ≔
      modifyCodecListByDescriptor ∪ { ch }
   @act1 codecList2(ch) ≔ cl // manually
end
```

```
event respondBySelectorToCodec refines
modify
   any ch
       c // manually
   where
    @grd_respondBySelectorToCodec_seq
      ch ∈ modifyCodecListByDescriptor
    @grd_respondBySelectorToCodec
      ch ∉ respondBySelectorToCodec
    @grd1 c ∈ codecList2(ch) // manually
then
    @act_respondBySelectorToCodec
       respondBySelectorToCodec ≔
       respondBySelectorToCodec ∪ { ch }
    @act2 codec(ch) ≔ c // manually
 end
```

Figure 7.17: Event-B Model, M2, Media Channel System



```
event modifyInitiatorPortByDescriptor refines modify
  any ch
      p // manually
  where
    @grd_modifyInitiatorPortByDescriptor_seq
      ch ∈ selectAndEstablishbyAcceptor ∪
           selectAndEstablishbyInitiator
    @grd_modifyInitiatorPortByDescriptor
      ch ∉ modifyInitiatorPortByDescriptor
    @grd_modifyInitiatorPortByDescriptor_loop ch ∉ close
    @grd1 p ≠ initiatorPort2(ch) // manually
  then
    @act_modifyInitiatorPortByDescriptor
      modifyInitiatorPortByDescriptor ≔
      modifyInitiatorPortByDescriptor ∪ { ch }
    @act1 initiatorPort2(ch) ≔ p // manually
  end
```

```
event respondBySelectorToInitiatorPort
  any ch
  where
    @grd_respondBySelectorToInitiatorPort_seq
      ch ∈ modifyInitiatorPortByDescriptor
    @grd_respondBySelectorToInitiatorPort
      ch ∉ respondBySelectorToInitiatorPort
  then
    @act_respondBySelectorToInitiatorPort
      respondBySelectorToInitiatorPort ≔
      respondBySelectorToInitiatorPort ∪ { ch }
  end
```

Figure 7.18: Event-B Model, M2, Media Channel System

```
event modify_Loop_Reset1
  any ch
  where
    @grd_reset ch ∈ respondBySelectorToCodec
  then
    @act_reset_modifyCodecListByDescriptor
      modifyCodecListByDescriptor ≔ modifyCodecListByDescriptor \ {ch}
    @act_reset_respondBySelectorToCodec
      respondBySelectorToCodec ≔ respondBySelectorToCodec \ {ch}
  end
```

Figure 7.19: Loop Resetting, M2, Media Channel System

Interactive proving: Failing proof obligations can lead to the identification of problems in the model that needed to be fixed. Discharging proof obligations in an interactive way, can lead us to make some changes in the model. In this level there are three EQL (Equality of a preserved variable) proof obligations which do not discharge without changing the model. One of them is explained in next paragraph, the other two are similar.

All variables in one Event-B machine can be changed only with the events of that machine in order to preserve consistency. *evt / v / EQL* is a proof obligation which ensures that abstract variable v is preserved in the concrete event *evt*. These kind of undischarged proof obligations occur because some abstract variables change in concrete events whose corresponding abstract events do not change the same variables. For example in this refinement level of the media channel development, the undischarged proof obligation

modifyInitiatorPortByDescriptor/initiatorPort/EQL

occurs because the concrete event *modifyInitiatorPortByDescriptor* changes the abstract variable *initiatorPort* whereas the corresponding abstract event, *modify*, did not change the same variable. One solution which Butler used in [44], is defining new variables in the refinement level which replace the abstract ones. Obviously some gluing invariants for linking them are necessary.

There are three new variables, *initiatorPort2*, *acceptorPort2* and *codecList2*, which replaced the abstract one, *initiatorPort*, *acceptorPort* and *codecList* respectively.

7.2.5 3rd Refinement: Breaking the Atomicity of Close Media Channel

This is a simple refinement in which the atomicity of the *close* event is broken into two sub-events, see Figure 7.20.

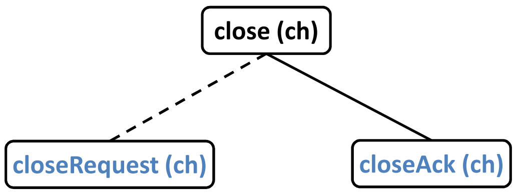


Figure 7.20: Breaking the Atomicity of Close a Media Channel, M3

7.2.6 4th Refinement: Second Level Breaking the Atomicity of Modify Media Channel

Up to the second refinement level, modifying a media channel was an atomic event which was done in a single step. In the second refinement level, the atomicity of the *modify* event has been decomposed, without considering which side of a channel, initiator or acceptor, is willing to send the modify signal and change the media channel's codec set. Considering initiator and acceptor endpoints, the fourth refinement level breaks the atomicity of modify events in a further level of decomposition. xor-constructor is used in breaking the atomicity of the *modifyBySelector* event, *modifyCodecByDescriptor* event and *respondBySelectortoCodec* event, illustrated in Figure 7.21.

Figure 7.22 illustrates the Event-B model of part (a) in Figure 7.21. The decision to use exclusive choice between sub-events is made based on the direction of the media channel. As presented in Figure 7.13, if the channel is from initiator to acceptor (ItoA), modelled in guard *grd2*, then the codec can be changed only by the initiator. Because the initiator has received the set of codecs from the acceptor, so the initiator can choose a new codec from the set. And if the channel is from acceptor to initiator (AtoI), modelled in guard *grd2*, then the codec can be changed only by the acceptor. Simply it can be said that only the sender of a media channel can choose a new codec from the received set of codecs, and the sender is the initiator when the direction is *ItoA* and is the acceptor when the direction is *AtoI*.

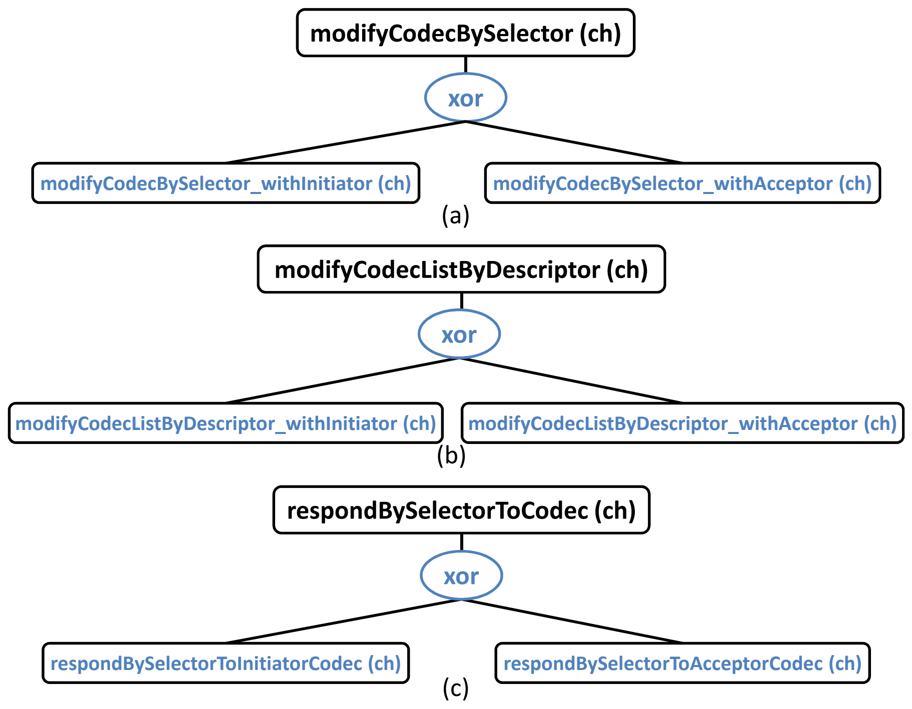
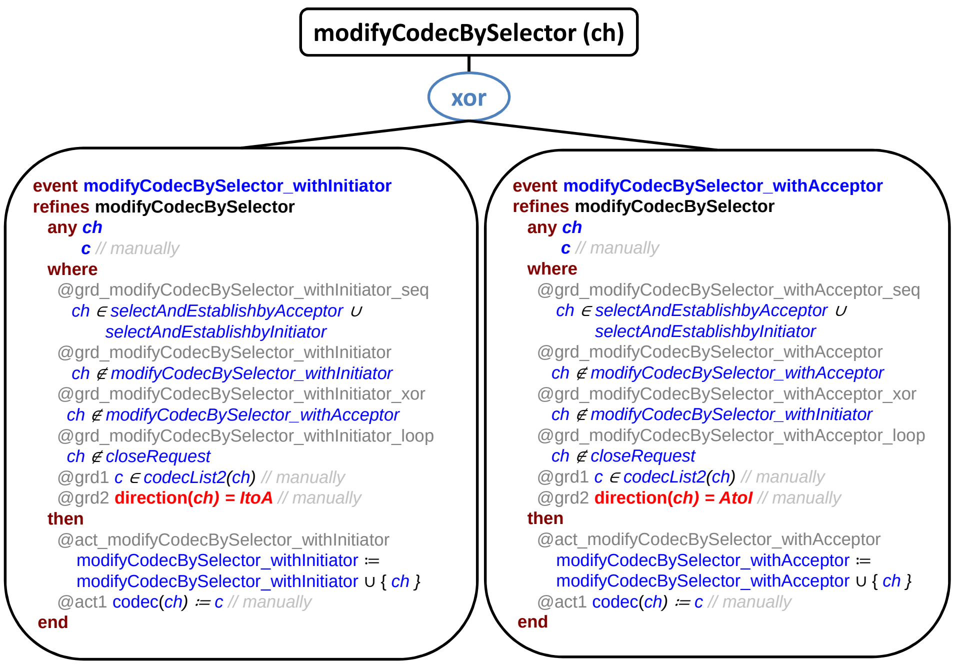

modifyCodecBySelector (ch)
xor
modifyCodecBySelector_withInitiator (ch)
modifyCodecBySelector_withAcceptor (ch)
(a)
modifyCodecListByDescriptor (ch)
xor
modifyCodecListByDescriptor_withInitiator (ch)
modifyCodecListByDescriptor_withAcceptor (ch)
(b)
respondBySelectorToCodec (ch)
xor
respondBySelectorToInitiatorCodec (ch)
respondBySelectorToAcceptorCodec (ch)
(c)

Figure 7.21: Further Breaking the Atomicity of Modify a Media Channel, M4



```
event modifyCodecBySelector_withInitiator
refines modifyCodecBySelector
  any ch
      c // manually
  where
    @grd_modifyCodecBySelector_withInitiator_seq
      ch ∈ selectAndEstablishbyAcceptor ∪
           selectAndEstablishbyInitiator
    @grd_modifyCodecBySelector_withInitiator
      ch ∉ modifyCodecBySelector_withInitiator
    @grd_modifyCodecBySelector_withInitiator_xor
      ch ∉ modifyCodecBySelector_withAcceptor
    @grd_modifyCodecBySelector_withInitiator_loop
      ch ∉ closeRequest
    @grd1 c ∈ codecList2(ch) // manually
    @grd2 direction(ch) = ItoA // manually
  then
    @act_modifyCodecBySelector_withInitiator
      modifyCodecBySelector_withInitiator ≔
      modifyCodecBySelector_withInitiator ∪ { ch }
    @act1 codec(ch) ≔ c // manually
end
```

```
event modifyCodecBySelector_withAcceptor
refines modifyCodecBySelector
  any ch
      c // manually
  where
    @grd_modifyCodecBySelector_withAcceptor_seq
      ch ∈ selectAndEstablishbyAcceptor ∪
           selectAndEstablishbyInitiator
    @grd_modifyCodecBySelector_withAcceptor
      ch ∉ modifyCodecBySelector_withAcceptor
    @grd_modifyCodecBySelector_withAcceptor_xor
      ch ∉ modifyCodecBySelector_withInitiator
    @grd_modifyCodecBySelector_withAcceptor_loop
      ch ∉ closeRequest
    @grd1 c ∈ codecList2(ch) // manually
    @grd2 direction(ch) = AtoI // manually
  then
    @act_modifyCodecBySelector_withAcceptor
      modifyCodecBySelector_withAcceptor ≔
      modifyCodecBySelector_withAcceptor ∪ { ch }
    @act1 codec(ch) ≔ c // manually
end
```

Figure 7.22: Event-B Model, M4, Media Channel System

7.2.7 5th Refinement: Second Level Breaking the Atomicity of Close Media Channel

Up to this refinement level closing a media channel is done with execution of the *closeRequest* event and *closeAck* event, without considering the direction of the channel. As presented in Figure 7.23, a *closeRequest* can be sent by either side of a channel, when the direction is either *AtoI* or *ItoA*.

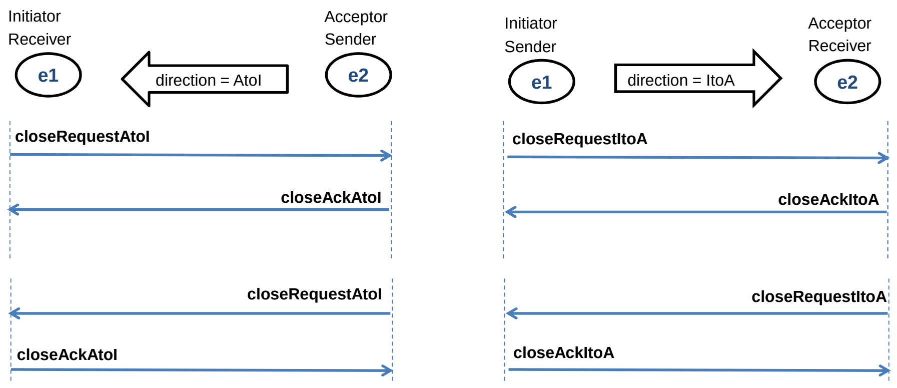


Figure 7.23: Close a Media Channel Scenarios

The final refinement level of the media channel system development contains further breaking of the atomicity of the *close* events. The atomicity decomposition diagrams of the 5th refinement level is illustrated in Figure 7.24. In the sub-events' guards the direction of the media channel is distinguished. As instance, the Event-B model of part (a) is presented in Figure 7.25.

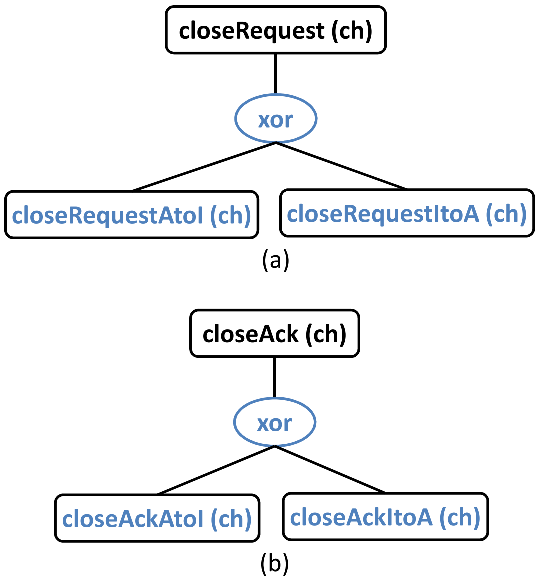
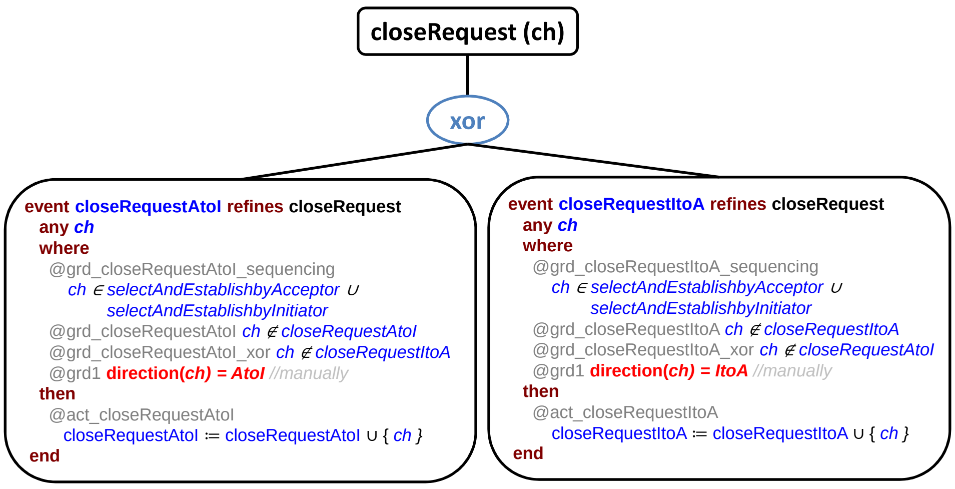


Figure 7.24: Further Breaking the Atomicity of Close a Media Channel, M5



```
event closeRequestAtoI refines closeRequest
  any ch
  where
    @grd_closeRequestAtoI_sequencing
      ch ∈ selectAndEstablishbyAcceptor ∪
           selectAndEstablishbyInitiator
    @grd_closeRequestAtoI ch ∉ closeRequestAtoI
    @grd_closeRequestAtoI_xor ch ∉ closeRequestItoA
    @grd1 direction(ch) = AtoI //manually
  then
    @act_closeRequestAtoI
      closeRequestAtoI ≔ closeRequestAtoI ∪ { ch }
  end
```

```
event closeRequestItoA refines closeRequest
  any ch
  where
    @grd_closeRequestItoA_sequencing
      ch ∈ selectAndEstablishbyAcceptor ∪
           selectAndEstablishbyInitiator
    @grd_closeRequestItoA ch ∉ closeRequestItoA
    @grd_closeRequestItoA_xor ch ∉ closeRequestAtoI
    @grd1 direction(ch) = ItoA //manually
  then
    @act_closeRequestItoA
      closeRequestItoA ≔ closeRequestItoA ∪ { ch }
  end
```

Figure 7.25: Event-B Model, M5, Media Channel System

7.2.8 Evaluation of Manual Event-B Model and Automatic Event-B Model

Use of the atomicity decomposition plug-in in creating the Event-B model of a system, ensures a consistent encoding of the atomicity decomposition diagrams in a systematic way. The manual version is less systematic and less consistent. Although the manual model and the automatic model, which is created with the plug-in, capture the same behaviours, there are some differences. Some of the differences of the automatic Event-B model, and the manual one in developing the media channel system, are described here. These differences can justify the higher level consistency of the Event-B model which is created with the plug-in.

7.2.8.1 Variable Naming Protocol

In the automatic Event-B model, following the patterns in Section 4.2 and translation rules in Section 5.4, each control variable has same name as the events' name. Whereas in the manual Event-B model, there is no specific naming protocol for variables' name. Providing a unique naming protocol helps to understand the model easier, and can help to track the ordering between events.

7.2.8.2 Different Approaches to Model Ordering In Event-B Model

As described in Section 4.4, there are different approaches to modelling ordering in Event-B. As justified in Section 4.4, we adopted the subset sets to model ordering. Therefore the automatic Event-B model of the media channel system, uses the subset sets. Whereas the manual Event-B model is a combination of subset sets and disjoint sets. In the manual Event-B model, in *close* event, the parameter *ch* is removed from set of established media channels. The Event-B model of *establishMediaChannel* event and *close* event in the abstract level of manual model are presented in Figure 7.26, which can be compared with the automatic Event-B model in Figure 7.6. The figure shows that the control variable name, *aMediaChannel*, is not same as the event name, *establishMediaChannel*, as explained in Section 7.2.8.1.

```
event establishMediaChannel
  any ch c
  where
   @grd1 ch ∉ aMediaChannel
   @grd2 c ∈ CODEC
  then
   @act1 aMediaChannel ≔ aMediaChannel ∪ { ch }
   @act2 codec(ch) ≔ c
 end
```

```
event close
 any ch
 where
  @grd1 ch ∈ aMediaChannel
 then
  @act1 aMediaChannel ≔ aMediaChannel \ {ch}
  @act2 codec ≔ {ch} ⩤ codec
end
```

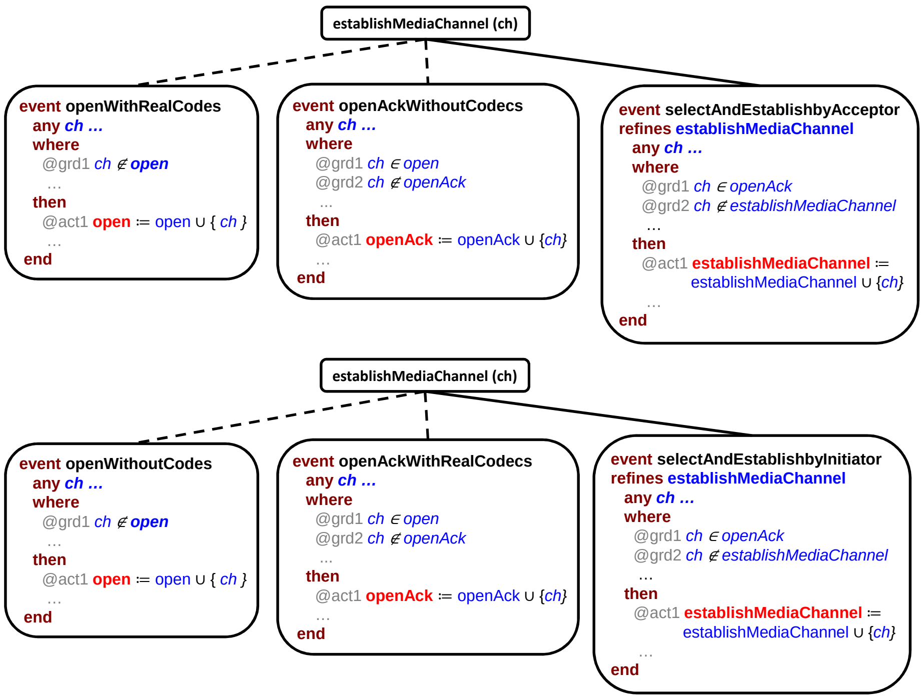
Figure 7.26: Manual Event-B Model, M0, Media Channel System

As a result, in the manual Event-B model, the relation between different states of a media channel, *establishMediaChannel* state and *close* state, can not be specified in the invariant. Whereas in the automatic Event-B model, invariant *inv_close_seq* presented in Figure 7.5, specifies the ordering between the *establishMediaChannel* event and the *close* event. This ensures that the abstract orderings are upheld in the refinement of the Event-B models more strongly than if specified only in the *close* event guard.

Having one more invariant in the automatic Event-B model, invariant *inv_close_seq* presented in Figure 7.5, slightly increases the number of proof obligations in the automatic Event-B model. The summary of proof obligations is reviewed in Section 7.2.9.

7.2.8.3 One More Refinement Level in the Manual Model

In the manual Event-B model, there was not a one-to-one relation between control variables and the events. For example considering the manual events in the first refinement level, presented in Figure 7.27, both *openWithRealCodecs* event and *openWithoutCodecs* event change the state of a channel, *ch*, to *open*. And both *openAckWithoutCodecs* event and *openAckWithrealCodecs* event change the state of a channel, *ch*, to *openAck*. And both *selectAndEstablishbyAcceptor* event and *selectAndEstablishbyInitiator* event change the state of a channel, *ch*, to *establishMediaChannel*.



```
event openWithRealCodes
  any ch ...
  where
   @grd1 ch ∉ open
   ...
  then
   @act1 open ≔ open ∪ { ch }
   ...
 end
```

```
event openAckWithoutCodecs
  any ch ...
  where
   @grd1 ch ∈ open
   @grd2 ch ∉ openAck
   ...
  then
   @act1 openAck ≔ openAck ∪ {ch}
   ...
 end
```

```
event selectAndEstablishbyAcceptor
refines establishMediaChannel
  any ch ...
  where
   @grd1 ch ∈ openAck
   @grd2 ch ∉ establishMediaChannel
   ...
  then
   @act1 establishMediaChannel ≔
          establishMediaChannel ∪ {ch}
   ...
end
```

```
event openWithoutCodes
  any ch ...
  where
   @grd1 ch ∉ open
   ...
  then
   @act1 open ≔ open ∪ { ch }
   ...
 end
```

```
event openAckWithRealCodecs
  any ch ...
  where
   @grd1 ch ∈ open
   @grd2 ch ∉ openAck
   ...
  then
   @act1 openAck ≔ openAck ∪ {ch}
   ...
 end
```

```
event selectAndEstablishbyInitiator
refines establishMediaChannel
  any ch ...
  where
   @grd1 ch ∈ openAck
   @grd2 ch ∉ establishMediaChannel
   ...
  then
   @act1 establishMediaChannel ≔
          establishMediaChannel ∪ {ch}
   ...
end
```

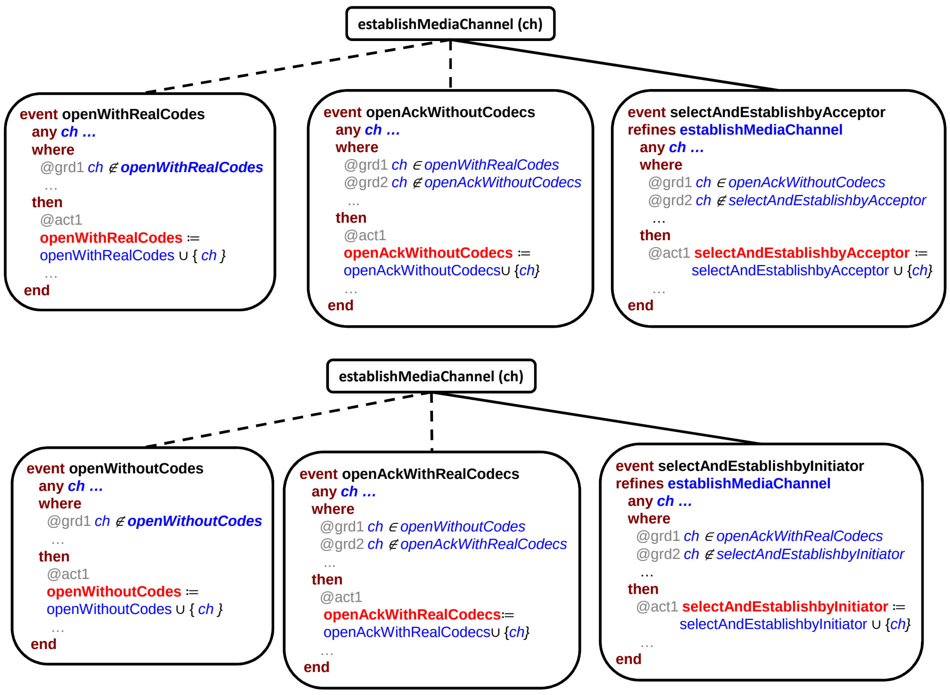
Figure 7.27: Manual Event-B Model, M1, Media Channel System

Whereas in the automatic Event-B model, as presented in Figure 7.28, there is a one-to-one relation between control variables and the events, and each event change the state of a media channel to a unique state with same name as the event.

In the manual model there is a further refinement level in order to introduced an unique state for each event, for instance, concrete variables: *openAckWithoutCodecs* and *openAckWithrealCodecs* which replace the single abstract variable *open*. The further refinement level makes the manual model larger and more complex, comparing to the automatic model. Also more effort is need to define the gluing invariants between abstract variables and concrete variables. The gluing invariants makes the proving more complex, it will explained later in Section 7.2.9.

7.2.8.4 Weak Guard versus Strong Guard

The combined atomicity decomposition of the *modify* event, including machine *M2* and machine *M4* refinements, is presented in Figure 7.29. In the automatic Event-B model, as the result of the xor-constructor, event *respondBySelectorToInitiatorCodec* is guarded with *modifyCodecListByDescriptor_withInitiator* variable and *modifyCodecListByDescriptor_withAcceptor* variable. The sequencing guard in *respondBySelectorToInitiatorCodec* event is as follows:



```
event openWithRealCodes
  any ch ...
  where
   @grd1 ch ∉ openWithRealCodes
   ...
  then
   @act1
   openWithRealCodes ≔
   openWithRealCodes ∪ { ch }
   ...
 end
```

```
event openAckWithoutCodecs
  any ch ...
  where
   @grd1 ch ∈ openWithRealCodes
   @grd2 ch ∉ openAckWithoutCodecs
   ...
  then
   @act1
   openAckWithoutCodecs ≔
   openAckWithoutCodecs∪ {ch}
   ...
 end
```

```
event selectAndEstablishbyAcceptor
refines establishMediaChannel
  any ch ...
  where
   @grd1 ch ∈ openAckWithoutCodecs
   @grd2 ch ∉ selectAndEstablishbyAcceptor
   ...
  then
   @act1 selectAndEstablishbyAcceptor ≔
         selectAndEstablishbyAcceptor ∪ {ch}
   ...
end
```

```
event openWithoutCodes
  any ch ...
  where
   @grd1 ch ∉ openWithoutCodes
   ...
  then
   @act1
   openWithoutCodes ≔
   openWithoutCodes ∪ { ch }
   ...
 end
```

```
event openAckWithRealCodecs
  any ch ...
  where
   @grd1 ch ∈ openWithoutCodes
   @grd2 ch ∉ openAckWithRealCodecs
   ...
  then
   @act1
   openAckWithRealCodecs≔
   openAckWithRealCodecs∪ {ch}
   ...
 end
```

```
event selectAndEstablishbyInitiator
refines establishMediaChannel
  any ch ...
  where
   @grd1 ch ∈ openAckWithRealCodecs
   @grd2 ch ∉ selectAndEstablishbyInitiator
   ...
  then
   @act1 selectAndEstablishbyInitiator ≔
         selectAndEstablishbyInitiator ∪ {ch}
   ...
end
```

Figure 7.28: Automatic Event-B Model, M1, Media Channel System

$@grd\_respondBySelectorToInitiatorCodec\_seq$
$ch \in modifyCodecListByDescriptor\_withInitiator \cup$
$modifyCodecListByDescriptor\_withAcceptor$

However based on the requirements, the *respondBySelectorToInitiatorCodec* event executes to respond only to *modifyCodecListByDescriptor_withInitiator* event. So the stated guard is too weak to model the requirement. The requirement is satisfied with another guard in both *modifyCodecListByDescriptor_withInitiator* event and *respondBySelector-ToInitiatorCodec* event:

$@grd2 \quad direction(ch) = AtoI$

Therefore if $direction(ch) = ItoA$, then the guards of *respondBySelectorToInitiator-Codec* event does not hold and the event can not execute.

In the manual Event-B model, the sequencing guard in *respondBySelectorToInitiator-Codec* event is enough strong to satisfy the requirement:

$@grd1 \quad ch \in modifyCodecListByDescriptor\_withInitiator$

Although *respondBySelectorToInitiatorCodec* event in the manual model is still guarded with the direction guard:

$@grd2 \quad direction(ch) = AtoI$

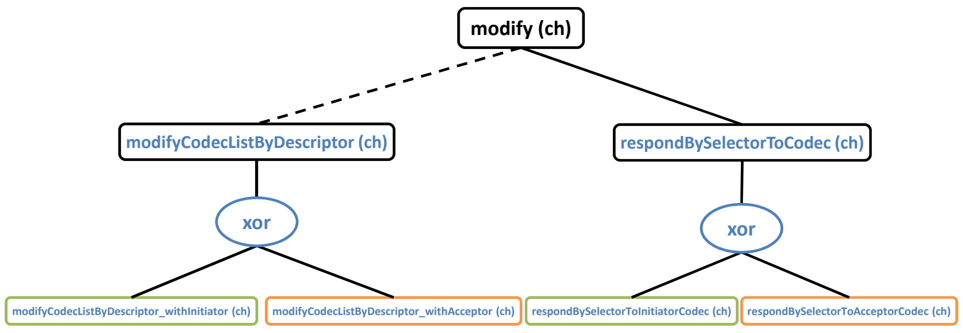

modify (ch)
modifyCodecListByDescriptor (ch)
respondBySelectorToCodec (ch)
xor
xor
modifyCodecListByDescriptor_withInitiator (ch)
modifyCodecListByDescriptor_withAcceptor (ch)
respondBySelectorToInitiatorCodec (ch)
respondBySelectorToAcceptorCodec (ch)

Figure 7.29: Combined Atomicity Decomposition Diagram of *modify* Event, Media Channel System

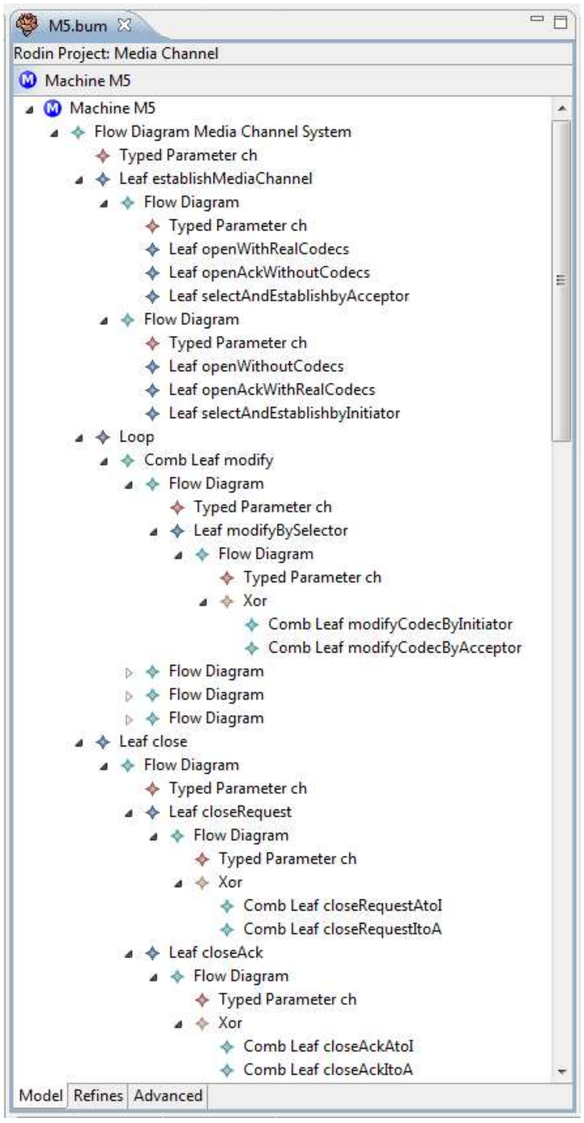
7.2.8.5 Tool Application: Atomicity Decomposition Model of the Media Channel System

The atomicity Decomposition model of the final refinement of the media channel system, generated with the atomicity decomposition plug-in is presented in Figure 7.30.

7.2.9 Overview of Proof Obligations

The result of the proof effort in the Rodin for the automatic Event-B model, is outlined in Figure 7.31. The *Total* column shows the total number of proof obligations generated for each level. The *Auto* column represents the number of those proof obligations that proved automatically by the prover and the *Manual* column shows the number of proof obligations which proved interactively. In the automatic model, almost all proof obligations are proved automatically.

Figure 7.32 presents the proof effort for the manual Event-B model. The total number of proofs are predominantly more then the total number of proofs in the automatic model, since the extra refinement level in the manual model, *Machine6*, as explained in Section 7.2.8.3, significantly increase the number of proof obligations. A large number of proof obligations are caused because of gluing invariants, that are needed to define the relations between the abstract non-unique states and concrete unique states. Also there are six proof obligations in *Machine6* which proved interactively. The interactive proofs are the gluing invariant preservation proofs. Therefore, as explained in Section 7.2.8.3, introducing the unique states in an extra refinement level, not only makes the model large and complex, but also it makes the proof more complex.



M5.bum

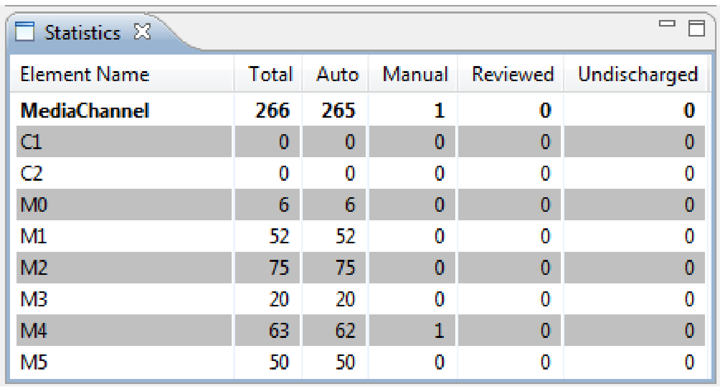
Rodin Project: Media Channel

Machine M5

```
Machine M5
  Flow Diagram Media Channel System
    Typed Parameter ch
    Leaf establishMediaChannel
      Flow Diagram
        Typed Parameter ch
        Leaf openWithRealCodecs
        Leaf openAckWithoutCodecs
        Leaf selectAndEstablishbyAcceptor
      Flow Diagram
        Typed Parameter ch
        Leaf openWithoutCodecs
        Leaf openAckWithRealCodecs
        Leaf selectAndEstablishbyInitiator
    Loop
      Comb Leaf modify
        Flow Diagram
          Typed Parameter ch
          Leaf modifyBySelector
            Flow Diagram
              Typed Parameter ch
              Xor
                Comb Leaf modifyCodecByInitiator
                Comb Leaf modifyCodecByAcceptor
        Flow Diagram
        Flow Diagram
        Flow Diagram
    Leaf close
      Flow Diagram
        Typed Parameter ch
        Leaf closeRequest
          Flow Diagram
            Typed Parameter ch
            Xor
              Comb Leaf closeRequestAtoI
              Comb Leaf closeRequestItoA
        Leaf closeAck
          Flow Diagram
            Typed Parameter ch
            Xor
              Comb Leaf closeAckAtoI
              Comb Leaf closeAckItoA
```

Model | Refines | Advanced

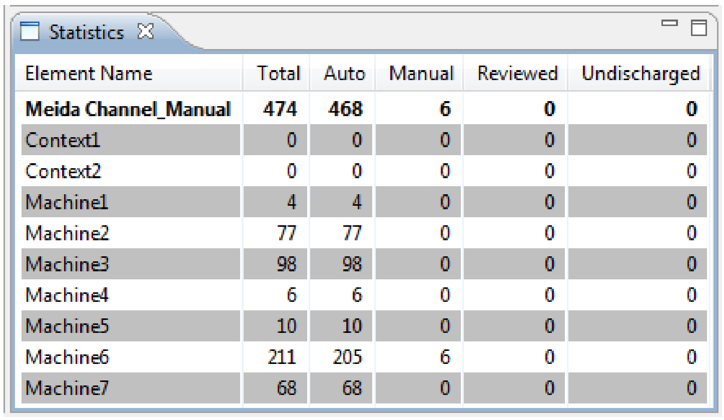
Figure 7.30: Atomicity Decomposition Model of the Media Channel System



Statistics

Element Name	Total	Auto	Manual	Reviewed	Undischarged
MediaChannel	**266**	**265**	**1**	**0**	**0**
C1	0	0	0	0	0
C2	0	0	0	0	0
M0	6	6	0	0	0
M1	52	52	0	0	0
M2	75	75	0	0	0
M3	20	20	0	0	0
M4	63	62	1	0	0
M5	50	50	0	0	0

Figure 7.31: Proof Obligation Statistics for the Automatic Media Channel Event-B Model



Statistics

Element Name	Total	Auto	Manual	Reviewed	Undischarged
Meida Channel_Manual	**474**	**468**	**6**	**0**	**0**
Context1	0	0	0	0	0
Context2	0	0	0	0	0
Machine1	4	4	0	0	0
Machine2	77	77	0	0	0
Machine3	98	98	0	0	0
Machine4	6	6	0	0	0
Machine5	10	10	0	0	0
Machine6	211	205	6	0	0
Machine7	68	68	0	0	0

Figure 7.32: Proof Obligation Statistics for the Manual Media Channel Event-B Model

7.3 BepiColombo Space Craft System (Published in NFM 2011 Symposium)

7.3.1 Overview of the BepiColombo System

BepiColombo mission [77] is one of the case studies of the DEPLOY project [78]. The overview of the BepiColombo space craft system in this section is based on the information of the Space System Finland Ltd [79, 80].

Exploration of the planet Mercury is the main goal of the BepiColombo mission. Two orbiters are sent by BepiColombo. One of these is the Mercury Planetry Orbiters (MPO). It carries remote sensing and radioscience instrumenation. The MIXS/SIXS data Processing Unit (DPU) is the important part of this orbiter. One of the responsibilities of MIXS/SIXS DPU is handling Telecommand (TC) and Telemetry (TM) communication.

There are two instruments which are controlled by MIXS/SIXS DPU: Solar Intensity X-ray and Spectrometer (SIXS) and Mercury Imaging X-ray Spectrometer (MIXS). Each instrument contains two sensors: SIXS-X (X-ray spectrometer), SIXS-P (Particle spectrometer), and MIXS-T (Telescope), MIXS-C (Collimator).

The MIXS/SIXS On-Board Software (OBSW) running on the DPUs' CPU consists of five different software components: the Core Software (CSW), SIXS-P ASW (Application Software), SIXS-X ASW, MIXS-T ASW and MIXS-C ASW. The high-level architecture of BepiColombo SIXS/MIXS OBSW is presented in Figure 7.33.

In our development as an abstract view all application softwares are seen as a single component called devices, presented in a single box in Figure 7.33. Developing the mode management particulary for each application software is a subject that requires further work.

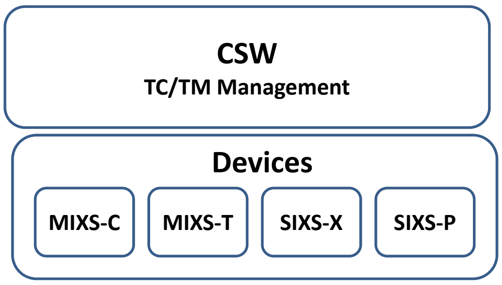


Figure 7.33: High-level Architecture of BepiColombo SIXS/MIXS OBSW

Here is the summary of the system requirements in a simple scenario:

- A *TC* (Telecommand) is received in the core from the earth.
- The Core Software (CSW) checks the syntax of the received *TC*.
- Further semantics checking have to be done for the validated *TC*. If the *TC* contains a message for one of the devices, it will send it to the device for semantics checking, otherwise the semantics checking in done in the core.
- For each valid *TC*, a control *TM* (Telemetry) is generated and sent to the earth.
- For some particular types of *TC*, some data *TMs* are generated and sent back to the earth.

As illustrated in Figure 7.33, the Core Software (CSW) plays a management role over the devices. CSW is responsible for communication with the earth on one hand and with the devices on the other hand. It plays a role of an interface between the earth and the devices.

7.3.2 Modelling Architecture

Figure 7.34 presents the development architecture of Event-B model of the BepiColombo system. *M0* is the abstract model of the system. After the abstraction there are three levels of refinement. In these models, *M1*, *M2* and *M3*, those events are refined which are not purely allocated to core side or device side of the system. In other words, in these models, refining an event results in a collection of sub-events which are a combination of core actions, device actions and shared actions between core and device. This concept will be more explained in Section 7.3.7. After three levels of refinement the model is decomposed to two sub-models, called *core* sub-model and *device* sub-model. Finally, the core sub-model is refined in two more refinement levels, called *M4*, *M5*. The atomicity decomposition approach is applied to the refinement levels both before and after model decomposition.

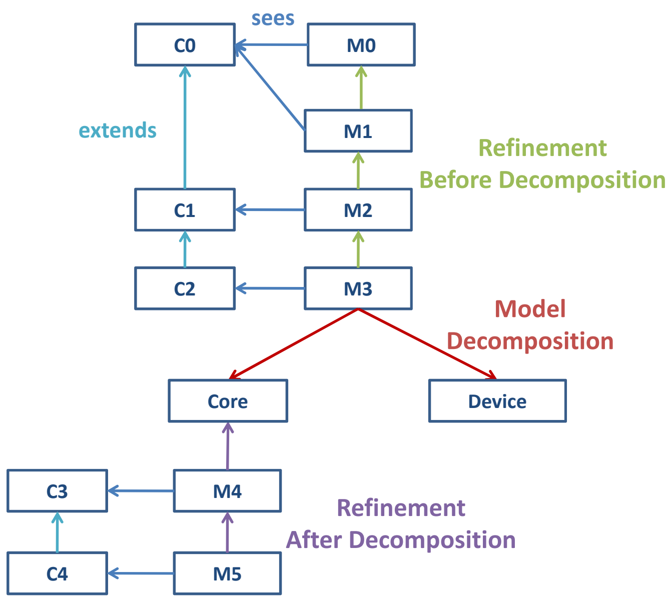


Figure 7.34: Development Architecture of the BepiColombo Event-B Model

In the abstraction the main goals of the system are modelled and the details of the protocol are added through refinement levels. The atomicity decomposition diagrams present explicit relationships between events of refinement levels. Table 7.1 summarizes new details which are added to each level of refinement.

- Machine *M0* models goals of the BepiColombo system. Three main phases are modelled. Receiving a *TC*, Validating the received *TC*, and if it is needed generating one or more *TM(s)*.
- In machine *M1* the validation phase is refined and further details of the validation protocol are added.
- Machine *M2* distinguishes between validation of core *TCs* and device *TCs*.
- In machine *M3* the protocol of sending a device *TC* to the device for validation and sending back the validation result is modelled.
- Machine *M4* models processing *TMs* in the core.
- Machine *M5* models producing and sending *TMs* in the core.

7.3.3 Abstract Specification

7.3.3.1 Static Part of the Specification

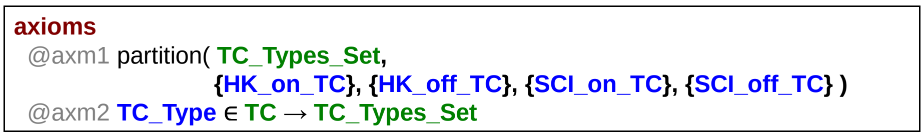
The abstract context, *C0*, which models the static part of the abstract model contains two sets. *TC* is the set of existing telecommands which would be received in the core, and *TC_Types_Set* shows types of a *TC*. There are four *TC*'s types :

Machine	Summary of the Model
M0	Receiving, validating a *TC* and generating *TMs*.
M1	Refining validation phase.
M2	Distinguishing difference between validating core *TCs* and device *TCs*.
M3	Refining validation phase of a device *TC*.
M4	Refining processing *TMs* in the core.
M5	Producing and sending *TMs* in the core.

Table 7.1: Summary of Event-B Refinements, BepiColombo System

- *HK_on_TC* (Housekeeping On TC)
- *HK_off_TC* (Housekeeping Off TC)
- *SCI_on_TC* (Science On TC)
- *SCI_off_TC* (Science Off TC)

A part of the abstract context, *C0*, is displayed in Figure 7.35. *TC_Type* is a total function from *TC* set to *TC_Types_Set* set. A total function guarantees that each *TC* has exactly one type.



```
axioms
  @axm1 partition( TC_Types_Set,
                          {HK_on_TC}, {HK_off_TC}, {SCI_on_TC}, {SCI_off_TC} )
  @axm2 TC_Type ∈ TC → TC_Types_Set
```

Figure 7.35: Context C0, BepiColombo System

For an *off TC* (SCI_off_TC, HK_off_TC), only a control Telemetry (*TM*) is produced, whereas for an *on TC* (SCI_on_TC, HK_on_TC) one or more data *TMs* are produced as well. This requirement is specified by a guard in the event of generating data *TMs*. It is shown later.

7.3.3.2 Events and Dynamic Part of the Specification

In the abstract model, the main goals of the system are modelled. The most abstract events are illustrated in Figure 7.36, using the diagram explained in Section 4.3.1. Three different scenarios are possible:

- **(a)** As it is presented in part (a) of Figure 7.36, first a *TC* is received by execution of *ReceiveTC* event, then it is validated by execution of *TC_Validation_Ok* event. In this case the *TC*'s type is *HK_off_TC* or *Sci_off_TC*, so there is no need to generate data *TMs* in response. Producing a control *TM* is later done by refining the *TC_Validation_Ok* event.

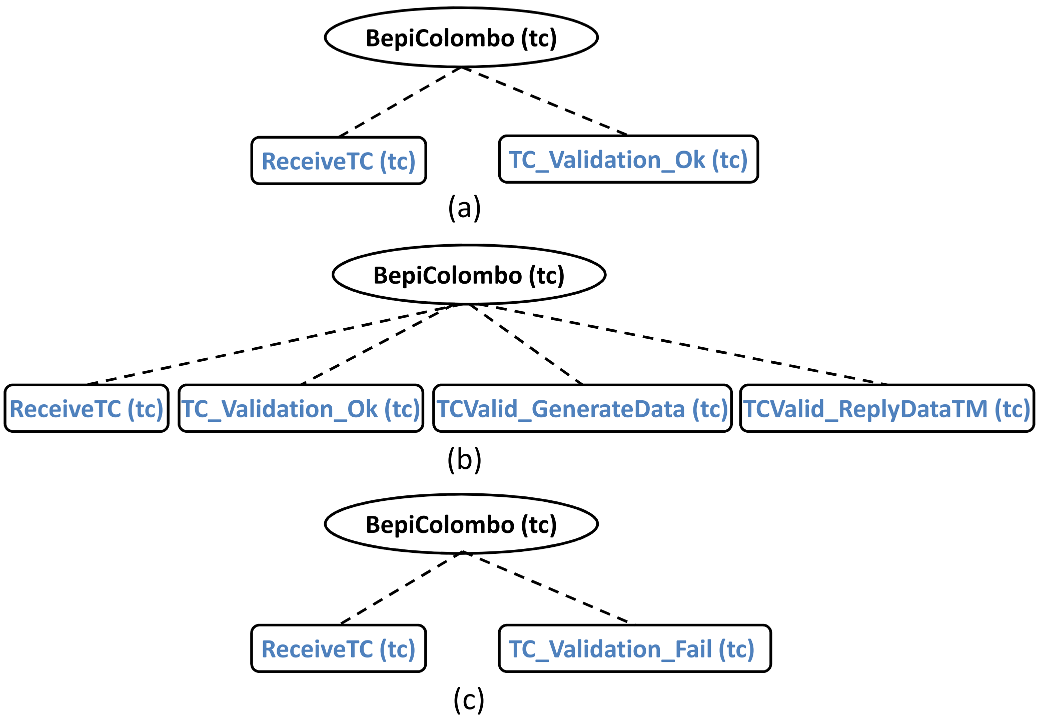


Figure 7.36: The Atomicity Decomposition Diagrams, M0, BepiColombo System

- **(b)** Another case is illustrated in part (b) of Figure 7.36. After receiving and validating a *TC* with type *HK_on_TC* or *Sci_on_TC*, some bunches of data are generated by execution of *TC_GenerateData* event in one of the devices, and finally by execution of *TCValid_ReplyDataTM* event in the core, one or more data *TM(s)* are produced and sent back to the earth.

- **(c)** Part (c) of Figure 7.36 shows the case when the received *TC*'s validation is failed. This is modelled by the *TC_Validation_Fail* event.

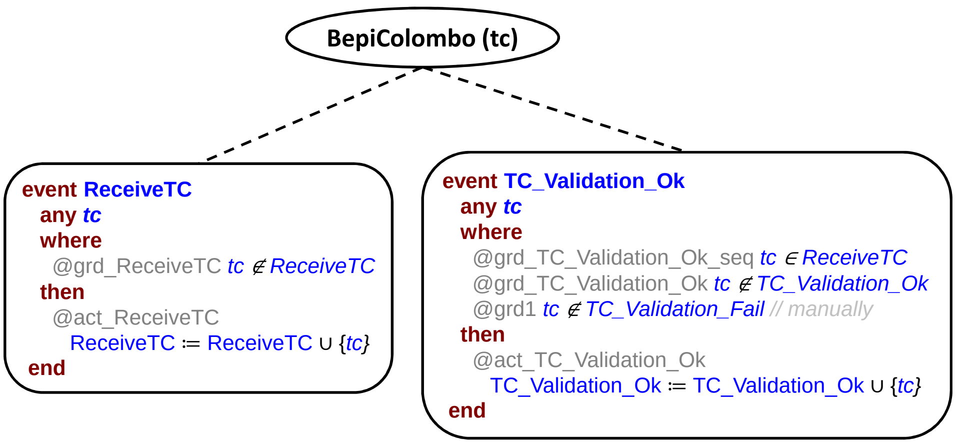
As explained in Section 4.2, the ordering between events is modelled using some control variables, invariants and guards. In machine *M0* there are five control variables. Figure 7.37 presents the control variables and invariants of *M0*. For each event there is a variable with the same name as the event, and if one event is executed after another one, the later variable is a subset of the former one. For example, *TC_Validation_Ok* event can be executed only after execution of the *ReceiveTC* event, so invariant *inv_TC_Validation_Ok_seq* describes *TC_Validation_Ok* variable as a subset of the *ReceiveTC* variable. Only invariant *inv1* is modelled manually. The other invariants and variables are generated by the tool. Invariant *inv1* describes that *TC_Validation_Ok* and *TC_Validation_Fail* are disjoint.

There are some guards for controlling the sequencing of events. As you can see in Figure 7.38 in the first event, *ReceiveTC*, one *TC* is added to the *ReceiveTC* set variable, and then *TC_Validation_Ok* event can executed only for a *TC* which belongs to

```
variables ReceiveTC
          TC_Validation_Ok
          TCValid_GenerateData
          TCValid_ReplyDataTM
          TC_Validation_Fail

invariants
 @inv_ReceiveTC                    ReceiveTC ⊆ TC
 @inv_TC_Validation_Ok_seq         TC_Validation_Ok ⊆ ReceiveTC
 @inv_TCValid_GenerateData_seq TCValid_GenerateData ⊆ TC_Validation_Ok
 @inv_TCValid_ReplyDataTM_seq TCValid_ReplyDataTM ⊆ TCValid_GenerateData
 @inv_TC_Validation_Fail_seq      TC_Validation_Fail  ⊆ ReceiveTC
 @inv1                            TC_Validation_Ok ∩ TC_Validation_Fail = ∅ // manually
```

Figure 7.37: Variables and Invariants, M0, BepiColombo System

the *ReceiveTC* variable, which is checked in *grd_TC_Validation_Ok_seq* guard in the *TC_Validation_Ok* event. Figure 7.38 is the Event-B model of the part (a) in Figure 7.36.



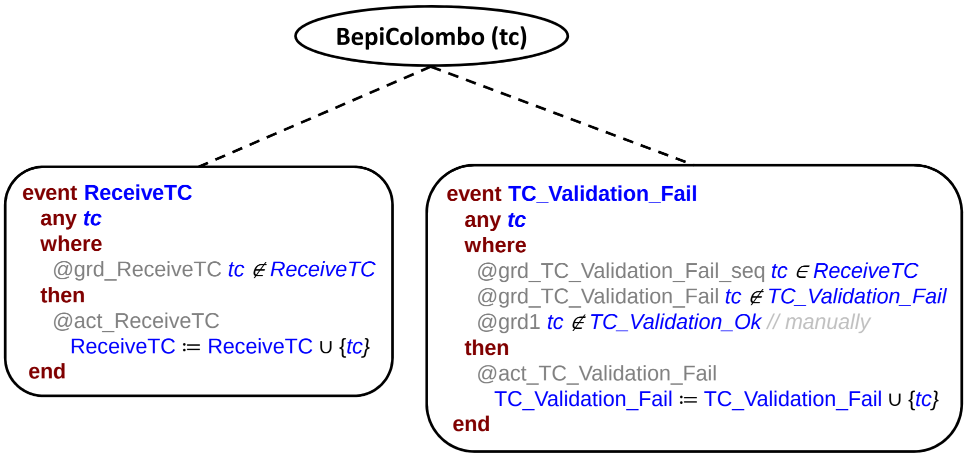
```
event ReceiveTC
  any tc
  where
   @grd_ReceiveTC tc ∉ ReceiveTC
  then
   @act_ReceiveTC
     ReceiveTC ≔ ReceiveTC ∪ {tc}
 end
```

```
event TC_Validation_Ok
  any tc
  where
   @grd_TC_Validation_Ok_seq tc ∈ ReceiveTC
   @grd_TC_Validation_Ok tc ∉ TC_Validation_Ok
   @grd1 tc ∉ TC_Validation_Fail // manually
  then
   @act_TC_Validation_Ok
     TC_Validation_Ok ≔ TC_Validation_Ok ∪ {tc}
 end
```

Figure 7.38: Event-B Model, M0, BepiColombo System

This sequence is repeated in Figure 7.39 for the *TC_Validation_Fail* event. It is the Event-B model of part (c) in Figure 7.36. The difference between the *TC_Validation_Ok* event and the *TC_Validation_Fail* event is that the *TC* is added to different variables in each event. In each of the *TC_Validation_Ok* event and the *TC_Validation_Fail* event, there is one guard, *grd1*, added manually. These guards ensure the invariant *inv1*.

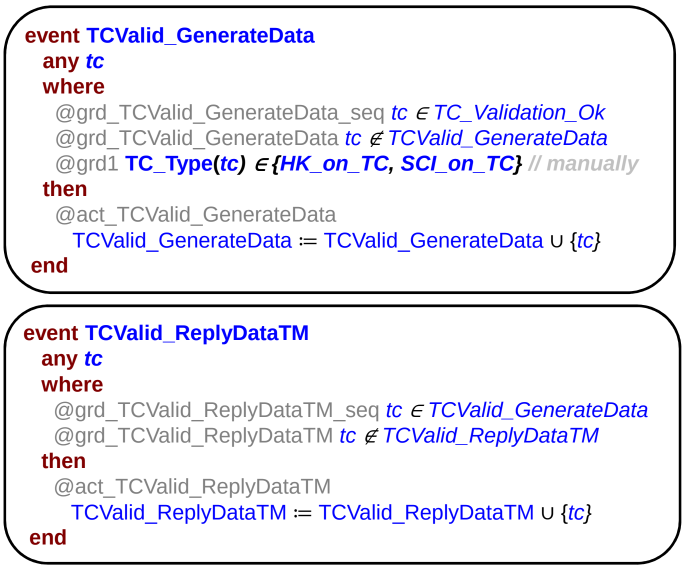
If a received *TC* is added to the *TC_Validation_Ok* variable and it's an *on TC*, the sequence can continue by execution of the *TCValid_GenerateData* event and then the *TCValid_ReplyDataTM* event. The Event-B model is illustrated in Figure 7.40. Guard *grd1* of the *TCValid_GenerateData* event checks the type of *TC*. If its type is either *SCI_on_TC* or *HK_on_TC* then the event can be executed.



```
event ReceiveTC
  any tc
  where
   @grd_ReceiveTC tc ∉ ReceiveTC
  then
   @act_ReceiveTC
     ReceiveTC ≔ ReceiveTC ∪ {tc}
 end
```

```
event TC_Validation_Fail
  any tc
  where
   @grd_TC_Validation_Fail_seq tc ∈ ReceiveTC
   @grd_TC_Validation_Fail tc ∉ TC_Validation_Fail
   @grd1 tc ∉ TC_Validation_Ok // manually
  then
   @act_TC_Validation_Fail
     TC_Validation_Fail ≔ TC_Validation_Fail ∪ {tc}
 end
```

Figure 7.39: Event-B Model, M0, BepiColombo System



```
event TCValid_GenerateData
  any tc
  where
   @grd_TCValid_GenerateData_seq tc ∈ TC_Validation_Ok
   @grd_TCValid_GenerateData tc ∉ TCValid_GenerateData
   @grd1 TC_Type(tc) ∈ {HK_on_TC, SCI_on_TC} // manually
  then
   @act_TCValid_GenerateData
     TCValid_GenerateData ≔ TCValid_GenerateData ∪ {tc}
 end
```

```
event TCValid_ReplyDataTM
  any tc
  where
   @grd_TCValid_ReplyDataTM_seq tc ∈ TCValid_GenerateData
   @grd_TCValid_ReplyDataTM tc ∉ TCValid_ReplyDataTM
  then
   @act_TCValid_ReplyDataTM
     TCValid_ReplyDataTM ≔ TCValid_ReplyDataTM ∪ {tc}
 end
```

Figure 7.40: Event-B Model, M0, BepiColombo System

7.3.4 1st Refinement: Refining the Validation Phase

In the abstract model, the validation phase is done by execution of one of two single atomic events, *TC_Validation_Ok* and *TC_Validation_Fail*. However validating a received *TC* is not atomic. It is done in two steps, checking the syntax and the semantics of a received *TC*. After syntax and semantics checks, in the third step a control *TM* is produced and sent back to the earth.

These details are modelled in the first refinement level, machine *M1*. *TC_Validation_Ok* and *TC_Validation_Fail* are decomposed to some sub-events which show further details of the validation phase. The atomicity decomposition diagrams are shown in Figure 7.41. Checking the syntax of a received *TC* is done by execution of *TCCheck_Ok* and *TCCheck_Fail* events and semantics checking is done by *TCExecute_Ok* and *TCExecute_Fail* events. *TCExecOk_ReplyCtrlTM*, *TCExecFail_ReplyCtrlTM* and

TCCheckFail_ReplyCtrlTM are events for generating control *TMs*. Considering Figure 7.41, part(a) illustrates the case when both syntax and semantic checks are ok; part(b) presented the case when syntax check is ok but semantic check is failed, and part(c) shows the case when syntax is failed.

As explained in Section 4.2.2, the refining event is the event which simulates the main behaviour of the abstract event. The behaviour of *TC_Validation_Ok* event is exhibited in the refinement level by a valid syntax check followed by a valid semantics check, therefore *TCExecute_Ok* event is the refining event in part(a). And The behaviour of *TC_Validation_Fail* event is exhibited in the refinement level either when syntax check is valid and semantics check is failed, part(b), or syntax check is failed, part(c), therefore *TCExecute_Fail* event and *TCCheck_Fail* are the refining events.

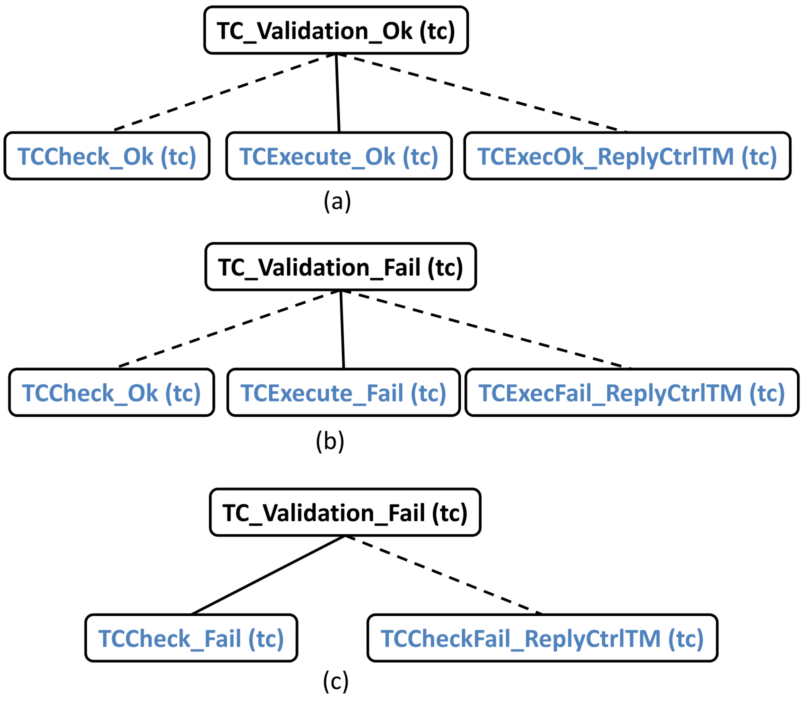
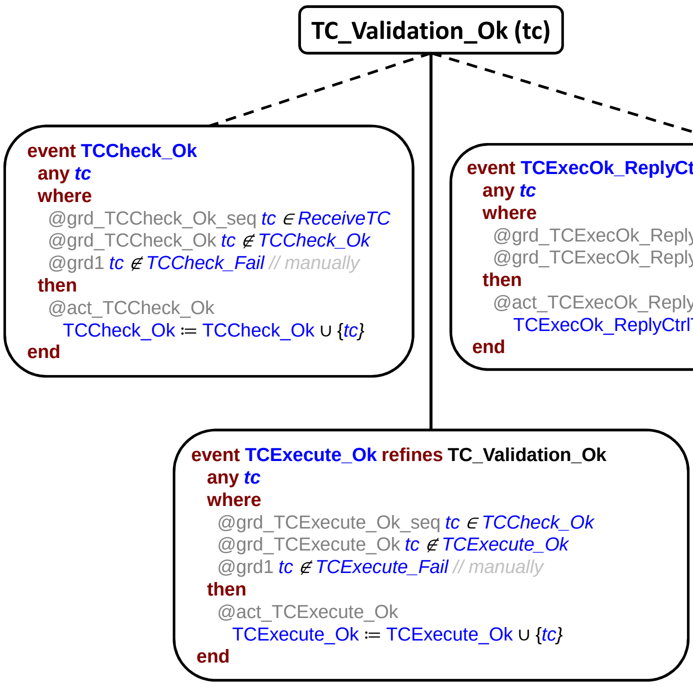


Figure 7.41: The Atomicity Decomposition Diagrams, M1, BepiColombo System

Considering a successful validation, the Event-B model is presented in Figure 7.42. *TCCheck_OK*, *TCExecute_Ok* and *TCExecOk_ReplyCtrlTM* are control variables. Clearly the sub-events with dashed lines, *TCCheck_Ok* and *TCExecOk_ReplyCtrlTM*, are new events which refine *skip* and the event with solid line, *TCExecute_Ok*, refines the abstract event, *TC_Validation_Ok*.

There are two gluing invariants in machine *M1* which define the relations between abstract variables and concrete variables. These invariants are shown in Figure 7.43. *inv_TCExecute_Ok_gluing* shows that concrete variable of *TCExecute_Ok* is equal to the abstract variable *TC_Validation_Ok*, since the *TCExecute_Ok* event refines *TC_Validation_Ok* event. Since two events refine the abstract event *TC_Validation_Fail*,



```
event TCCheck_Ok
  any tc
  where
    @grd_TCCheck_Ok_seq tc ∈ ReceiveTC
    @grd_TCCheck_Ok tc ∉ TCCheck_Ok
    @grd1 tc ∉ TCCheck_Fail // manually
  then
    @act_TCCheck_Ok
      TCCheck_Ok ≔ TCCheck_Ok ∪ {tc}
end
```

```
event TCExecOk_ReplyCtrlTM
  any tc
  where
    @grd_TCExecOk_ReplyCtrlTM_seq tc ∈ TCExecute_Ok
    @grd_TCExecOk_ReplyCtrlTM tc ∉ TCExecOk_ReplyCtrlTM
  then
    @act_TCExecOk_ReplyCtrlTM
      TCExecOk_ReplyCtrlTM ≔ TCExecOk_ReplyCtrlTM ∪ {tc}
end
```

```
event TCExecute_Ok refines TC_Validation_Ok
  any tc
  where
    @grd_TCExecute_Ok_seq tc ∈ TCCheck_Ok
    @grd_TCExecute_Ok tc ∉ TCExecute_Ok
    @grd1 tc ∉ TCExecute_Fail // manually
  then
    @act_TCExecute_Ok
      TCExecute_Ok ≔ TCExecute_Ok ∪ {tc}
end
```

Figure 7.42: Event-B Model, M1, BepiColombo System

the union of the corresponding variables is equal to the abstract variable, shown in *inv_gluing* invariant.

```
@inv_TCExecute_Ok_gluing TCExecute_Ok = TC_Validation_Ok
@inv_gluing TCExecute_Fail ∪ TCCheck_Fail = TC_Validation_Fail
```

Figure 7.43: Gluing Invariants, M1, BepiColombo System

7.3.5 2nd Refinement: Refining the Semantic Check

As it is presented in Figure 7.34, the next level of refinement, machine *M2*, sees context *C1*. There is a new field defined for *TC* called *PID* and it is a total function which shows the type of *TC*. A *TC* belongs to the core (*csw*) , or one of four devices, (*mixsc, mixst, sixsp, sixsx*). The properties are presented in Figure 7.44.

```
axioms
  @axm1 partition( PIDS, {csw}, {mixsc}, {mixst}, {sixsp}, {sixsx} )
  @axm2 PID ∈ TC → PIDS
```

Figure 7.44: Context C1, BepiColombo System

Up to this level of modelling, semantics checking of a received *TC* is done regardless of considering the type of *TC*. If a received *TC* belongs to the core, its semantics should be checked in the core. Otherwise it should be sent to a proper device and validating its semantics is done in the device. These details of the semantics checking are applied in the second refinement level, machine *M2*. The atomicity decomposition diagrams are illustrated in Figure 7.45. *TCExecute_Ok* event and *TCExecute_Fail* event are split into two sub-events identifying the type of a received *TC*. It's helpful to recall that syntax checking is always done in the core before semantics checking, and a received *TC* needs to be semantics checked only when its syntax check is ok.

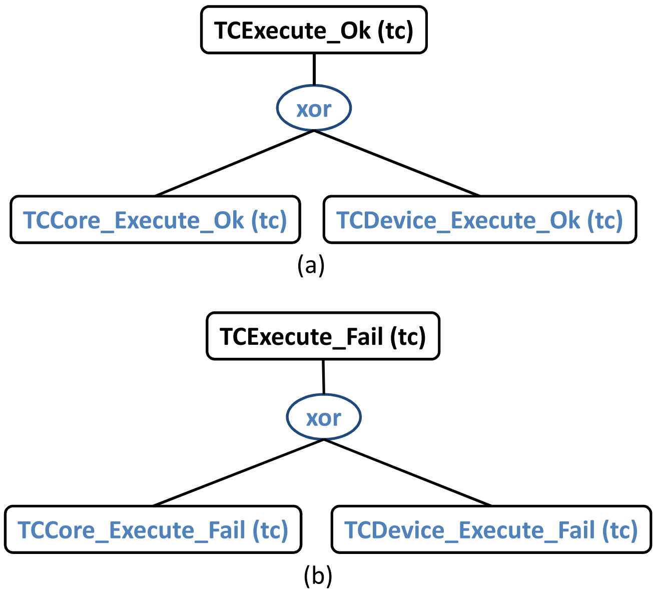
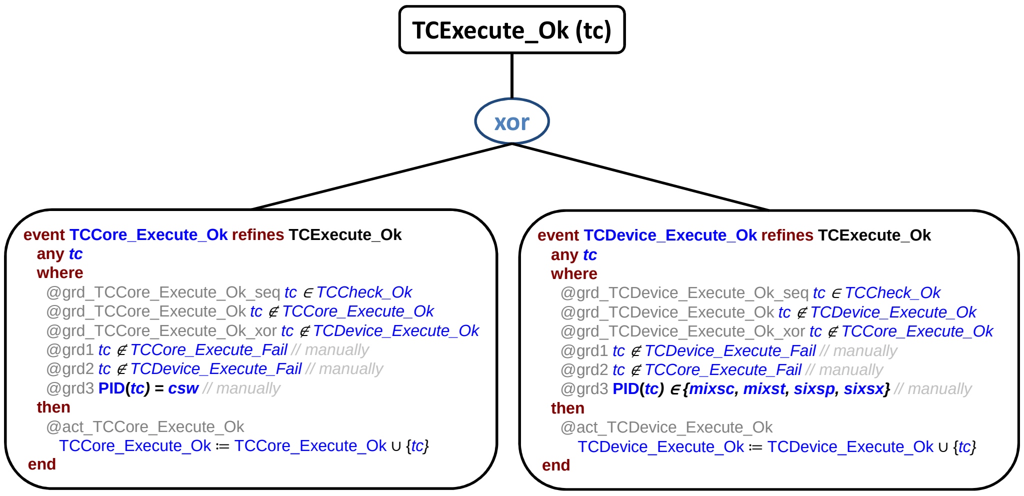


Figure 7.45: The Atomicity Decomposition Diagram, M2, BepiColombo System



```
event TCCore_Execute_Ok refines TCExecute_Ok
  any tc
  where
   @grd_TCCore_Execute_Ok_seq tc ∈ TCCheck_Ok
   @grd_TCCore_Execute_Ok tc ∉ TCCore_Execute_Ok
   @grd_TCCore_Execute_Ok_xor tc ∉ TCDevice_Execute_Ok
   @grd1 tc ∉ TCCore_Execute_Fail // manually
   @grd2 tc ∉ TCDevice_Execute_Fail // manually
   @grd3 PID(tc) = csw // manually
  then
   @act_TCCore_Execute_Ok
     TCCore_Execute_Ok ≔ TCCore_Execute_Ok ∪ {tc}
 end
```

```
event TCDevice_Execute_Ok refines TCExecute_Ok
  any tc
  where
   @grd_TCDevice_Execute_Ok_seq tc ∈ TCCheck_Ok
   @grd_TCDevice_Execute_Ok tc ∉ TCDevice_Execute_Ok
   @grd_TCDevice_Execute_Ok_xor tc ∉ TCCore_Execute_Ok
   @grd1 tc ∉ TCDevice_Execute_Fail // manually
   @grd2 tc ∉ TCCore_Execute_Fail // manually
   @grd3 PID(tc) ∈ {mixsc, mixst, sixsp, sixsx} // manually
  then
   @act_TCDevice_Execute_Ok
     TCDevice_Execute_Ok ≔ TCDevice_Execute_Ok ∪ {tc}
 end
```

Figure 7.46: Event-B Model, M2, BepiColombo System

In the Event-B model, as shown in Figure 7.46, both sub-events refine the abstract event the only difference is the guard, *grd3*, which checks the type of *TC*.

7.3.6 3rd Level of Refinement

In the third refinement level, machine *M3*, the atomicity of three events are decomposed, see Figure 7.47. For checking the semantics of a received *TC* which belongs

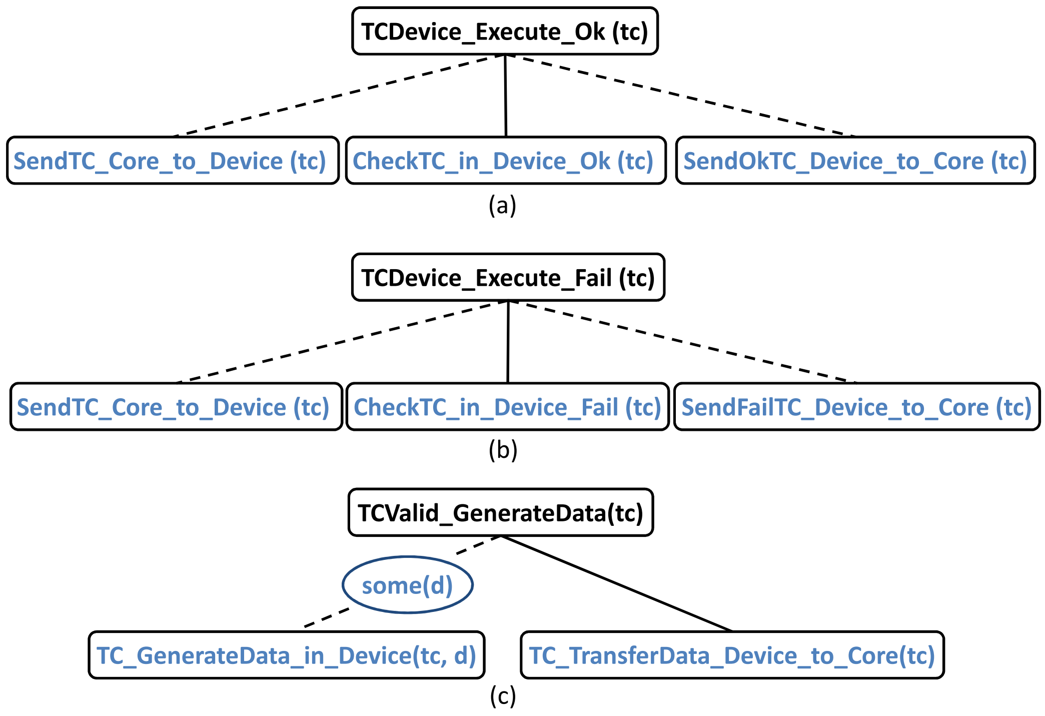

TCDevice_Execute_Ok (tc)
SendTC_Core_to_Device (tc)
CheckTC_in_Device_Ok (tc)
SendOkTC_Device_to_Core (tc)
(a)
TCDevice_Execute_Fail (tc)
SendTC_Core_to_Device (tc)
CheckTC_in_Device_Fail (tc)
SendFailTC_Device_to_Core (tc)
(b)
TCValid_GenerateData(tc)
some(d)
TC_GenerateData_in_Device(tc, d)
TC_TransferData_Device_to_Core(tc)
(c)

Figure 7.47: The Atomicity Decomposition Diagrams, M3, BepiColombo System

to one of the devices, the *TC* is sent to the proper device, *SendTC_Core_to_Device* event, the device checks the semantics of the *TC* , *CheckTC_in_Device_Ok* event and *CheckTC_in_Device_Fail* event, and finally the device sends back the result of semantics checking to the core, *SendOkTC_Device_to_Core* event and *SendFailTC_Device_to_Core* event. Part (a) shows a successful semantics checking, and part (b) shows when the *TC* is failed in semantics checking.

The *TCValid_GenerateData* event is decomposed to two sub-events, part (c) of Figure 7.47. As described before for an *on TC*, some data *TMs* are generated. Up to this level the generation is done in one atomic event. In machine *M3* the abstract event is broken to two sub-events. The data is generated in the device by execution of *TC_GenerateData_in_Device* event and then it is transferred to the core by execution of *TC_TransferData_Device_to_Core* event. Later details of producing data *TMs* from the transferred data in the core are added to the model, in the *M4* and *M5* machines.

The control stream and gluing invariants in Event-B model are same as the ones in the Event-B models described before. As described in Section 4.2.2, the refining event is the event which simulates the main behaviour of the abstract event. Considering part (a) and (b) of Figure 7.47, *CheckTC_in_Device_Ok* event and *CheckTC_in_Device_Fail* event exhibit the behaviours of *TCDevice_Execute_Ok* abstract event and *TCDevice_Execute_Fail* abstract event respectively, and the other sub-events model the data transformation from core to device or device to core. Considering part (c), as explained in Section 4.2.8, some-replicator has to be only with a dashed line. Therefore in part (c) the refining event (event with a solid line) is the other event.

7.3.7 Decomposing BepiColombo Model to Core and Device Sub-models

7.3.7.1 Combining Atomicity Decomposition and Model Decomposition in Event-B

So far we have decomposed the atomicity of those events which are not purely belonging to the core or the device part of the system. Refining purely core events, such as events which are related to generating data *TMs* and control *TMs*, are postponed after model decomposition for simplicity, since after decomposition of the model to some sub-models, the sub-models are smaller than the main model. Refinement has continued until reaching the state that all events are purely core events or device events or shared events between core and device. For instance, in the first refinement level, *M1*, the *TC_Validation_Ok* event has been decomposed to some sub-events, because validating a *TC* is an action which is composed of checking syntax of a received *TC* which should be done in core, checking the semantics of that *TC* which is a device action if the *TC* belongs to device. After three levels of refinement in the BepiColombo development process, all events can be allocated to core or device.

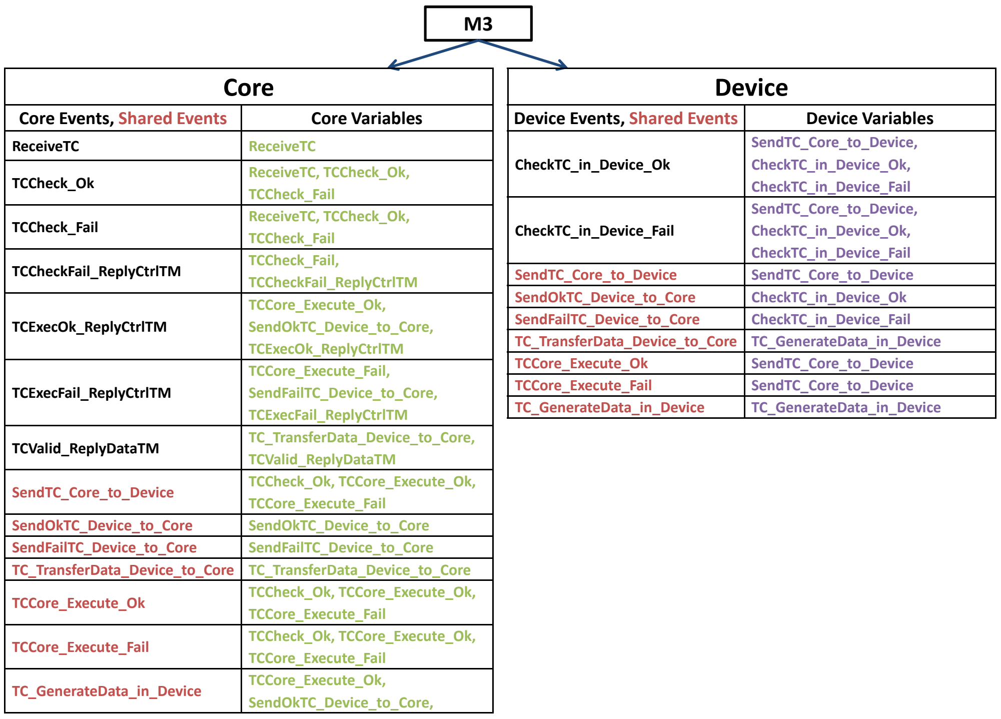
In this level the model is decomposed to two separate sub-models (Core and Device), as shown in Figure 7.34.

7.3.7.2 Shared Event Model Decomposition

We use the shared-event style decomposition, as described in Section 2.5.2.2, for decomposing the system to the core and device sub-models. The variables of *M3* are partitioned among the core and device sub-models, see Figure 7.48. Events using variables allocated to one sub-models are allocated to that sub-model. There are seven events using some variables allocated to the core and some variables allocated to the devices. These events, called shared events, are split.

Figure 7.49 shows shared events. Each of the shared events uses some core variables, which is at left hand side of Figure 7.49 and one device variables, at right of the figure. For simplicity just one of the core variables is shown in the figure. For instance, as shown in Figure 7.50, the *SendTC_Core_to_Device* event uses some core variables, i.e., *TCCheck_Ok*, *TCCore_Execute_Ok* and *TCCore_Execute_Fail*, and a device variable *SendTC_Core_to_Device*.

Thus far the model contains sixteen events and sixteen variables. After decomposition the events and variables are divided to sub-models, so each sub-model becomes simpler



Core

Core Events, Shared Events	Core Variables
ReceiveTC	ReceiveTC
TCCheck_Ok	ReceiveTC, TCCheck_Ok, TCCheck_Fail
TCCheck_Fail	ReceiveTC, TCCheck_Ok, TCCheck_Fail
TCCheckFail_ReplyCtrlTM	TCCheck_Fail, TCCheckFail_ReplyCtrlTM
TCExecOk_ReplyCtrlTM	TCCore_Execute_Ok, SendOkTC_Device_to_Core, TCExecOk_ReplyCtrlTM
TCExecFail_ReplyCtrlTM	TCCore_Execute_Fail, SendFailTC_Device_to_Core, TCExecFail_ReplyCtrlTM
TCValid_ReplyDataTM	TC_TransferData_Device_to_Core, TCValid_ReplyDataTM
SendTC_Core_to_Device	TCCheck_Ok, TCCore_Execute_Ok, TCCore_Execute_Fail
SendOkTC_Device_to_Core	SendOkTC_Device_to_Core
SendFailTC_Device_to_Core	SendFailTC_Device_to_Core
TC_TransferData_Device_to_Core	TC_TransferData_Device_to_Core
TCCore_Execute_Ok	TCCheck_Ok, TCCore_Execute_Ok, TCCore_Execute_Fail
TCCore_Execute_Fail	TCCheck_Ok, TCCore_Execute_Ok, TCCore_Execute_Fail
TC_GenerateData_in_Device	TCCore_Execute_Ok, SendOkTC_Device_to_Core,

Device

Device Events, Shared Events	Device Variables
CheckTC_in_Device_Ok	SendTC_Core_to_Device, CheckTC_in_Device_Ok, CheckTC_in_Device_Fail
CheckTC_in_Device_Fail	SendTC_Core_to_Device, CheckTC_in_Device_Ok, CheckTC_in_Device_Fail
SendTC_Core_to_Device	SendTC_Core_to_Device
SendOkTC_Device_to_Core	CheckTC_in_Device_Ok
SendFailTC_Device_to_Core	CheckTC_in_Device_Fail
TC_TransferData_Device_to_Core	TC_GenerateData_in_Device
TCCore_Execute_Ok	SendTC_Core_to_Device
TCCore_Execute_Fail	SendTC_Core_to_Device
TC_GenerateData_in_Device	TC_GenerateData_in_Device

Figure 7.48: Model Decomposition, Shared Events Style, BepiColombo System

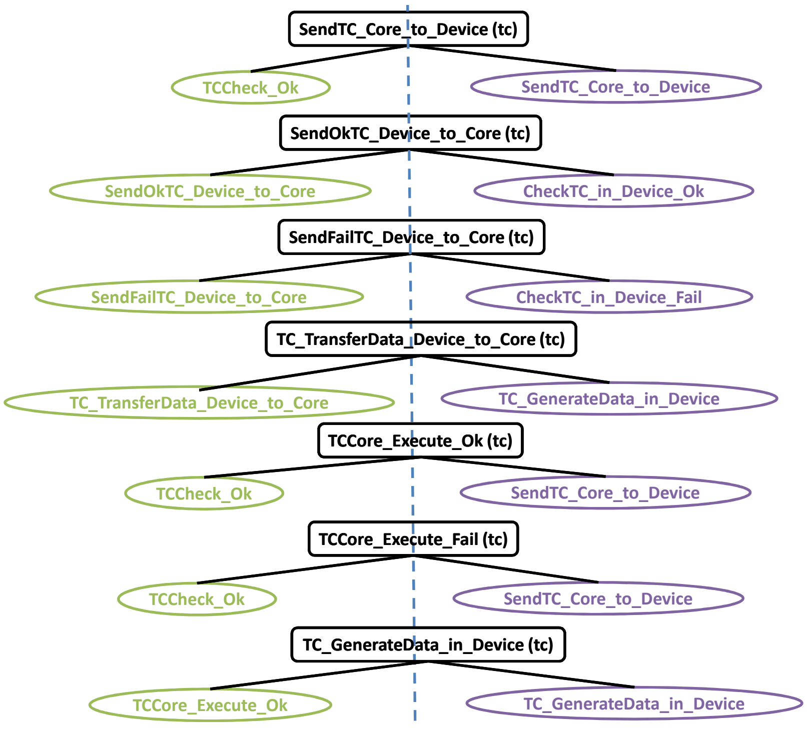
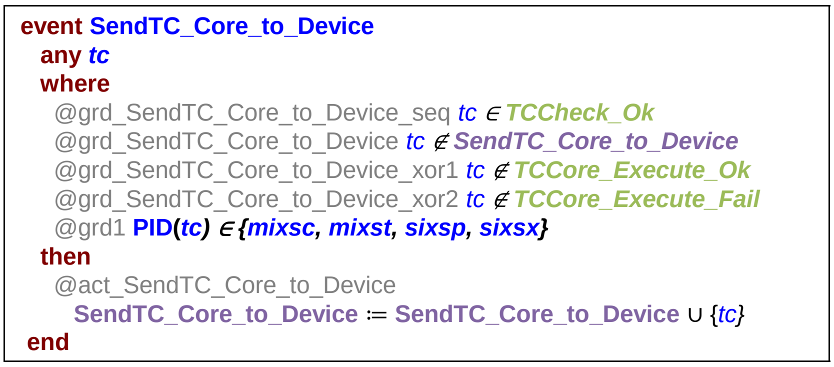

SendTC_Core_to_Device (tc)
TCCheck_Ok
SendTC_Core_to_Device
SendOkTC_Device_to_Core (tc)
SendOkTC_Device_to_Core
CheckTC_in_Device_Ok
SendFailTC_Device_to_Core (tc)
SendFailTC_Device_to_Core
CheckTC_in_Device_Fail
TC_TransferData_Device_to_Core (tc)
TC_TransferData_Device_to_Core
TC_GenerateData_in_Device
TCCore_Execute_Ok (tc)
TCCheck_Ok
SendTC_Core_to_Device
TCCore_Execute_Fail (tc)
TCCheck_Ok
SendTC_Core_to_Device
TC_GenerateData_in_Device (tc)
TCCore_Execute_Ok
TC_GenerateData_in_Device

Figure 7.49: Shared Events, BepiColombo System



```
event SendTC_Core_to_Device
  any tc
  where
   @grd_SendTC_Core_to_Device_seq tc ∈ TCCheck_Ok
   @grd_SendTC_Core_to_Device tc ∉ SendTC_Core_to_Device
   @grd_SendTC_Core_to_Device_xor1 tc ∉ TCCore_Execute_Ok
   @grd_SendTC_Core_to_Device_xor2 tc ∉ TCCore_Execute_Fail
   @grd1 PID(tc) ∈ {mixsc, mixst, sixsp, sixsx}
  then
   @act_SendTC_Core_to_Device
     SendTC_Core_to_Device ≔ SendTC_Core_to_Device ∪ {tc}
 end
```

Figure 7.50: An Instance of a Shared Event Before Model Decomposition, Bepi-Colombo System

and easier to manage. The division is illustrated in Figure 7.48. Shared events appear in both sub-models. The last six events in each sub-model are shared events.

The shared event's guards and actions are divided to two separate events, each in different sub-models. The division is done based on using sub-models variables. Each shared event in each sub-model only contains the guards and actions which use its own sub-model variables.

For example Figure 7.51 presents the *SendTC_Core_to_Device* shared events after model decomposition. Considering Figure 7.51 comparing to Figure 7.50, *SendTC_Core_to_Device* event in the core sub-model contains *grd_SendTC_Core_to_Device_seq*, *grd_SendTC_Core_to_Device_xor1* and *grd_SendTC_Core_to_Device_xor2* which use core variables, and in the device sub-model, it contains *grd_SendTC_Core_to_Device* and *act_SendTC_Core_to_Device* that use the device variable.

Since no core variable is modified in the action of the *SendTC_Core_to_Device* event, the core on its own does not know that it has send a *TC*. Therefore the *SendTC_Core_to_Device* event in the core sub-model would be enabled more often; the action which disables the *SendTC_Core_to_Device* event is in the device sub-model. One solution to disable the core shared event, is providing a preparation in the atomicity decomposition approach before applying model decomposition. As a preparation, one action can be added to the *SendTC_Core_to_Device* event which disables one core variable. As a result the action would be placed in the core sub-model when model decomposed. This case can be considered as future work in combining atomicity decomposition approach and model decomposition approach.

The decomposition was performed using the decomposition plug-in [21, 56]. The typing guard in each event (Figure 7.51) are added by the decomposition plug-in.

```
Core Sub-model
event SendTC_Core_to_Device
  any tc
  where
   @typing_tc tc ∈ TC
   @grd_SendTC_Core_to_Device_seq tc ∈ TCCheck_Ok
   @grd_SendTC_Core_to_Device_xor1 tc ∉ TCCore_Execute_Ok
   @grd_SendTC_Core_to_Device_xor2 tc ∉ TCCore_Execute_Fail
   @grd1 PID(tc) ∈ {mixsc, mixst, sixsp, sixsx}
 end
```

```
Device Sub-model
event SendTC_Core_to_Device
  any tc
  where
   @typing_tc tc ∈ TC
   @grd_SendTC_Core_to_Device tc ∉ SendTC_Core_to_Device
   @grd1 PID(tc) ∈ {mixsc, mixst, sixsp, sixsx}
  then
   @act_SendTC_Core_to_Device
     SendTC_Core_to_Device ≔ SendTC_Core_to_Device ∪ {tc}
 end
```

Figure 7.51: Instances of a Shared Event After Model Decomposition, Bepi-Colombo System

7.3.8 4th and 5th Refinements: Core Sub-model Refinements

After model decomposition, each sub-model can be refined independently. It is one of the benefits of decomposing big models to some smaller sub-models, as described in Section 2.5. There are two more refinement levels for the core sub-model. In these refinements, the atomicity of four core events which are related to generating data *TMs* and control *TMs*, is decomposed in two levels, machine *M4* and machine *M5* in Figure 7.34.

The atomicity decomposition sequencing in the second refinement level, machine *M5* follows an approach called weak sequencing, which is described in Section 4.3.4. Considering atomicity decomposition of the *TCValid_ReplyCtrlTM* event in Figure 7.52, in first level, machine *M4*, it decomposed to two sub-events, and in the second level, machine *M5*, the *TCValid_ProcessCtrlTM* event decomposed to *Produce_DataTM* and *Send_DataTM* sub-events. The weak sequencing is applied to the atomicity decomposition of the *TCValid_ProcessCtrlTM* event. Considering weak sequencing, there is a sequence between the *Produce_DataTM* event, with solid line, and the *TCValid_CompleteCtrlTM* event. It means that before completing the process of generating data *TMs* for a valid *TC*, the corresponding data *TMs* should be produced. In contrast when sending produced *TMs*, the *Send_DataTM* event with a dashed line, can executed before or after the *TCValid_CompleteCtrlTM* event.

In the Event-B model, weak sequencing is applied using some control variables in guards like described before. Figure 7.53 presents the Event-B model of weak sequencing between *Send_DataTM* event and *TCValid_CompleteCtrlTM* event. In

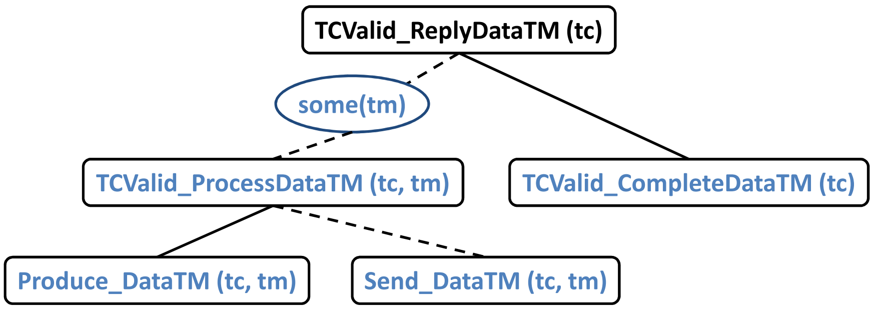
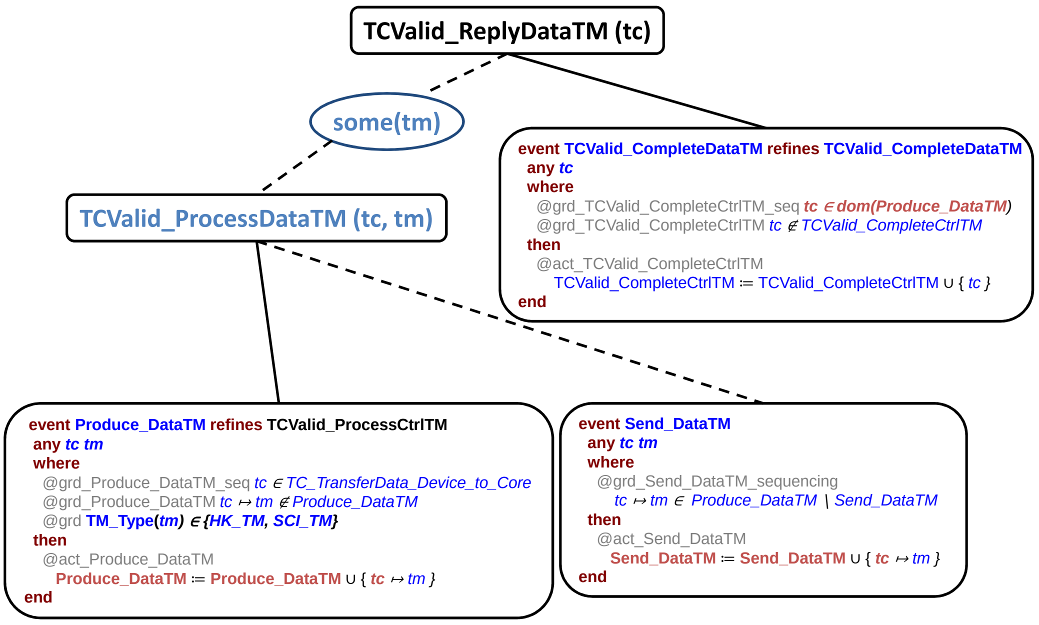


Figure 7.52: The Atomicity Decomposition Diagram in the Core Sub-Model, M4 and M5, BepiColombo System

grd_TCValid_CompleteCtrlTM of *TCValid_CompleteCtrlTM* event, *tc* is checked for belonging to *TCProduce_DataTM* variable which has been assigned in *Produce_DataTM* event, so there is no sequencing between *Send_DataTM* event and *TCValid_CompleteCtrlTM* event.



```
event TCValid_CompleteDataTM refines TCValid_CompleteDataTM
 any tc
 where
  @grd_TCValid_CompleteCtrlTM_seq tc ∈dom(Produce_DataTM)
  @grd_TCValid_CompleteCtrlTM tc ∉ TCValid_CompleteCtrlTM
 then
  @act_TCValid_CompleteCtrlTM
    TCValid_CompleteCtrlTM ≔ TCValid_CompleteCtrlTM ∪ { tc }
end
```

```
event Produce_DataTM refines TCValid_ProcessCtrlTM
 any tc tm
 where
  @grd_Produce_DataTM_seq tc ∈ TC_TransferData_Device_to_Core
  @grd_Produce_DataTM tc ↦ tm ∉ Produce_DataTM
  @grd TM_Type(tm) ∈{HK_TM, SCI_TM}
 then
  @act_Produce_DataTM
    Produce_DataTM ≔ Produce_DataTM ∪ { tc ↦ tm }
end
```

```
event Send_DataTM
 any tc tm
 where
  @grd_Send_DataTM_sequencing
    tc ↦ tm ∈ Produce_DataTM ∖ Send_DataTM
 then
  @act_Send_DataTM
    Send_DataTM ≔ Send_DataTM ∪ { tc ↦ tm }
end
```

Figure 7.53: Event-B Model of Weak Sequencing in the Core Sub-Model, M4 and M5, BepiColombo System

The some-replicator, described in Section 4.2.8, is used in the first refinement level. It adds a new parameter to the related sub-event, in this case *tm* is added to *TCValid_ProcessCtrlTM* in the first refinement and *Produce_DataTM* and *Send_DataTM* events in the second refinement.

This pattern is repeated for the other three events in production of control *TMs*. It is presented in Figure 7.54.

The difference between processing of control *TMs* and data *TMs* is that for each *TC* only one control *TM* is produced and sent from the core to the earth but for each *on*

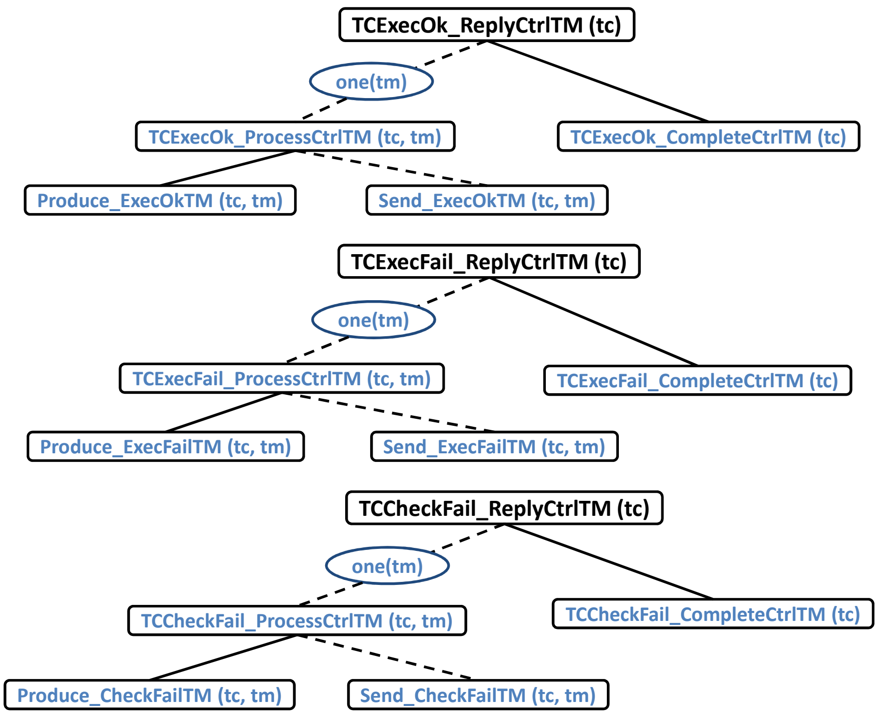

TCExecOk_ReplyCtrlTM (tc)
one(tm)
TCExecOk_ProcessCtrlTM (tc, tm)
TCExecOk_CompleteCtrlTM (tc)
Produce_ExecOkTM (tc, tm)
Send_ExecOkTM (tc, tm)
TCExecFail_ReplyCtrlTM (tc)
one(tm)
TCExecFail_ProcessCtrlTM (tc, tm)
TCExecFail_CompleteCtrlTM (tc)
Produce_ExecFailTM (tc, tm)
Send_ExecFailTM (tc, tm)
TCCheckFail_ReplyCtrlTM (tc)
one(tm)
TCCheckFail_ProcessCtrlTM (tc, tm)
TCCheckFail_CompleteCtrlTM (tc)
Produce_CheckFailTM (tc, tm)
Send_CheckFailTM (tc, tm)

Figure 7.54: The Atomicity Decomposition Diagrams in the Core Sub-Model, M4 and M5, BepiColombo System

TC one or more data *TM(s)* are produced and sent from the core to the earth. So the one-replicator, described in Section 4.2.9, is used in processing of control *TMs*. The invariants which specify the one-replicator properties are presented in Figure 7.55 for the first refinement, machine *M4*, and in Figure 7.56 for the second refinement, machine *M5*.

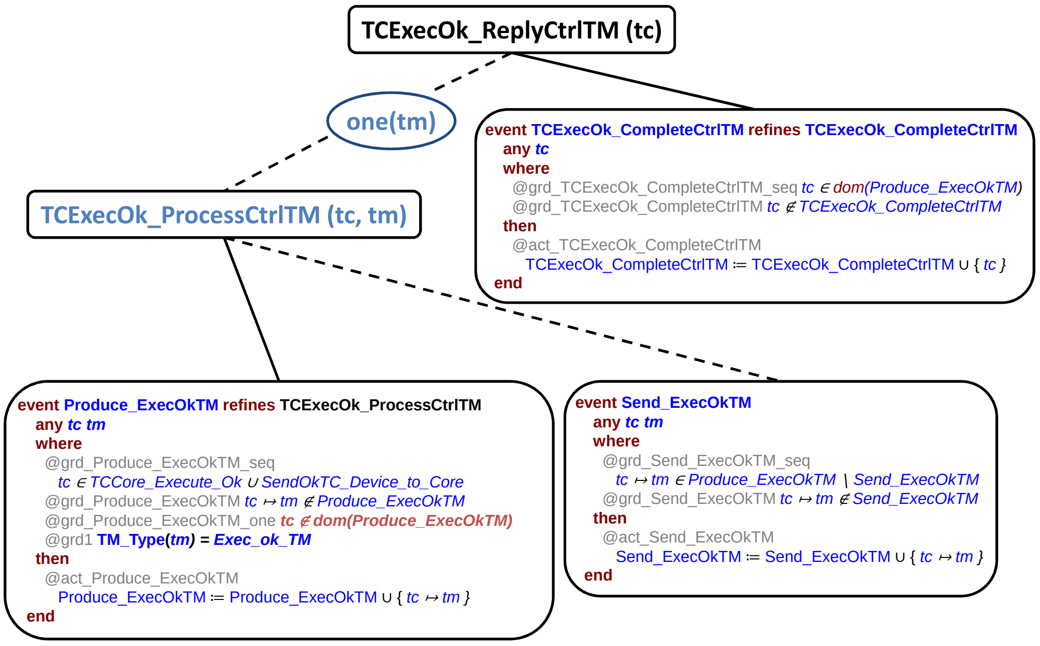
```
@inv_TCExecOk_ProcessCtrlTM_one ∀tc· card(TCExecOk_ProcessCtrlTM[{tc}]) ≤ 1
@inv_TCExecFail_ProcessCtrlTM_one ∀tc· card(TCExecFail_ProcessCtrlTM[{tc}]) ≤ 1
@inv_TCCheckFail_ProcessCtrlTM_one ∀tc· card(TCCheckFail_ProcessCtrlTM[{tc}]) ≤ 1
```

Figure 7.55: one-replicator Invariants, M4, BepiColombo System

```
@inv_TCExecOk_ProcessCtrlTM_one ∀tc· card(Produce_ExecOkTM[{tc}]) ≤ 1
@inv_TCExecFail_ProcessCtrlTM_one ∀tc· card(Produce_ExecFailTM[{tc}]) ≤ 1
@inv_TCCheckFail_ProcessCtrlTM_one ∀tc· card(Produce_CheckFailTMtc}]) ≤ 1
```

Figure 7.56: one-replicator Invariants, M5, BepiColombo System

Figure 7.57 presents the Event-B model of the first decomposition in Figure 7.54. Guard *grd_Produce_ExecOkTM_one* in the *Produce_ExecOkTM* event models the one-replicator property.



```
event TCExecOk_CompleteCtrlTM refines TCExecOk_CompleteCtrlTM
  any tc
  where
   @grd_TCExecOk_CompleteCtrlTM_seq tc ∈ dom(Produce_ExecOkTM)
   @grd_TCExecOk_CompleteCtrlTM tc ∉ TCExecOk_CompleteCtrlTM
  then
   @act_TCExecOk_CompleteCtrlTM
     TCExecOk_CompleteCtrlTM ≔ TCExecOk_CompleteCtrlTM ∪ { tc }
 end
```

```
event Produce_ExecOkTM refines TCExecOk_ProcessCtrlTM
  any tc tm
  where
   @grd_Produce_ExecOkTM_seq
     tc ∈ TCCore_Execute_Ok ∪ SendOkTC_Device_to_Core
   @grd_Produce_ExecOkTM tc ↦ tm ∉ Produce_ExecOkTM
   @grd_Produce_ExecOkTM_one tc ∉ dom(Produce_ExecOkTM)
   @grd1 TM_Type(tm) = Exec_ok_TM
  then
   @act_Produce_ExecOkTM
     Produce_ExecOkTM ≔ Produce_ExecOkTM ∪ { tc ↦ tm }
 end
```

```
event Send_ExecOkTM
  any tc tm
  where
   @grd_Send_ExecOkTM_seq
     tc ↦ tm ∈ Produce_ExecOkTM ∖ Send_ExecOkTM
   @grd_Send_ExecOkTM tc ↦ tm ∉ Send_ExecOkTM
  then
   @act_Send_ExecOkTM
     Send_ExecOkTM ≔ Send_ExecOkTM ∪ { tc ↦ tm }
 end
```

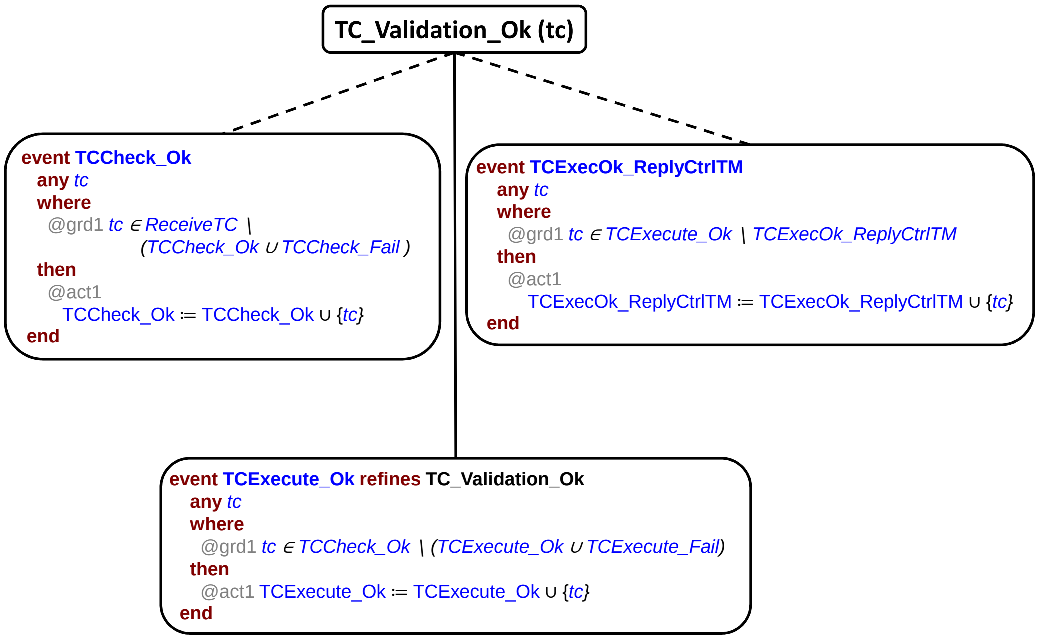
Figure 7.57: Event-B Model of one-replicator and Weak Sequencing in the Core Sub-Model, M4 and M5, BepiColombo System

7.3.9 Evaluation of Manual Event-B Model and Automatic Event-B Model

As described in the media channel system evaluation in Section 7.2.8, use of atomicity decomposition plug-in in creating the Event-B model of a system, ensures a higher level of consistency in encoding of the atomicity decomposition diagrams comparing to the manual version. Some differences between the automatic Event-B model, which is created with the plug-in, and the manual one in developing the BepiColombo system are presented here.

7.3.9.1 A Merged Guard versus Separate Guards

In the automatic Event-B model, there is a separate guard for each predicate generated in a separate translation rule, whereas in the manual Event-B model, we modelled all of the predicates in one unique guard. For instance here we compare three events from machine *M1* in the automatic Event-B model presented in Figure 7.42, and in the manual Event-B model presented in Figure 7.58. For example, considering *TCExecute_Ok* event in Figure 7.42, a sequencing guard called *grd_TCExecute_Ok_seq* is generated in TR_leaf8 (Section 5.4.3.8), a guard called *grd_TCExecute_Ok* is generated via TR_leaf9 (Section 5.4.3.9) and finally *grd1* is added manually to the event. Whereas in Figure 7.58, the predicates are merged in one guard called *grd1*.



```
event TCCheck_Ok
  any tc
  where
   @grd1 tc ∈ ReceiveTC ∖
              (TCCheck_Ok ∪ TCCheck_Fail )
  then
   @act1
     TCCheck_Ok ≔ TCCheck_Ok ∪ {tc}
 end
```

```
event TCExecOk_ReplyCtrlTM
  any tc
  where
   @grd1 tc ∈ TCExecute_Ok ∖ TCExecOk_ReplyCtrlTM
  then
   @act1
     TCExecOk_ReplyCtrlTM ≔ TCExecOk_ReplyCtrlTM ∪ {tc}
 end
```

```
event TCExecute_Ok refines TC_Validation_Ok
  any tc
  where
   @grd1 tc ∈ TCCheck_Ok ∖ (TCExecute_Ok ∪ TCExecute_Fail)
  then
   @act1 TCExecute_Ok ≔ TCExecute_Ok ∪ {tc}
 end
```

Figure 7.58: Manual Event-B Model, M1, BepiColombo System

Having separate guards slightly increases the number of *GRD* proof obligations, Section 2.4.4, which are generated for a refining event in the next refinement level; Since for each separate guard, a separate *GRD* proof is generated. Whereas in the manual Event-B model, just one *GRD* proof is generated for the merged guard. However the generated proof obligations in the automatic Event-B model are slightly simpler, because the corresponding separated guards are slightly simpler. In both manual and automatic Event-B models, the generated *GRD* proof obligations are discharged automatically.

7.3.9.2 Gluing Invariants

In the automatic Event-B model, a gluing invariant specifies an equality relationship between an abstract variable and the corresponding concrete variable. Whereas in the manual Event-B model, we specified a gluing invariant as a subset relationship between an abstract variable and the corresponding concrete variable. Because when the manual model was developed, the patterns for gluing invariants were insufficient.

As an instance, in the automatic Event-B model of machine *M1* as presented in Section 7.3.4, a gluing invariant is defined as follow:

$@inv\_TCExecute\_Ok\_gluing \quad TCExecute\_Ok = TC\_Validation\_Ok$

Whereas in the manual Event-B model the gluing invariant was defined as a subset relationship:

$@inv9 \quad TCExecute\_Ok \subseteq TC\_Validation\_Ok$

The later invariant causes some non-discharged proof obligations. For instance the *TCExecute_Ok/grd_TC_Validation_Ok/GRD* proof can not be proved in the manual model. To prove the stated proof obligation, we needed to add some more invariants. The extra invariants made the model complex and large and results in more number of proof obligations.

7.3.9.3 Model Decomposition

Decomposing the automatic Event-B model of machine *M3* to the core and device sub-models, results in seven shared events, presented in Figure 7.49. Whereas in model decomposition of the manual Event-B model to core and device sub-models, there were four shared events. *TCCore_Execute_Ok*, *TCCore_Execute_Fail* and *TC_GenerateData_in_Device* are shared events in the automatic model; Whereas in the manual model, *TCCore_Execute_Ok* and *TCCore_Execute_Fail* are core events, and *TC_GenerateData_in_Device* is a device event. The reason is explained in next paragraph.

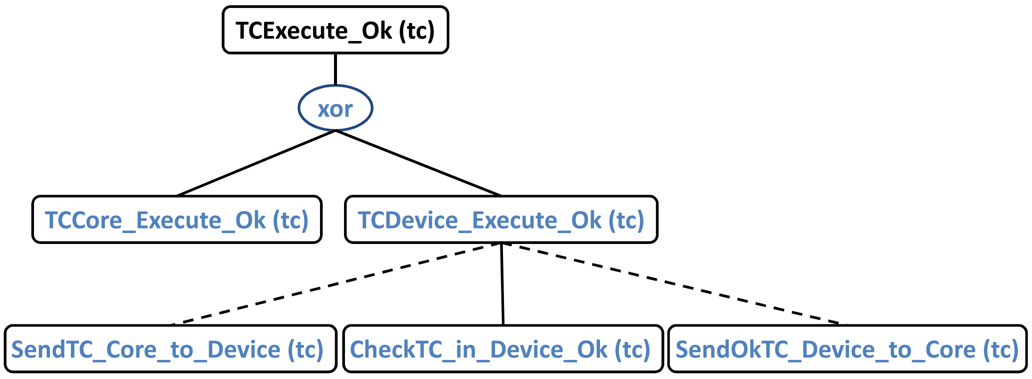

TCExecute_Ok (tc)
xor
TCCore_Execute_Ok (tc)
TCDevice_Execute_Ok (tc)
SendTC_Core_to_Device (tc)
CheckTC_in_Device_Ok (tc)
SendOkTC_Device_to_Core (tc)

Figure 7.59: Combined Atomicity Decomposition Diagram of *TCExecute_Ok* Event, BepiColombo System

Considering machine *M2*, Figure 7.45, and machine *M3*, Figure 7.47, the combined atomicity decomposition diagram of *TCExecute_Ok* event is presented in Figure 7.59. As the result of TR_xor3 (Section 5.4.4.3), in machine *M3*, a guard is generated for *TCCore_Execute_Ok* event to ensure that the other xor child is not executed before:

$@grd\_TCCore\_Execute\_Ok\_xor \quad tc \notin SendTC\_Core\_to\_Device$

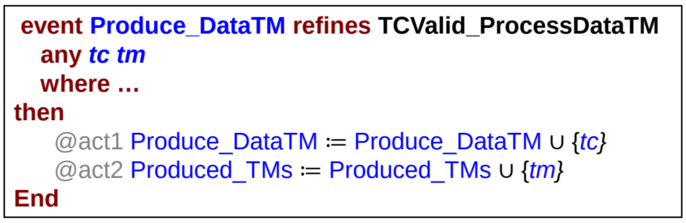
On one hand *SendTC_Core_to_Device* is a device variable, while *TCCore_Execute_Ok* event uses other core variables, i.e., *TCCheck_Ok* in below guard:

$@grd\_TCCore\_Execute\_Ok\_seq \quad tc \in TCCheck\_Ok$

Therefore the xor guard which uses a device variable make the *TCCore_Execute_Ok* event as a shared event in the automatic event-B model. Whereas in the manual Event-B model we did not add the xor guard, since, as presented in Figure 7.46, guard *grd3* ensures the mutual exclusiveness of *TCCore_Execute_Ok* event and *SendTC_Core_to_Device* event. In the automatic Event-B model, guard *@grd_TCCore_Execute_Ok_xor* is generated automatically from TR_xor3 (Section 5.4.4.3), and guard *grd3* is added manually to the automatic Event-B model.

7.3.9.4 some-replicator, one-replicator

In the manual Event-B model, we did not explicitly specify the *tm*s associated with a valid *tc*. For instance part of the *Produce_DataTM* event in the manual Event-B model is presented in Figure 7.60. As the result of the some-replicator there is one new parameter called *tm* added to the *Produce_DataTM* event. The *Produce_DataTM* variable is a one dimension set and, *tm*s are added to a separate set called *Produced_TMs*. As a result in the manual model we were not able to track the *tm*s produced for a specific valid *tc*.



```
 event Produce_DataTM refines TCValid_ProcessDataTM
   any tc tm
   where ...
then
     @act1 Produce_DataTM ≔ Produce_DataTM ∪ {tc}
     @act2 Produced_TMs ≔ Produced_TMs ∪ {tm}
End
```

Figure 7.60: some-replicator Event, Manual Event-B Model, BepiColombo System

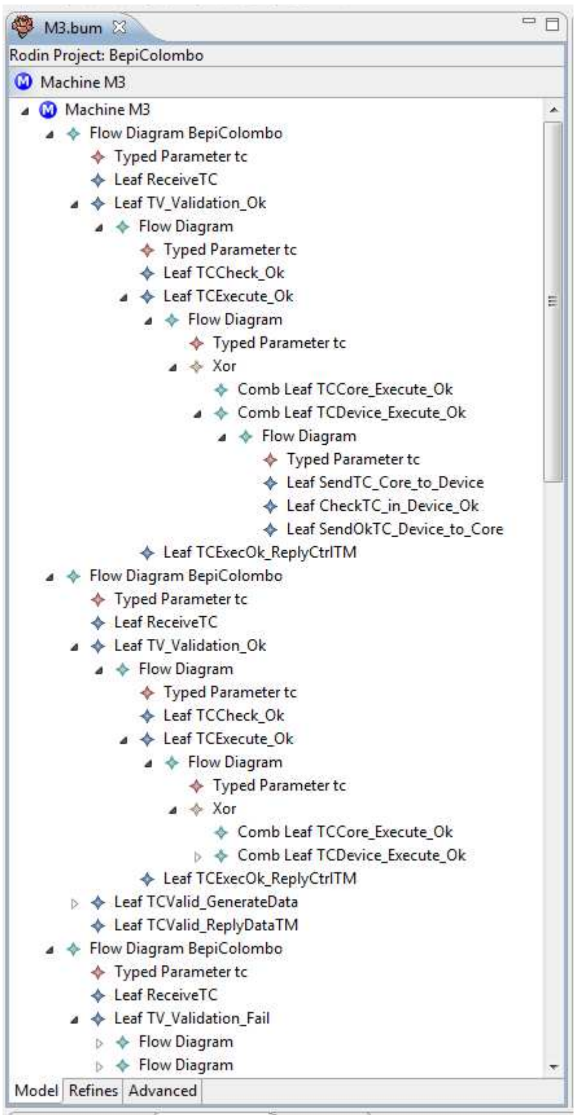
Whereas in the automatic Event-B model, as presented in Figure 7.53, the *Produce_DataTM* variable is a cartesian product of *TC* and *TM* as a result of being a child of a some-replicator. Therefore in the automatic model tracking *tm*s associated with a valid *tc* is possible, and the model is more accurate. The same modelling style is used for the one-replicator in the atomicity decompositions presented in Figure 7.54.

7.3.9.5 Naming Protocol

In the automatic Event-B model, invariants and guards have clear labels following a unique labelling protocol which is used in the patterns in Section 4.2 and the translation rules in Section 5.4. Whereas in the manual Event-B model the invariants and guards do not follow a specific labelling protocol, for example see Figure 7.58. Having a clear labelling protocol helps to understand the model easily as it can help to recognise the aim of each invariant or guard. For example, in the automatic Event-B model, invariants and guards which describe the sequencing between events are labelled with *_seq* suffix.

7.3.9.6 Tool Application: Atomicity Decomposition Model of the Bepi-Colombo System

Tha atomicity Decomposition model of the third refinement level of the BepiColombo system, generated with the atomicity decomposition plug-in is presented in Figure 7.61.



```
M3.bum
Rodin Project: BepiColombo
Machine M3
Machine M3
  Flow Diagram BepiColombo
    Typed Parameter tc
    Leaf ReceiveTC
    Leaf TV_Validation_Ok
      Flow Diagram
        Typed Parameter tc
        Leaf TCCheck_Ok
        Leaf TCExecute_Ok
          Flow Diagram
            Typed Parameter tc
            Xor
              Comb Leaf TCCore_Execute_Ok
              Comb Leaf TCDevice_Execute_Ok
                Flow Diagram
                  Typed Parameter tc
                  Leaf SendTC_Core_to_Device
                  Leaf CheckTC_in_Device_Ok
                  Leaf SendOkTC_Device_to_Core
        Leaf TCExecOk_ReplyCtrlTM
  Flow Diagram BepiColombo
    Typed Parameter tc
    Leaf ReceiveTC
    Leaf TV_Validation_Ok
      Flow Diagram
        Typed Parameter tc
        Leaf TCCheck_Ok
        Leaf TCExecute_Ok
          Flow Diagram
            Typed Parameter tc
            Xor
              Comb Leaf TCCore_Execute_Ok
              Comb Leaf TCDevice_Execute_Ok
        Leaf TCExecOk_ReplyCtrlTM
    Leaf TCValid_GenerateData
    Leaf TCValid_ReplyDataTM
  Flow Diagram BepiColombo
    Typed Parameter tc
    Leaf ReceiveTC
    Leaf TV_Validation_Fail
      Flow Diagram
      Flow Diagram
Model | Refines | Advanced
```

Figure 7.61: Atomicity Decomposition Model of the BepiColombo System

7.3.10 Overview of Proof Obligation

The entire development of the BepiColombo system involves one abstract model followed by three refinement levels before model decomposition and two refinement levels of the core sub-model after model decomposition. In the last refinement level before model decomposition, *M3*, there are 16 variables and 16 events as seen in Table 7.2. After model decomposition *Core_M3* contains 12 variables and 14 events; and *Device_M3* contains 4 variables and 9 events. It shows one of the benefits of model decomposition in breaking a big model into some smaller sub-models. The sum of the variables for each sub-model is equal to the number of variables of non-decomposed model *M3*. That is not a coincidence since in a shared event model decomposition, the variables are partitioned among sub-models. However the sum of the events of *Core_M3* and *Device_M3* is not equal to the number of events of *M3*, since there are seven shared events which appear in both sub-models, as seen Figure 7.48.

Component	**variables**	**Events**
M0	5	5
M1	10	10
M2	12	12
M3	16	16
Core_M3	12	14
Device_M3	4	9
Core_M4	16	18
Core_M5	20	22

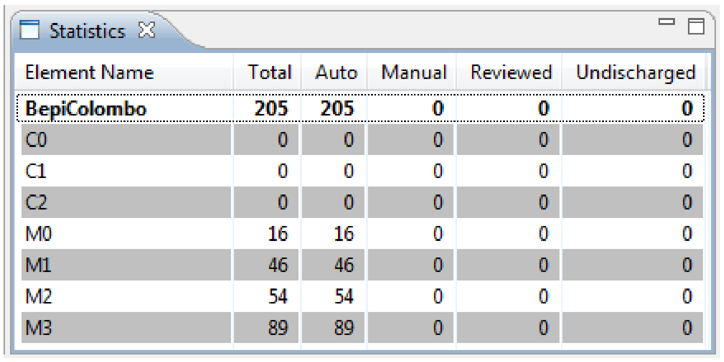
Table 7.2: Summary of the Automatic BepiColombo Development, Number of Variables and Events

Table 7.3 shows the number of variables and events for the manual Event-B model. As can be seen in the table, the number of variables in machine *M3* of the manual Event-B model is more than the one in the automatic Event-M model. As explained in Section 7.3.9.4, considering a separate variable for the new some-replicator parameter causes in greater number of variables in the manual Event-B model.

Component	**variables**	**Events**
M0	5	5
M1	10	10
M2	12	12
M3	18	16
Core_M3	13	13
Device_M3	5	7
Core_M4	21	17
Core_M5	29	21

Table 7.3: Summary of the Manual BepiColombo Development, Number of Variables and Events

A summary of the proof obligations for the automatic Event-B model can be seen in Figure 7.62. The overall 205 generated proof obligations discharged automatically. Most of the proof obligations are related to gluing invariants and guard strengthening. Gluing invariants which show connections between abstract variables and concrete variables, should be proved to be preserved by each action of each event. In guard strengthening proof obligations it should be proved that for refining events the concrete guards are stronger than the abstract guards.

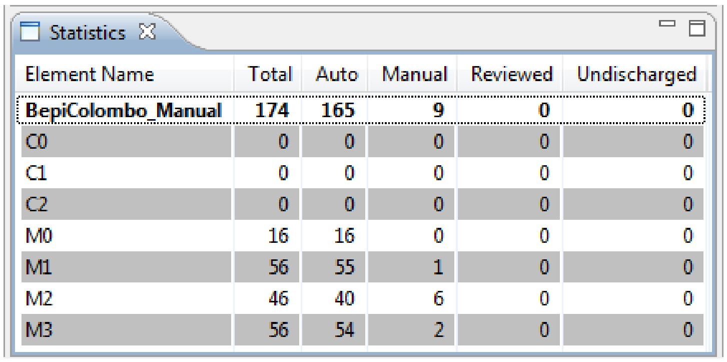


Statistics

Element Name	Total	Auto	Manual	Reviewed	Undischarged
BepiColombo	**205**	**205**	**0**	**0**	**0**
C0	0	0	0	0	0
C1	0	0	0	0	0
C2	0	0	0	0	0
M0	16	16	0	0	0
M1	46	46	0	0	0
M2	54	54	0	0	0
M3	89	89	0	0	0

Figure 7.62: Proof Obligation Statistics for the Automatic BepiColombo Event-B Model

Figure 7.63 presents the summary of the proof obligations for the manual Event-B model. The number of proof obligations in the manual model is slightly less than the automatic ones. As described in Section 7.3.9.1, having separate guards in the automatic model increases the number of proof obligations. However all of the automatic model's proofs are discharged automatically, whereas in the manual model, nine proofs had to be discharged interactively.



Statistics

Element Name	Total	Auto	Manual	Reviewed	Undischarged
BepiColombo_Manual	**174**	**165**	**9**	**0**	**0**
C0	0	0	0	0	0
C1	0	0	0	0	0
C2	0	0	0	0	0
M0	16	16	0	0	0
M1	56	55	1	0	0
M2	46	40	6	0	0
M3	56	54	2	0	0

Figure 7.63: Proof Obligation Statistics for the Manual BepiColombo Event-B Model

7.4 Conclusion

We modelled the media channel system and the BepiColombo system, a space craft, using the atomicity decomposition approach. The developments of both case studies have been done first manually and later using our atomicity decomposition tool support. The automatic models, created by our atomicity decomposition tool support, have been outlined and then an evaluation to compare the manual models with the automatic ones have been presented. Although the manual and automatic models capture the same behaviours, as a result of using our atomicity decomposition plug-in in creating the automatic model of the systems, the automatic models are more consistent and systematic in encoding of the diagrams.

Since the atomicity decomposition approach and the model decomposition approach aim to tackle the difficulties of modelling complex systems in Event-B, combining them is of interest. The BepiColombo development addresses the combination of the atomicity decomposition approach and model decomposition approach. Further refinements structured with the atomicity decomposition diagrams, have been applied to the BepiColombo system after decomposing it to the core and device sub-models using model decomposition.

The major benefit of using atomicity decomposition diagrams in structuring refinement were highlighted in the development of the case studies. During manual development of the case studies, the atomicity decomposition approach has been improved. Some new constructors and features have been discovered. The assessment results gained from the development of the case studies are presented in Chapter 8.

Chapter 8

Evaluation of Atomicity Decomposition in Case Study Developments

8.1 Introduction

The major benefit of using atomicity decomposition diagrams in showing the explicit relationships between events of different levels of refinement and presenting the diagrammatic notation of event sequencing were highlighted in the development of both case studies.

Moreover, in the media channel development the diagrams facilitated the linking of requirements of the different protocol phases (establish, modify and close) with the formal development. As presented in the initial model of the system in Figure 7.4, each phase corresponds to one node in a diagram which is modelled in one event in the Event-B model. Then, in each level of refinement we focused on breaking the atomicity of a specific phase, the establish phase in the first refinement, the modify phase in the second refinement, the close phase in the third refinement and so on.

This chapter discusses how the atomicity decomposition approach helped us in the development of the media channel system and the BepiColombo system. We will explain what we have discovered in terms of methodological results, new constructors and new features in the atomicity decomposition approach. Finally we will outline how the outputs of the case studies influenced the definition of the atomicity decomposition diagram patterns.

8.2 Exploring Alternatives

The possibility of a diagrammatic view of the developments has given us the chance to decide about alternatives in atomicity decomposition of an event. This decision can be done before taking the effort of changing the Event-B model. For instance in the media channel development, for refining the *modify* event we had two possible ways. The first one is shown in Figure 8.1, and the second is shown in Figure 8.2. The atomicity decomposing of the *modify* event is done in two levels of refinement in Figure 8.1 whereas by using the second decomposition in Figure 8.2, we can reduce it to one level of refinement. In the second way we separate the case splitting in two separate decomposition diagrams, shown in Figure 8.2, We use the technique presented in Section 4.3.3. In the media channel system development, as presented in Section 7.2.4, we chose the atomicity decomposition in Figure 8.2 with fewer number of refinements to reduce the effort of modelling. This case shows how we can explore event refinement alternatives using atomicity decomposition diagrams before creating the Event-B model.

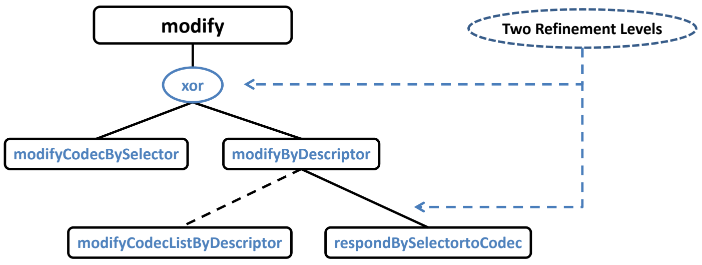


Figure 8.1: Decomposing Atomicity of *modify* Event in Two Levels of Refinement

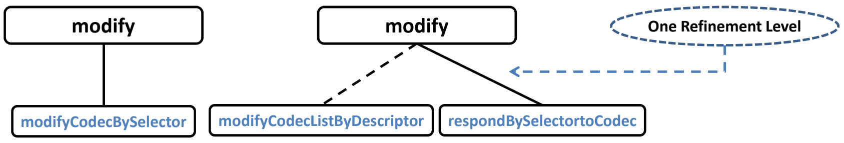


Figure 8.2: Decomposing Atomicity of *modify* Event in One Level of Refinement

Therefore the atomicity decomposition approach can help us find good ways of refining events before getting involved with the complex Event-B model, and this output can be highlighted as one of the outcomes of using the atomicity decomposition approach.

8.3 Preventing of Wrong Event Decomposition

Using atomicity decomposition diagrams can prevent wrong event refinement before starting Event-B modelling . It can result in earlier detection of wrong refinements in the modelling process. Figure 8.3 presents one possible way of decomposing the atomicity of validation phase in the development of BepiColombo system. Figure 8.3 states that a validation can succeed, *TC_Validation_Ok* event, or fail, *TC_Validation_Fail* event. Then a successful validation means successful syntax validation, *TCCheck_Ok* event, followed by a successful semantic validation, *TCExecute_Ok* event. And a failed validation fails either in the syntax check, *TCCheck_Fail* event, or the semantics check, *TCExecute_Fail*.

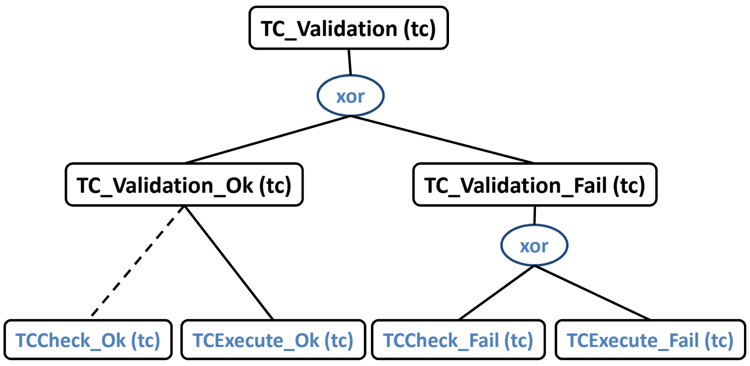


Figure 8.3: Wrong Atomicity Decomposition

Considering xor-constructor and sequencing definitions, in the diagram, the possible event executions are:

$< TCCheck\_OK(tc), TCExecute\_OK(tc) >$
$< TCCheck\_Fail(tc) >$
$< TCExecute\_Fail(tc) >$

Therefore this decomposition does not cover all necessary event execution according to the requirements, explained in Section 7.3.4. It does not cover the following trace:

$< TCCheck\_OK(tc), TCExecute\_Fail(tc) >$

Therefore using atomicity decomposition diagram helped us to prevent a wrong refinement before doing the effort of Event-B modelling. As a result, we have changed the decomposition of the validation phase to a valid one which was presented in Section 7.3.4.

8.4 Events Tracking

A combined atomicity decomposition diagram provides the overall visualization of refinement structure. Figure 8.4 presents a part of the overall refinement structure of the BepiColombo system.

Using the overall view of refinement structure gives us the chance of tracking possible event execution traces by following leaf events from left to right. It provides the visualization of the entire Event-B model which is not possible by just using the refinement process. Event tracking helps us to describe the system requirements which can help us to identify requirement coverage.

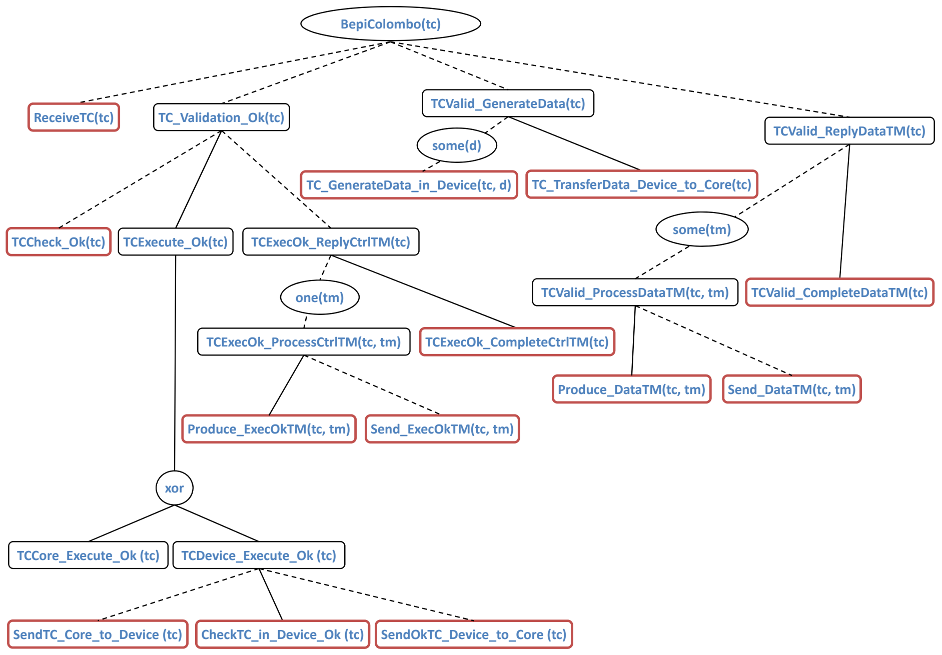


Figure 8.4: Overall Refinement Structure After Model Decomposition, Bepi-Colombo System

For instance, in Figure 8.4 one of the possible execution traces is shown below. It shows the model covers the requirements in the case that the validation is ok and the TC belongs to a device.

$< ReceiveTC,$
$TCCheck\_Ok,$
$SendTC\_Core\_to\_Device, CheckTC\_in\_Device\_Ok, SendOkTC\_Device\_to\_Core,$
$Produce\_ExecOkTM, Send\_ExecOkTM, TCExecOk\_CompleteCtrlTM,$
$TC\_GenerateData\_in\_Device, TC\_TransferData\_Device\_to\_Core,$
$Produce\_DataTM, Send\_DataTM, TCValid\_CompleteDataTM >$

Having xor-constructor and weak sequencing result in possibilities of other event traces. For instance considering xor-constructor in decomposing the $TCExecute\_Ok$ event into $TCCore\_Execute\_Ok$ and $TCDevice\_Execute\_Ok$ sub-events, another possible event trace, when the TC belongs to the core, is to the replace execution of

$< SendTC\_Core\_to\_Device, CheckTC\_in\_Device\_Ok, SendOkTC\_Device\_to\_Core >$

with $TCCore\_Execute\_Ok$. Also considering weak interpretation between $Send\_ExecOkTM$ and $TCExecOk\_CompleteCtrlTM$, we can swap the place of $Send\_ExecOkTM$ and $TCExecOk\_CompleteCtrlTM$ in the previous execution trace.

8.5 Requirements Clarification

We experienced clarifying and re-structuring requirements during the BepiColombo development using atomicity decomposition diagrams. As presented in the previous section the diagrams help us to identify the possible events execution, and it can result in clarifying the requirements.

In the BepiColombo development, in the second refinement level, we recognised that the difference between TCs belong to the core and TCs belong to one of the devices should be distinguished. This recognition which is a result of diagrams, helped us to structure the requirements related to the core and device TCs in the next refinement level. As shown in Section 7.3.5, using the xor-constructor to split the core case and the device case, the requirements related to the core and devices are explicitly structure in the diagram.

As another example, we came up with the diagram shown in Figure 8.5 in the third refinement level. Reviewing the event traces, showed us that it does not cover the data generation which should be done in a device. Therefore we ended up with the diagram shown in Figure 8.6. In this diagram the data generation, *TCValid_GenerateData* event, is added and refined in one level.

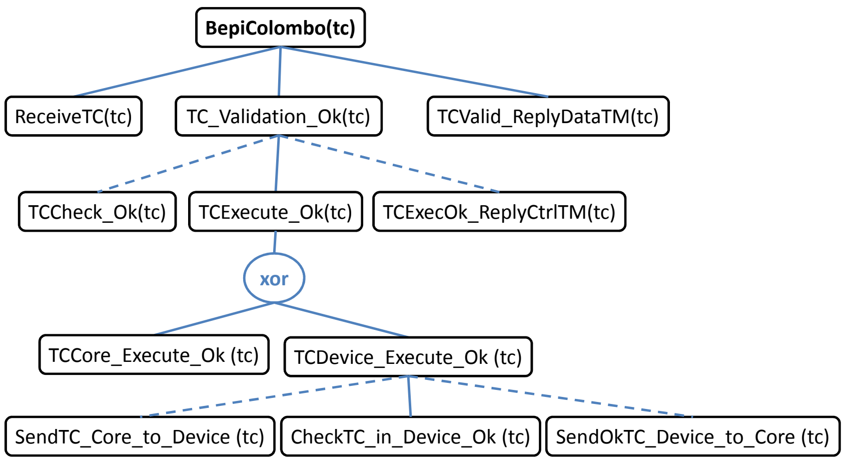


Figure 8.5: Atomicity Decomposition Diagram Without Considering Data Generation Requirement, BepiColombo System

Atomicity decomposition diagrams make the process of clarifying and re-structuring requirements easier comparing with just using the Event-B textual model, since dealing with the visual view of the event sequencing of the model is easier than dealing with the textual model only.

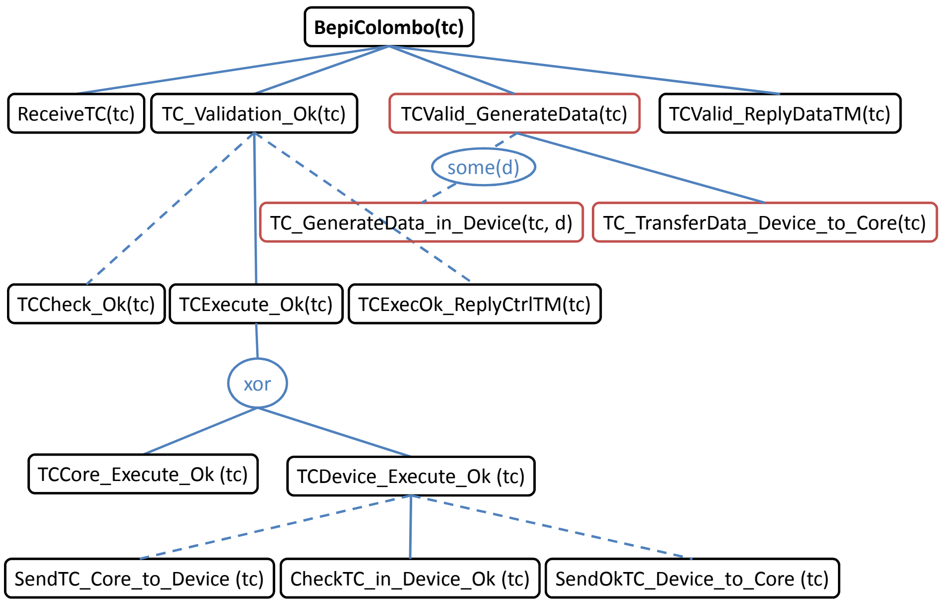


Figure 8.6: Atomicity Decomposition Diagram After Clarifying Data Generation Requirement, BepiColombo System

8.6 Combining Atomicity Decomposition and Model Decomposition

Development of the BepiColombo system addresses the use of atomicity decomposition and model decomposition together in Event-B modelling. Atomicity decomposition diagrams help us find the appropriate point to apply model decomposition. Atomicity decomposition provides an overall visualization of the refinement process which helps us to decide about decomposing atomicity of those events which lead us to an appropriate point to apply model decomposition. This decision can be made in a visual diagrammatic environment of atomicity decomposition which is easier to deal with compared to getting involved in difficulties of a complex Event-B model. The strategy to decide about an appropriate point of applying model decomposition in this case study, is explained in the next paragraph.

Figure 8.7 illustrates the overall refinement view of the abstract model followed by two refinement levels. At this point without getting involved in the complications of the Event-B model, we can decide about having more atomicity decomposition before model decomposition. Our strategy in this case study is to end up with leaf events which belong to one of these categories before starting model decomposition: core sub-model events, device sub-model events or shared events. A leaf event is a node without any child, which appears as an event of the last refinement level in the Event-B model. The $ReceiveTC$, $TCCheck\_Ok$, $TCExecOk\_ReplyCtrlTM$ and $TCValid\_ReplyDataTM$ events are the core events, and the $TCCore\_Execute\_Ok$ event is a shared event. On the other hand the $TCDevice\_Execute\_Ok$ event and the $TCValid\_GenerateData$

event partly belong to the device and are partly related to a shared activity between the core and devices. So we come up with one more atomicity decomposition level which is shown in Figure 8.6. In this figure, including the abstract model followed by three refinement levels, all leaf events belong to one of the mentioned categories. Among the newest events *CheckTC_in_Device_Ok* belongs to the device sub-model and *TC_GenerateData_in_Device*, *SendTC_Core_to_Device*, *SendOkTC_Device_to_Core* and *TC_TransferData_Device_to_Core* belong to the shared events category. Considering our strategy for this case study, this step is a appropriate point to apply model decomposition, since each leaf event belongs to one of these categories: core sub-model events device sub-model events or shared events.

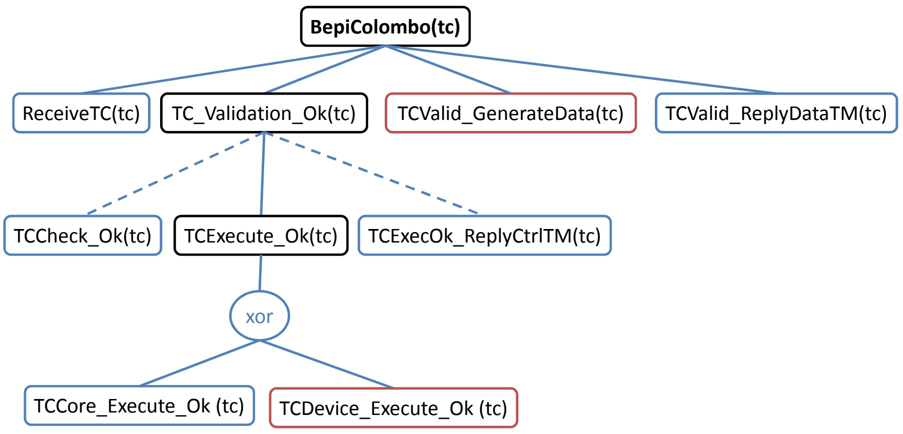
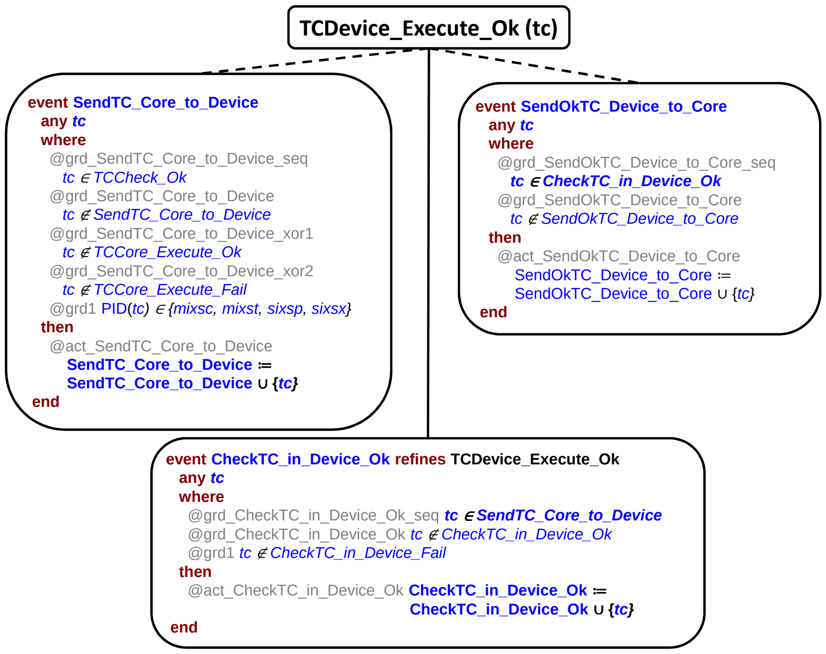


Figure 8.7: Overall Refinement Structure, Abstract Model and Two Refinement Levels, BepiColombo System

Model decomposition preserves refinement including event sequencing of the overall system in atomicity decomposition. Event sequencing in the atomicity decomposition approach is preserved after applying model decomposition to the Event-B model. Consider atomicity decomposition of *TCDevice_Execute_Ok* in the last refinement level before model decomposition in Figure 8.8. As described before, the sequencing is managed with some control variables added in some guards and actions of the events. Figure 8.9 presents the device sub-model events after applying shared-event model decomposition. The event sequencing is preserved in the device sub-model, although variables are divided between two sub-models. *CheckTC_in_Device_Ok* is a device event and is left without any change. *SendTC_Core_to_Device* and *SendOkTC_Device_to_Core* are shared events. As a result of model decomposition the guards which use core variables, *TCCkeck_Ok*, *TCCore_Execute_Ok* and *SendOkTC_Device_to_Core*, are removed. This does not affect the sequencing since the control variable, *SendTC_Core_to_Device* and *CheckTC_in_Device_Ok*, are device variables.

Finally, as shown in Figure 7.34, atomicity decomposition can be continued after model decomposition. So based on our experience we believe that applying atomicity decomposition and model decomposition together can be beneficial in Event-B modelling since both of them are intend to manage complexity in developing the model of large systems.

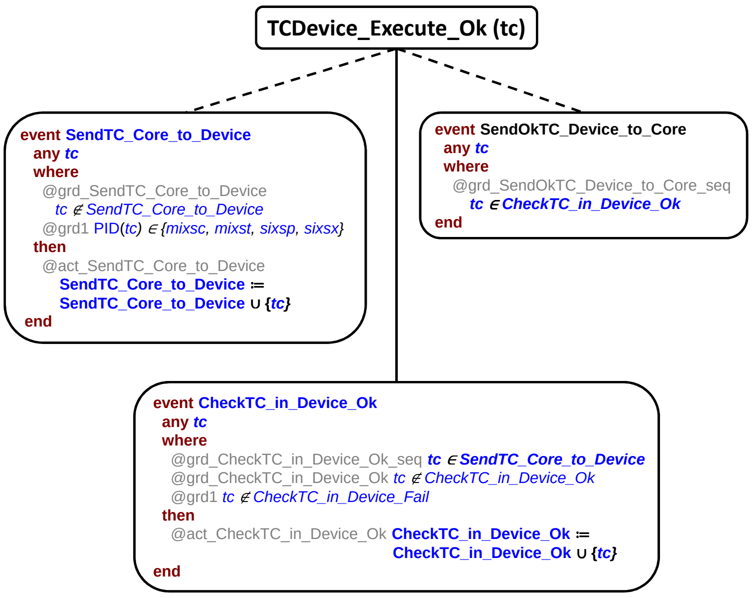


```
event SendTC_Core_to_Device
  any tc
  where
   @grd_SendTC_Core_to_Device_seq
     tc ∈ TCCheck_Ok
   @grd_SendTC_Core_to_Device
     tc ∉ SendTC_Core_to_Device
   @grd_SendTC_Core_to_Device_xor1
     tc ∉ TCCore_Execute_Ok
   @grd_SendTC_Core_to_Device_xor2
     tc ∉ TCCore_Execute_Fail
   @grd1 PID(tc) ∈ {mixsc, mixst, sixsp, sixsx}
  then
   @act_SendTC_Core_to_Device
      SendTC_Core_to_Device ≔
      SendTC_Core_to_Device ∪ {tc}
 end
```

```
event SendOkTC_Device_to_Core
  any tc
  where
   @grd_SendOkTC_Device_to_Core_seq
     tc ∈ CheckTC_in_Device_Ok
   @grd_SendOkTC_Device_to_Core
     tc ∉ SendOkTC_Device_to_Core
  then
   @act_SendOkTC_Device_to_Core
      SendOkTC_Device_to_Core ≔
      SendOkTC_Device_to_Core ∪ {tc}
 end
```

```
event CheckTC_in_Device_Ok refines TCDevice_Execute_Ok
  any tc
  where
   @grd_CheckTC_in_Device_Ok_seq tc ∈ SendTC_Core_to_Device
   @grd_CheckTC_in_Device_Ok tc ∉ CheckTC_in_Device_Ok
   @grd1 tc ∉ CheckTC_in_Device_Fail
  then
   @act_CheckTC_in_Device_Ok CheckTC_in_Device_Ok ≔
                             CheckTC_in_Device_Ok ∪ {tc}
 end
```

Figure 8.8: Event Sequencing Before Model Decomposition, BepiColombo System



```
event SendTC_Core_to_Device
  any tc
  where
   @grd_SendTC_Core_to_Device
     tc ∉ SendTC_Core_to_Device
   @grd1 PID(tc) ∈ {mixsc, mixst, sixsp, sixsx}
  then
   @act_SendTC_Core_to_Device
      SendTC_Core_to_Device ≔
      SendTC_Core_to_Device ∪ {tc}
 end
```

```
event SendOkTC_Device_to_Core
  any tc
  where
   @grd_SendOkTC_Device_to_Core_seq
     tc ∈ CheckTC_in_Device_Ok
 end
```

```
event CheckTC_in_Device_Ok
  any tc
  where
   @grd_CheckTC_in_Device_Ok_seq tc ∈ SendTC_Core_to_Device
   @grd_CheckTC_in_Device_Ok tc ∉ CheckTC_in_Device_Ok
   @grd1 tc ∉ CheckTC_in_Device_Fail
  then
   @act_CheckTC_in_Device_Ok CheckTC_in_Device_Ok ≔
                             CheckTC_in_Device_Ok ∪ {tc}
 end
```

Figure 8.9: Event Sequencing Preserved After Model Decomposition, BepiColombo System

8.7 New Constructors and Features

8.7.1 Introduction

This section outlines how the manual development of case studies led us to improvements in the atomicity decomposition approach. During manual development of case studies, the need for some new constructors and features was discovered. The discovered constructors and features helped us to define the atomicity decomposition patterns and features, presented in Chapter 4. And then they helped us to describe the language and translation rules in a formal description, presented in Chapter 5. Finally based on the patterns, the language description and translation rules, tool support was developed, presented in Chapter 6.

This section first addresses identified constructors and then identified features.

8.7.2 New Constructors

In the media channel development, two constructors have been identified. First, the loop constructor in the most abstract level, presented in Section 7.2.2. Second, the xor-constructor in the fourth and fifth refinement levels, presented in Section 7.2.6 and Section 7.2.7 respectively. Later the loop constructor was presented as a pattern in Section 4.2.3, and the xor-constructor was presented as a pattern in Section 4.2.6. The xor-constructor motivated us to define other logic operators: the and-constructor presented in Section 4.2.4 and the or-constructor is presented in Section 4.2.5.

The xor-constructor later has been applied to the second refinement level of the BepiColombo development as presented in Section 7.3.5. In the BepiColombo development, the need for some-replicator has been discovered, and some-replicator is used in the third refinement level, presented in Section 7.3.6. Also it is applied to the fourth refinement level of the core sub-model for several times, presented in Section 7.3.8. The some-replicator pattern is presented in Section 4.2.8. The some-replicator motivated us to define all-replicator (Section 4.2.7) and the one-replicator (Section 4.2.9). The all-replicator is first introduced in [24], and the some-replicator is first introduced in the presentation slides of [24].

We believe that these new constructors would be practical in the future.

8.7.3 Additional Features

The features that were explained in Section 4.3, are derived from case study developments. These features are addressed here.

In both case studies, we found that describing the most abstract level in an informal diagram, can help understanding. Therefore the most abstract level diagram defined and presented in Section 4.3.1.

As have been seen in previous sections of this chapter, combining atomicity diagrams of different refinement levels is beneficial in our developments. A combined atomicity decomposition diagram provides an overall visualization of the refinement structure in Event-B modelling. This feature was presented in Section 4.3.2.

Multiple atomicity decompositions in the process of refining a single event have been used during both case study developments. In the media channel development, it is used in the first and the second refinement levels. And in the BepiColombo development, it is used in the first refinement level. As described in Section 8.2, having multiple atomicity decompositions for a single event can reduce the number of refinement levels and as a result can reduce the complexity of a Event-B model. This feature was presented in Section 4.3.3.

In the refinements after model decomposition in the BepiColombo system, presented in Section 7.3.8, we found out that, a weaker interpretation of sequencing is needed. It motivated us to define the strong and weak sequencing, which presented in Section 4.3.4.

Different atomicity decomposition diagrams can share a single sub-event. The shared sub-event is transformed into a single event in the Event-B model. Considering the BepiColombo development in Figure 7.41, *TCCheck_Ok* sub-event node is shared in the first two atomicity decomposition diagrams. In the Event-B model, it is modelled with a single *TCCheck_Ok* event.

We have tried all alternatives presented in Section 4.3.5, for loop resetting in the manual development of the media channel system. As a result, as justified in Section 4.3.5, we decided to use a separate event as a loop resetting event, as presented for the media channel system in Section 7.2.4.

Finally, we have applied different approaches to model ordering in Event-B, presented in Section 4.4, for the media channel development. And as justified in Section 4.4, we adopted to use the subset approach. Considering the subset approach which was used in both case studies, each node in diagram corresponds to a set in each Event-B event. These sets play the role of control variables for controlling event sequences. This experience helps us to define the translation rules from diagram to the Event-B model.

8.8 Conclusion

The benefits of the atomicity decomposition approach were gradually presented via an overview of the approach in Chapter 3, and the presentation of patterns in Chapter 4.

The methodological results of using this approach in the development of two complex case studies have been reviewed in the current chapter.

The benefits of the atomicity decomposition approach are summarised as follows:

- The atomicity decomposition diagrams explicitly illustrate the relationships between refinement levels, which is not explicit just using the Event-B notation.
- The explicit ordering between events are presented in a diagrammatic notation of the atomicity decomposition approach. Whereas the Event-B text can model ordering in an implicit way.
- Using atomicity decomposition diagrams enables us to explore alternatives of refining an abstract event before getting involved with the complexity of Event-B modelling.
- Earlier detection of wrong refinement in the modelling process is one of the benefits of using atomicity decomposition diagrams.
- The atomicity decomposition approach provides the overall visualization of refinement structure, which gives us the ability to track events and requirement clarification via a combined atomicity decomposition diagram.
- The atomicity decomposition approach can be combined effectively with model decomposition. Since these two techniques aim to tackle the difficulties of modelling complex systems, combining them is of interest.
- The atomicity decomposition approach provided with tool support, can address automatic model generation in Event-B, which can decrease the modelling effort.

Chapter 9

Conclusions and Future Works

9.1 Conclusion

It was mentioned that modelling should be considered as an early stage in the software development process. However we are aware of difficulties in building models of complex systems. If these difficulties make software engineers reluctant to do modelling, it will be left out from the developing cycle. Thus some techniques are required to solve these difficulties.

The key factor in this thesis was presenting the atomicity decomposition approach and improving its methodology, as a technique helping us to model complex systems in Event-B notation using the Rodin tool. We have outlined how atomicity decomposition can be beneficial in the incremental development of two large case studies, and how the formal description of atomicity decomposition language and translation rules can be helpful in improving the methodology of the atomicity decomposition approach. The atomicity decomposition tool was developed as a plug-in supported by Event-B toolkit, Rodin.

The contributions we have completed consist of five parts:

- (i) Modelling and proof of the media channel system which contains a level of abstraction followed by five refinement layers (published in "Formal Methods for Components and Objects" (FMCO) 2009 conference [1]). In developing the Event-B model of the system we focus on evaluating the atomicity decomposition approach using structural diagrams in modelling the requirements of different phases.

  From the evaluation we outlined how using atomicity decomposition augmented with refinement in Event-B can be useful in the modelling process of a complex system. Exploring alternatives of decomposing atomicity of an event using the atomicity decomposition diagrams before getting involved with complexity of an Event-B model is evaluated. Also we have shown how using atomicity decomposition diagrams can prevent a wrong event refinement.

Some new construct patterns such as the loop constructor and xor-constructor have been discovered. The media channel development presented how different atomicity decomposition constructs, such as sequential events, loop constructor and case splitting (xor-constructor) are modelled in Event-B model by providing some refining/non-refining events, guards and invariants.

- (ii) Modelling and proof of the BepiColombo system which contains a level of abstraction followed by three levels of refinement followed by a model decomposition and two more refinement levels of one of the sub-models (published in "Formal Methods for Components and Objects" (FMCO) 2009 conference [1]). This development experience showed the benefits of using the atomicity decomposition approach with the model decomposition approach together. During development, the some-replicator construct and weak sequencing feature have been discovered and modelled in the Event-B model.

  Case study developments (Chapter 7), helped us to define some features which improved the atomicity decomposition approach. These features include the definitions of the most abstract level diagram, the combined atomicity decomposition diagram and multiple diagrams for a single root event. Different alternatives to model ordering in Event-B have been evaluated and the subset approach is chosen. The justification of choosing the subset approach has been presented (Section 4.4).

- (iii) As stated above, during the development of case studies, some new construct patterns and features were discovered. The discovered patterns and features were presented (Chapter 4). Each pattern was allocated to illustrate one constructor in a single refinement level. For each pattern the diagrammatic notation and the corresponding Event-B model have been described.

- (iv) We presented a formal and general description of the atomicity decomposition language (ADL) and translation rules to the Event-B model (Chapter 5). The ADL is described using Augmented Backus-Naur Form (ABNF) and includes the semantics to present the general combination of constructors in one or more refinement levels. Translation rules were presented per construct in a modular way.

- (v) We developed a plug-in that supports the atomicity decomposition approach in the Event-B toolkit, Rodin, (Chapter 6). The developed tool helps the automatic generation of the Event-B model from a graphical environment, which can results in making the modelling process of complex systems more manageable in Event-B.

Based on our experience - specifying, modelling and proving the media channel system and the BepiColombo system using the Event-B notation in the Rodin toolkit - the most difficult part in mathematical modelling using Rodin is dealing with complex and large models. Building the large model of media channel and BepiColombo showed how the atomicity decomposition approach can facilitate the use of the Event-B notation. The atomicity decomposition approach provides a graphical notation to explicitly illustrate the refinement structure in Event-B. The ordering between events are explicitly shown in the atomicity decomposition diagram. Different constructors of atomicity decomposition graphical notation have been discovered and presented. However, still some difficulties in building large and complex models are a notable barrier when encouraging developers to build mathematical models of their systems before implementing them. In summary it is hoped that the atomicity decomposition approach makes it convenient to model complicated systems using the Rodin toolkit.

The multiple instance (MI) style (Section 4.2.2) is applied to the case studies presented in Chapter 7. The single instance (SI) style is applied to a Controller Area Network (CAN) bus case study [81, 82]. Also a SI case example from [24] is addressed in Chapter 3.

9.2 Future Works

The work described in the thesis leaves open some opportunities for improvement. We list the future works as follows:

- Developing a graphical environment for the atomicity decomposition plug-in.

  To develop this graphical environment, the Eclipse Graphical modelling Framework (GMF) [83] and EuGENia [84] tool can be considered as two useful technologies. The Eclipse Graphical Modelling Framework (GMF) provides a generative component (GMF Tooling) and runtime infrastructure (GMF Runtime) for developing graphical editors based on EMF. EuGENia is a tool that automatically generates the models needed to implement a GMF editor from an EMF meta-model.

- Improving tool support by developing all translation rules.

  As described in Chapter 6, some of the defined translation rules, presented in Chapter 5, are not included in the current plug-in. The plug-in can be improved by providing all translation rules.

- Identifying other potential atomicity decomposition constructors.

  We believe that there can be other potential constructs for the atomicity decomposition approach. These constructs can be identified by developing more industrial and complex case studies. After identifying the potential constructors, they need to be defined as patterns and included as a part of the ADL and translation rules.

Appendix A

The Event-B Model of the Media Channel System

A.1 Abstract Specification

A.1.1 Context: *C1*

CONTEXT C1

SETS

ENDPOINT, MEDIUM, CODEC, MEDIACHANNEL, DIRECTION

CONSTANTS

ItoA, AtoI, medium, initiator, acceptor, direction

AXIOMS

axm1 : $partition(DIRECTION, \{ItoA\}, \{AtoI\})$

axm2 : $medium \in MEDIACHANNEL \rightarrow MEDIUM$

axm3 : $initiator \in MEDIACHANNEL \rightarrow ENDPOINT$

axm4 : $acceptor \in MEDIACHANNEL \rightarrow ENDPOINT$

axm5 : $direction \in MEDIACHANNEL \rightarrow DIRECTION$

END

A.1.2 Machine: *M0*

MACHINE M0

SEES C1

VARIABLES

establishMediaChannel, close, code \\ manually

INVARIANTS

inv_establishMediaChannel : $establishMediaChannel \subseteq MEDIACHANNEL$

inv_close_seq : $close \subseteq establishMediaChannel$

inv1 : $codec \in establishMediaChannel \rightarrow CODEC$ \\ manually

EVENTS

Initialisation

begin

act_establishMediaChannel : $establishMediaChannel := \varnothing$

```
        act_close :  $close := \varnothing$
        act1 :  $codec := \varnothing$  \\ manually
    end
Event   $establishMediaChannel \mathrel{\widehat{=}}$
    any
        ch, c \\ manually
    where
        grd_establishMediaChannel :  $ch \notin establishMediaChannel$
        grd1 :  $c \in CODEC$  \\ manually
    then
        act_establishMediaChannel :  $establishMediaChannel := establishMediaChannel \cup \{ch\}$
        act1 :  $codec(ch) := c$  \\ manually
    end
Event   $modify \mathrel{\widehat{=}}$
    any
        ch, c
    where
        grd_modify_seq :  $ch \in establishMediaChannel$
        grd_modify_loop :  $ch \notin close$
        grd1 :  $c \in CODEC$  \\ manually
    then
        act1 :  $codec(ch) := c$  \\ manually
    end
Event   $close \mathrel{\widehat{=}}$
    any
        ch
    where
        grd_close_seq :  $ch \in establishMediaChannel$
        grd_close :  $ch \notin close$
    then
        act_close :  $close := close \cup \{ch\}$
    end
END
```

A.2 1st Refinement

A.2.1 Context: *C2*

```
CONTEXT   C2
EXTENDS   C1
SETS
    PORT, IP
CONSTANTS
    endpointIp
AXIOMS
    axm1 :  $endpointIp \in ENDPOINT \rightarrowtail IP$
END
```

A.2.2 Machine: M1

```
MACHINE   M1
REFINES   M0
SEES   C2
```

VARIABLES

openWithRealCodecs, openAckWithoutCodecs, selectAndEstablishbyAcceptor, openWithoutCodecs, openAckWithRealCodecs, selectAndEstablishbyInitiator, close, codec \\ manually, initiatorPort \\ manually, acceptorPort \\ manually, codecList \\ manually

INVARIANTS

inv_openWithRealCodecs : $openWithRealCodecs \subseteq MEDIACHANNEL$

inv_openAckWithoutCodecs_seq : $openAckWithoutCodecs \subseteq openWithRealCodecs$

inv_selectAndEstablishbyAcceptor_seq : $selectAndEstablishbyAcceptor \subseteq openAckWithoutCodecs$

inv_openWithoutCodecs : $openWithoutCodecs \subseteq MEDIACHANNEL$

inv_openAckWithRealCodecs_seq : $openAckWithRealCodecs \subseteq openWithoutCodecs$

inv_selectAndEstablishbyInitiator_seq : $selectAndEstablishbyInitiator \subseteq openAckWithRealCodecs$

inv_close_seq : $close \subseteq selectAndEstablishbyAcceptor \cup selectAndEstablishbyInitiator$

inv_gluing : $selectAndEstablishbyAcceptor \cup selectAndEstablishbyInitiator = establishMediaChannel$

inv1 : $initiatorPort \in (openWithRealCodecs \cup openWithoutCodecs) \rightarrow PORT$ \\ manually

inv2 : $acceptorPort \in (openAckWithoutCodecs \cup openAckWithRealCodecs) \rightarrow PORT$ \\ manually

inv3 : $codecList \in (openWithRealCodecs \cup openAckWithRealCodecs) \rightarrow \mathbb{P}(CODEC)$ \\ manually

inv5 : $openWithRealCodecs \subseteq dom(direction \rhd \{AtoI\})$ \\ manually

inv6 : $openWithoutCodecs \subseteq dom(direction \rhd \{ItoA\})$ \\ manually

inv7 : $openWithRealCodecs \cap openWithoutCodecs = \varnothing$
\\ manually, derived from inv5, inv6, added to prove (SelectAndEstablishby.../GRD)

EVENTS

Initialisation

begin

act_openWithRealCodecs : $openWithRealCodecs := \varnothing$

act_openAckWithoutCodecs : $openAckWithoutCodecs := \varnothing$

act_selectAndEstablishbyAcceptor : $selectAndEstablishbyAcceptor := \varnothing$

act_openWithoutCodecs : $openWithoutCodecs := \varnothing$

act_openAckWithRealCodecs : $openAckWithRealCodecs := \varnothing$

act_selectAndEstablishbyInitiator : $selectAndEstablishbyInitiator := \varnothing$

act_close : $close := \varnothing$

act1 : $codec := \varnothing$ \\ manually

act2 : $initiatorPort := \varnothing$ \\ manually

act3 : $acceptorPort := \varnothing$ \\ manually

act4 : $codecList := \varnothing$ \\ manually

end

Event $openWithRealCodecs \mathrel{\widehat{=}}$

any

ch, cl \\ manually, p \\ manually, i \\ manually

where

grd_openWithRealCodecs : $ch \notin openWithRealCodecs$

grd1 : $ch \notin openWithoutCodecs$
\\ manually, derived from direction(ch) = AtoI, add to prove (inv7/INV)

grd2 : $cl \subseteq CODEC$ \\ manually

grd3 : $cl \neq \varnothing$ \\ manually

grd4 : $p \in PORT$ \\ manually

grd5 : $i \in IP$ \\ manually

grd6 : $i \in dom(endpointIp^{-1})$ \\ manually, WD

grd7 : $initiator(ch) = endpointIp^{-1}(i)$ \\ manually

grd8 : $direction(ch) = AtoI$ \\ manually

then

act_openWithRealCodecs : $openWithRealCodecs := openWithRealCodecs \cup \{ch\}$

act1 : $codecList(ch) := cl$ \\ manually

act2 : $initiatorPort(ch) := p$ \\ manually

end

Event $openAckWithoutCodecs \mathrel{\widehat{=}}$

```
    any
        ch, cl \\ manually, p \\ manually, i \\ manually, t
        \\ manually, to prove (openAckWithoutCodecs/inv5/INV) in M2
    where
        grd_openAckWithoutCodecs_seq : $ch \in openWithRealCodecs$
        grd_openAckWithoutCodecs : $ch \notin openAckWithoutCodecs$
        grd1 : $cl \subseteq CODEC$  \\ manually
        grd2 : $cl = \varnothing$  \\ manually
        grd3 : $p \in PORT$  \\ manually
        grd4 : $i \in IP$  \\ manually
        grd5 : $i \in dom(endpointIp^{-1})$  \\ manually, WD
        grd6 : $acceptor(ch) = endpointIp^{-1}(i)$  \\ manually
        grd7 : $t = codecList(ch)$
            \\ manually, to prove (openAckWithoutCodecs/inv5/INV) in M2
    then
        act_openAckWithoutCodecs : $openAckWithoutCodecs := openAckWithoutCodecs \cup \{ch\}$
        act1 : $acceptorPort(ch) := p$  \\ manually
        act2 : $codecList(ch) := t$
            \\ manually, to prove (openAckWithoutCodecs/inv5/INV) in M2
    end
Event  selectAndEstablishbyAcceptor $\widehat{=}$
refines  establishMediaChannel
    any
        ch, c \\ manually
    where
        grd_selectAndEstablishbyAcceptor_seq : $ch \in openAckWithoutCodecs$
        grd_selectAndEstablishbyAcceptor : $ch \notin selectAndEstablishbyAcceptor$
        grd1 : $c \in codecList(ch)$  \\ manually
    then
        act_selectAndEstablishbyAcceptor : $selectAndEstablishbyAcceptor :=$
            $selectAndEstablishbyAcceptor \cup \{ch\}$
        act1 : $codec(ch) := c$  \\ manually
    end
Event  openWithoutCodecs $\widehat{=}$
    any
        ch, cl \\ manually, p \\ manually, i \\ manually
    where
        grd_openWithoutCodecs : $ch \notin openWithoutCodecs$
        grd9 : $ch \notin openWithRealCodecs$
            \\ manually, derived from direction(ch) = ItoA, add to prove (inv7/INV)
        grd2 : $cl \subseteq CODEC$  \\ manually
        grd3 : $cl = \varnothing$  \\ manually
        grd4 : $p \in PORT$  \\ manually
        grd5 : $i \in IP$  \\ manually
        grd6 : $i \in dom(endpointIp^{-1})$  \\ manually, WD
        grd7 : $initiator(ch) = endpointIp^{-1}(i)$  \\ manually
        grd8 : $direction(ch) = ItoA$  \\ manually
    then
        act_openWithoutCodecs : $openWithoutCodecs := openWithoutCodecs \cup \{ch\}$
        act1 : $initiatorPort(ch) := p$  \\ manually
    end
Event  openAckWithRealCodecs $\widehat{=}$
    any
        ch, cl \\ manually, p \\ manually, i \\ manually
    where
        grd_openAckWithRealCodecs_seq : $ch \in openWithoutCodecs$
        grd_openAckWithRealCodecs : $ch \notin openAckWithRealCodecs$
        grd1 : $cl \subseteq CODEC$  \\ manually
        grd2 : $cl \neq \varnothing$  \\ manually
        grd3 : $p \in PORT$  \\ manually
        grd4 : $i \in IP$  \\ manually
        grd5 : $i \in dom(endpointIp^{-1})$  \\ manually, WD
        grd6 : $acceptor(ch) = endpointIp^{-1}(i)$  \\ manually
    then
        act_openAckWithRealCodecs : $openAckWithRealCodecs := openAckWithRealCodecs \cup$
            $\{ch\}$
        act1 : $codecList(ch) := cl$  \\ manually
        act2 : $acceptorPort(ch) := p$  \\ manually
    end
Event  $selectAndEstablishbyInitiator \mathrel{\widehat{=}}$
refines  $establishMediaChannel$
    any
        ch, c \\ manually
    where
        grd_selectAndEstablishbyInitiator_seq : $ch \in openAckWithRealCodecs$
        grd_selectAndEstablishbyInitiator : $ch \notin selectAndEstablishbyInitiator$
        grd1 : $c \in codecList(ch)$  \\ manually
    then
        act_selectAndEstablishbyInitiator : $selectAndEstablishbyInitiator :=$
            $selectAndEstablishbyInitiator \cup \{ch\}$
        act1 : $codec(ch) := c$  \\ manually
    end
Event  $modify \mathrel{\widehat{=}}$
refines  $modify$
    any
        ch, c
    where
        grd_modify_sequencing : $ch \in selectAndEstablishbyAcceptor \cup$
            $selectAndEstablishbyInitiator$
        grd_modify_loop : $ch \notin close$
        grd1 : $c \in CODEC$  \\ manually
    then
        act1 : $codec(ch) := c$  \\ manually
    end
Event  $close \mathrel{\widehat{=}}$
refines  $close$
    any
        $ch$
    where
        grd_close_seq : $ch \in selectAndEstablishbyAcceptor \cup selectAndEstablishbyInitiator$
        grd_close : $ch \notin close$
    then
        act_close : $close := close \cup \{ch\}$
    end
END
```

A.3 2nd Refinement

A.3.1 Machine: M2

```
MACHINE  M2
REFINES  M1
SEES  C2
```

VARIABLES

openWithRealCodecs, openAckWithoutCodecs, selectAndEstablishbyAcceptor, openWithoutCodecs, openAckWithRealCodecs, selectAndEstablishbyInitiator, modifyCodecListByDescriptor, respondBySelectorToCodec, modifyInitiatorPortByDescriptor, respondBySelectorToInitiatorPort, modifyAcceptorPortByDescriptor, respondBySelectorToAcceptorPort, close, codec \\ manually, initiatorPort2 \\ manually, acceptorPort2 \\ manually, codecList2 \\ manually

INVARIANTS

`inv_modifyCodecByDescriptor_seq` : $modifyCodecListByDescriptor \subseteq selectAndEstablishbyAcceptor \cup selectAndEstablishbyInitiator$

`inv_respondBySelectortoCodec_seq` : $respondBySelectorToCodec \subseteq modifyCodecListByDescriptor$

`inv_modifyInitiatorPortByDescriptor_seq` : $modifyInitiatorPortByDescriptor \subseteq selectAndEstablishbyAcceptor \cup selectAndEstablishbyInitiator$

`inv_respondBySelectorToInitiatorPort_seq` : $respondBySelectorToInitiatorPort \subseteq modifyInitiatorPortByDescriptor$

`inv_modifyAcceptorPortByDescriptor_seq` : $modifyAcceptorPortByDescriptor \subseteq selectAndEstablishbyAcceptor \cup selectAndEstablishbyInitiator$

`inv_respondBySelectorToAcceptorPort_seq` : $respondBySelectorToAcceptorPort \subseteq modifyAcceptorPortByDescriptor$

`inv1` : $initiatorPort2 \in (openWithRealCodecs \cup openWithoutCodecs) \rightarrow PORT$
\\ manually, to prove (EQL)

`inv2` : $acceptorPort2 \in (openAckWithoutCodecs \cup openAckWithRealCodecs) \rightarrow PORT$ \\ manually

`inv3` : $codecList2 \in (openWithRealCodecs \cup openAckWithRealCodecs) \rightarrow \mathbb{P}(CODEC)$ \\ manually

`inv4` : $\forall ch\cdot$
$(ch \in openAckWithRealCodecs \land$
$ch \notin selectAndEstablishbyInitiator$
$\Rightarrow$
$codecList2(ch) = codecList(ch))$
\\ manually, to prove (selectAndEstablishbyInitiator/grd1/GRD)

`inv5` : $\forall ch\cdot$
$(ch \in openAckWithoutCodecs \land$
$ch \notin selectAndEstablishbyAcceptor$
$\Rightarrow$
$codecList2(ch) = codecList(ch))$
\\ manually, to prove (selectAndEstablishbyAcceptor/grd1/GRD)

`inv6` : $\forall ch\cdot$
$(ch \in openWithRealCodecs \land$
$ch \notin openAckWithoutCodecs$
$\Rightarrow$
$codecList2(ch) = codecList(ch))$
\\ manually, to prove (openAckWithoutCodecs/grd7/GRD)

EVENTS

Initialisation

begin

`act_openWithRealCodecs` : $openWithRealCodecs := \varnothing$
`act_openAckWithoutCodecs` : $openAckWithoutCodecs := \varnothing$
`act_selectAndEstablishbyAcceptor` : $selectAndEstablishbyAcceptor := \varnothing$
`act_openWithoutCodecs` : $openWithoutCodecs := \varnothing$
`act_openAckWithRealCodecs` : $openAckWithRealCodecs := \varnothing$
`act_selectAndEstablishbyInitiator` : $selectAndEstablishbyInitiator := \varnothing$
`act_modifyCodecListByDescriptor` : $modifyCodecListByDescriptor := \varnothing$
`act_respondBySelectorToCodec` : $respondBySelectorToCodec := \varnothing$
`act_modifyInitiatorPortByDescriptor` : $modifyInitiatorPortByDescriptor := \varnothing$
`act_respondBySelectorToInitiatorPort` : $respondBySelectorToInitiatorPort := \varnothing$
`act_modifyAcceptorPortByDescriptor` : $modifyAcceptorPortByDescriptor := \varnothing$
`act_respondBySelectorToAcceptorPort` : $respondBySelectorToAcceptorPort := \varnothing$
`act_close` : $close := \varnothing$

```
        act1 : $codec := \varnothing$  \\ manually
        act2 : $initiatorPort2 := \varnothing$  \\ manually
        act3 : $acceptorPort2 := \varnothing$  \\ manually
        act4 : $codecList2 := \varnothing$  \\ manually
    end
Event   $openWithRealCodecs \mathrel{\widehat{=}}$
refines $openWithRealCodecs$
    any
        $ch$
        $cl$     manually
        $p$      manually
        $i$      manually
    where
        grd_openWithRealCodecs : $ch \notin openWithRealCodecs$
        grd1 : $ch \notin openWithoutCodecs$
            \\ manually, derived from direction(ch) = AtoI, add to prove (inv7/INV)
        grd2 : $cl \subseteq CODEC$  \\ manually
        grd3 : $cl \neq \varnothing$  \\ manually
        grd4 : $p \in PORT$  \\ manually
        grd5 : $i \in IP$  \\ manually
        grd6 : $i \in dom(endpointIp^{-1})$  \\ manually, WD
        grd7 : $initiator(ch) = endpointIp^{-1}(i)$  \\ manually
        grd8 : $direction(ch) = AtoI$  \\ manually
    then
        act_openWithRealCodecs : $openWithRealCodecs := openWithRealCodecs \cup \{ch\}$
        act1 : $codecList2(ch) := cl$  \\ manually
        act2 : $initiatorPort2(ch) := p$  \\ manually
    end
Event   $openAckWithoutCodecs \mathrel{\widehat{=}}$
refines $openAckWithoutCodecs$
    any
        ch, cl \\ manually, p \\ manually, i \\ manually, t
        \\ manually, to prove (openAckWithoutCodecs/inv5/INV)
    where
        grd_openAckWithoutCodecs_seq : $ch \in openWithRealCodecs$
        grd_openAckWithoutCodecs : $ch \notin openAckWithoutCodecs$
        grd1 : $cl \subseteq CODEC$  \\ manually
        grd2 : $cl = \varnothing$  \\ manually
        grd3 : $p \in PORT$  \\ manually
        grd4 : $i \in IP$  \\ manually
        grd5 : $i \in dom(endpointIp^{-1})$  \\ manually, WD
        grd6 : $acceptor(ch) = endpointIp^{-1}(i)$  \\ manually
        grd7 : $t = codecList2(ch)$
            \\ manually, to prove (openAckWithoutCodecs/inv5/INV)
    then
        act_openAckWithoutCodecs : $openAckWithoutCodecs := openAckWithoutCodecs \cup \{ch\}$
        act1 : $acceptorPort2(ch) := p$  \\ manually
        act2 : $codecList2(ch) := t$  \\ manually, to prove (openAckWithoutCodecs/inv5/INV)
    end
Event   $selectAndEstablishbyAcceptor \mathrel{\widehat{=}}$
refines $selectAndEstablishbyAcceptor$
    any
        ch, c \\ manually
    where
        grd_selectAndEstablishbyAcceptor_seq : $ch \in openAckWithoutCodecs$
        grd_selectAndEstablishbyAcceptor : $ch \notin selectAndEstablishbyAcceptor$
        grd1 : $c \in codecList2(ch)$  \\ manually
    then
```

```
        act_selectAndEstablishbyAcceptor : $selectAndEstablishbyAcceptor :=$
            $selectAndEstablishbyAcceptor \cup \{ch\}$
        act1 : $codec(ch) := c$  \\ manually
    end
Event  $openWithoutCodecs \;\widehat{=}$
refines  $openWithoutCodecs$
    any
        ch, cl \\ manually, p \\ manually, i \\ manually
    where
        grd_openWithoutCodecs : $ch \notin openWithoutCodecs$
        grd9 : $ch \notin openWithRealCodecs$
            \\ manually, derived from direction(ch) = ItoA, add to prove (inv7/INV)
        grd2 : $cl \subseteq CODEC$  \\ manually
        grd3 : $cl = \varnothing$  \\ manually
        grd4 : $p \in PORT$  \\ manually
        grd5 : $i \in IP$  \\ manually
        grd6 : $i \in dom(endpointIp^{-1})$  \\ manually, WD
        grd7 : $initiator(ch) = endpointIp^{-1}(i)$  \\ manually
        grd8 : $direction(ch) = ItoA$  \\ manually
    then
        act_openWithoutCodecs : $openWithoutCodecs := openWithoutCodecs \cup \{ch\}$
        act1 : $initiatorPort2(ch) := p$  \\ manually
    end
Event  $openAckWithRealCodecs \;\widehat{=}$
refines  $openAckWithRealCodecs$
    any
        ch, cl \\ manually, p \\ manually, i \\ manually
    where
        grd_openAckWithRealCodecs_seq : $ch \in openWithoutCodecs$
        grd_openAckWithRealCodecs : $ch \notin openAckWithRealCodecs$
        grd1 : $cl \subseteq CODEC$  \\ manually
        grd2 : $cl \neq \varnothing$  \\ manually
        grd3 : $p \in PORT$  \\ manually
        grd4 : $i \in IP$  \\ manually
        grd5 : $i \in dom(endpointIp^{-1})$  \\ manually, WD
        grd6 : $acceptor(ch) = endpointIp^{-1}(i)$  \\ manually
    then
        act_openAckWithRealCodecs : $openAckWithRealCodecs := openAckWithRealCodecs \cup \{ch\}$
        act1 : $codecList2(ch) := cl$  \\ manually
        act2 : $acceptorPort2(ch) := p$  \\ manually
    end
Event  $selectAndEstablishbyInitiator \;\widehat{=}$
refines  $selectAndEstablishbyInitiator$
    any
        ch, c \\ manually
    where
        grd_selectAndEstablishbyInitiator_seq : $ch \in openAckWithRealCodecs$
        grd_selectAndEstablishbyInitiator : $ch \notin selectAndEstablishbyInitiator$
        grd1 : $c \in codecList2(ch)$  \\ manually
    then
        act_selectAndEstablishbyInitiator : $selectAndEstablishbyInitiator :=$
            $selectAndEstablishbyInitiator \cup \{ch\}$
        act1 : $codec(ch) := c$  \\ manually
    end
Event  $modifyCodecBySelector \;\widehat{=}$
refines  $modify$
    any
        ch, c \\ manually
    where
        grd_modifyCodecBySelector_seq : $ch \in$
            $selectAndEstablishbyAcceptor \cup selectAndEstablishbyInitiator$
        grd_modifyCodecBySelector_loop : $ch \notin close$
        grd1 : $c \in codecList2(ch)$ \\ manually
    then
        act1 : $codec(ch) := c$ \\ manually
    end
Event $modifyCodecListByDescriptor \widehat{=}$
    any
        ch, cl \\ manually
    where
        grd_modifyCodecListByDescriptor_seq : $ch \in$
            $selectAndEstablishbyAcceptor \cup selectAndEstablishbyInitiator$
        grd_modifyCodecListByDescriptor : $ch \notin modifyCodecListByDescriptor$
        grd_modifyCodecListByDescriptor_loop : $ch \notin close$
        grd1 : $cl \subseteq CODEC$ \\ manually
        grd2 : $cl \neq \varnothing$ \\ manually
    then
        act_modifyCodecListByDescriptor : $modifyCodecListByDescriptor :=$
            $modifyCodecListByDescriptor \cup \{ch\}$
        act1 : $codecList2(ch) := cl$ \\ manually
    end
Event $respondBySelectorToCodec \widehat{=}$
refines $modify$
    any
        ch, c \\ manually
    where
        grd_respondBySelectorToCodec_seq : $ch \in modifyCodecListByDescriptor$
        grd_respondBySelectorToCodec : $ch \notin respondBySelectorToCodec$
        grd1 : $c \in codecList2(ch)$ \\ manually
        grd2 : $ch \notin close$ \\ manually, to prove (respondBySelectorToCodec/GRD)
    then
        act_respondBySelectorToCodec : $respondBySelectorToCodec := respondBySelectorToCodec \cup \{ch\}$
        act2 : $codec(ch) := c$ \\ manually
    end
Event $modify\_Loop\_Reset1 \widehat{=}$
    any
        $ch$
    where
        grd_reset : $ch \in respondBySelectorToCodec$
    then
        act_reset_modifyCodecListByDescriptor : $modifyCodecListByDescriptor :=$
            $modifyCodecListByDescriptor \setminus \{ch\}$
        act_reset_respondBySelectorToCodec : $respondBySelectorToCodec :=$
            $respondBySelectorToCodec \setminus \{ch\}$
    end
Event $modifyInitiatorPortByDescriptor \widehat{=}$
refines $modify$
    any
        ch, p \\ manually
    where
        grd_modifyInitiatorPortByDescriptor_seq : $ch \in$
            $selectAndEstablishbyAcceptor \cup selectAndEstablishbyInitiator$
        grd_modifyInitiatorPortByDescriptor : $ch \notin modifyInitiatorPortByDescriptor$
        grd_modifyInitiatorPortByDescriptor_loop : $ch \notin close$
        grd1 : $p \neq initiatorPort2(ch)$ \\ manually
    with
```

```
        c : $c = codec(ch)$  \\ manually
    then
        act_modifyInitiatorPortByDescriptor : $modifyInitiatorPortByDescriptor :=$
            $modifyInitiatorPortByDescriptor \cup \{ch\}$
        act1 : $initiatorPort2(ch) := p$  \\ manually
    end
Event  $respondBySelectorToInitiatorPort \;\widehat{=}$
    any
        $ch$
    where
        grd_respondBySelectorToInitiatorPort_seq : $ch \in modifyInitiatorPortByDescriptor$
        grd_respondBySelectorToInitiatorPort : $ch \notin respondBySelectorToInitiatorPort$
    then
        act_respondBySelectorToInitiatorPort : $respondBySelectorToInitiatorPort :=$
            $respondBySelectorToInitiatorPort \cup \{ch\}$
    end
Event  $modify\_Loop\_Reset2 \;\widehat{=}$
    any
        $ch$
    where
        grd_reset : $ch \in respondBySelectorToInitiatorPort$
    then
        act_reset_modifyCodecListByDescriptor : $modifyInitiatorPortByDescriptor :=$
            $modifyInitiatorPortByDescriptor \setminus \{ch\}$
        act_reset_respondBySelectorToInitiatorPort : $respondBySelectorToInitiatorPort :=$
            $respondBySelectorToInitiatorPort \setminus \{ch\}$
    end
Event  $modifyAcceptorPortByDescriptor \;\widehat{=}$
refines  $modify$
    any
        ch, p \\ manually
    where
        grd_modifyAcceptorPortByDescriptor_seq : $ch \in$
            $selectAndEstablishbyAcceptor \cup selectAndEstablishbyInitiator$
        grd_modifyAcceptorPortByDescriptor : $ch \notin modifyAcceptorPortByDescriptor$
        grd_modifyAcceptorPortByDescriptor_loop : $ch \notin close$
        grd1 : $p \neq acceptorPort2(ch)$  \\ manually
    with
        c : $c = codec(ch)$  \\ manually
    then
        act_modifyAcceptorPortByDescriptor : $modifyAcceptorPortByDescriptor :=$
            $modifyAcceptorPortByDescriptor \cup \{ch\}$
        act1 : $acceptorPort2(ch) := p$  \\ manually
    end
Event  $respondBySelectorToAcceptorPort \;\widehat{=}$
    any
        $ch$
    where
        grd_respondBySelectorToAcceptorPort_seq : $ch \in modifyAcceptorPortByDescriptor$
        grd_respondBySelectorToAcceptorPort : $ch \notin respondBySelectorToAcceptorPort$
    then
        act_respondBySelectorToAcceptorPort : $respondBySelectorToAcceptorPort :=$
            $respondBySelectorToAcceptorPort \cup \{ch\}$
    end
Event  $modify\_Loop\_Reset3 \;\widehat{=}$
    any
        $ch$
```

where

grd_reset : $ch \in respondBySelectorToAcceptorPort$

then

act_reset_modifyAcceptorPortByDescriptor : $modifyAcceptorPortByDescriptor :=$ $modifyAcceptorPortByDescriptor \setminus \{ch\}$

act_reset_respondBySelectorToAcceptorPort : $respondBySelectorToAcceptorPort :=$ $respondBySelectorToAcceptorPort \setminus \{ch\}$

end

Event $close \;\widehat{=}$

extends $close$

any

ch

where

grd_close_seq : $\mathtt{ch} \in$ $\mathtt{selectAndEstablishbyAcceptor} \cup \mathtt{selectAndEstablishbyInitiator}$

grd_close : $\mathtt{ch} \notin \mathtt{close}$

then

act_close : $\mathtt{close} := \mathtt{close} \cup \{\mathtt{ch}\}$

end

END

A.4 3rd Refinement

A.4.1 Machine: M3

MACHINE M3

REFINES M2

SEES C2

VARIABLES

openWithRealCodecs, openAckWithoutCodecs, selectAndEstablishbyAcceptor, openWithoutCodecs, openAckWithRealCodecs, selectAndEstablishbyInitiator, modifyCodecListByDescriptor, respondBySelectorToCodec, modifyInitiatorPortByDescriptor, respondBySelectorToInitiatorPort, modifyAcceptorPortByDescriptor, respondBySelectorToAcceptorPort, closeRequest, closeAck, codec \\ manually, initiatorPort2 \\ manually, acceptorPort2 \\ manually, codecList2 \\ manually

INVARIANTS

inv_closeRequest_seq : $closeRequest \subseteq$ $selectAndEstablishbyAcceptor \cup selectAndEstablishbyInitiator$

inv_closeAck_seq : $closeAck \subseteq closeRequest$

inv_closeAck_gluing : $closeAck = close$

EVENTS

Initialisation

begin

act_openWithRealCodecs : $openWithRealCodecs := \varnothing$

act_openAckWithoutCodecs : $openAckWithoutCodecs := \varnothing$

act_selectAndEstablishbyAcceptor : $selectAndEstablishbyAcceptor := \varnothing$

act_openWithoutCodecs : $openWithoutCodecs := \varnothing$

act_openAckWithRealCodecs : $openAckWithRealCodecs := \varnothing$

act_selectAndEstablishbyInitiator : $selectAndEstablishbyInitiator := \varnothing$

act_modifyCodecListByDescriptor : $modifyCodecListByDescriptor := \varnothing$

act_respondBySelectorToCodec : $respondBySelectorToCodec := \varnothing$

act_modifyInitiatorPortByDescriptor : $modifyInitiatorPortByDescriptor := \varnothing$

act_respondBySelectorToInitiatorPort : $respondBySelectorToInitiatorPort := \varnothing$

act_modifyAcceptorPortByDescriptor : $modifyAcceptorPortByDescriptor := \varnothing$

```
        act_respondBySelectorToAcceptorPort : $respondBySelectorToAcceptorPort := \varnothing$
        act_closeRequest : $closeRequest := \varnothing$
        act_closeAck : $closeAck := \varnothing$
        act1 : $codec := \varnothing$  \\ manually
        act2 : $initiatorPort2 := \varnothing$  \\ manually
        act3 : $acceptorPort2 := \varnothing$  \\ manually
        act4 : $codecList2 := \varnothing$  \\ manually
    end
Event openWithRealCodecs $\widehat{=}$
extends openWithRealCodecs
    any
        ch, cl \\ manually, p \\ manually, i \\ manually
    where
        grd_openWithRealCodecs : $ch \notin openWithRealCodecs$
        grd1 : $ch \notin openWithoutCodecs$
            \\ manually, derived from direction(ch) = AtoI, add to prove (inv7/INV)
        grd2 : $cl \subseteq CODEC$  \\ manually
        grd3 : $cl \neq \varnothing$  \\ manually
        grd4 : $p \in PORT$  \\ manually
        grd5 : $i \in IP$  \\ manually
        grd6 : $i \in dom(endpointIp^{-1})$  \\ manually, WD
        grd7 : $initiator(ch) = endpointIp^{-1}(i)$  \\ manually
        grd8 : $direction(ch) = AtoI$  \\ manually
    then
        act_openWithRealCodecs : $openWithRealCodecs := openWithRealCodecs \cup \{ch\}$
        act1 : $codecList2(ch) := cl$  \\ manually
        act2 : $initiatorPort2(ch) := p$  \\ manually
    end
Event openAckWithoutCodecs $\widehat{=}$
extends openAckWithoutCodecs
    any
        ch, cl \\ manually, p \\ manually, i \\ manually, t
        \\ manually, to prove (openAckWithoutCodecs/inv5/INV)
    where
        grd_openAckWithoutCodecs_seq : $ch \in openWithRealCodecs$
        grd_openAckWithoutCodecs : $ch \notin openAckWithoutCodecs$
        grd1 : $cl \subseteq CODEC$  \\ manually
        grd2 : $cl = \varnothing$  \\ manually
        grd3 : $p \in PORT$  \\ manually
        grd4 : $i \in IP$  \\ manually
        grd5 : $i \in dom(endpointIp^{-1})$  \\ manually, WD
        grd6 : $acceptor(ch) = endpointIp^{-1}(i)$  \\ manually
        grd7 : $t = codecList2(ch)$  \\ manually, to prove (openAckWithoutCodecs/inv5/INV)
    then
        act_openAckWithoutCodecs : $openAckWithoutCodecs := openAckWithoutCodecs \cup \{ch\}$
        act1 : $acceptorPort2(ch) := p$  \\ manually
        act2 : $codecList2(ch) := t$  \\ manually, to prove (openAckWithoutCodecs/inv5/INV)
    end
Event selectAndEstablishbyAcceptor $\widehat{=}$
extends selectAndEstablishbyAcceptor
    any
        ch, c \\ manually
    where
        grd_selectAndEstablishbyAcceptor_seq : $ch \in openAckWithoutCodecs$
        grd_selectAndEstablishbyAcceptor : $ch \notin selectAndEstablishbyAcceptor$
        grd1 : $c \in codecList2(ch)$  \\ manually
    then
```

act_selectAndEstablishbyAcceptor : $\mathtt{selectAndEstablishbyAcceptor} :=$
$\mathtt{selectAndEstablishbyAcceptor} \cup \{\mathtt{ch}\}$
act1 : $\mathtt{codec(ch)} := \mathtt{c}$ \\ manually
end

Event *openWithoutCodecs* $\widehat{=}$
extends *openWithoutCodecs*
any
ch, cl \\ manually, p \\ manually, i \\ manually
where
grd_openWithoutCodecs : $\mathtt{ch} \notin \mathtt{openWithoutCodecs}$
grd9 : $\mathtt{ch} \notin \mathtt{openWithRealCodecs}$
\\ manually, derived from direction(ch) = ItoA, add to prove (inv7/INV)
grd2 : $\mathtt{cl} \subseteq \mathtt{CODEC}$ \\ manually
grd3 : $\mathtt{cl} = \varnothing$ \\ manually
grd4 : $\mathtt{p} \in \mathtt{PORT}$ \\ manually
grd5 : $\mathtt{i} \in \mathtt{IP}$ \\ manually
grd6 : $\mathtt{i} \in \mathtt{dom}(\mathtt{endpointIp}^{-1})$ \\ manually, WD
grd7 : $\mathtt{initiator(ch)} = \mathtt{endpointIp}^{-1}(\mathtt{i})$ \\ manually
grd8 : $\mathtt{direction(ch)} = \mathtt{ItoA}$ \\ manually
then
act_openWithoutCodecs : $\mathtt{openWithoutCodecs} := \mathtt{openWithoutCodecs} \cup \{\mathtt{ch}\}$
act1 : $\mathtt{initiatorPort2(ch)} := \mathtt{p}$ \\ manually
end

Event *openAckWithRealCodecs* $\widehat{=}$
extends *openAckWithRealCodecs*
any
ch, cl \\ manually, p \\ manually, i \\ manually
where
grd_openAckWithRealCodecs_seq : $\mathtt{ch} \in \mathtt{openWithoutCodecs}$
grd_openAckWithRealCodecs : $\mathtt{ch} \notin \mathtt{openAckWithRealCodecs}$
grd1 : $\mathtt{cl} \subseteq \mathtt{CODEC}$ \\ manually
grd2 : $\mathtt{cl} \neq \varnothing$ \\ manually
grd3 : $\mathtt{p} \in \mathtt{PORT}$ \\ manually
grd4 : $\mathtt{i} \in \mathtt{IP}$ \\ manually
grd5 : $\mathtt{i} \in \mathtt{dom}(\mathtt{endpointIp}^{-1})$ \\ manually, WD
grd6 : $\mathtt{acceptor(ch)} = \mathtt{endpointIp}^{-1}(\mathtt{i})$ \\ manually
then
act_openAckWithRealCodecs : $\mathtt{openAckWithRealCodecs} := \mathtt{openAckWithRealCodecs} \cup \{\mathtt{ch}\}$
act1 : $\mathtt{codecList2(ch)} := \mathtt{cl}$ \\ manually
act2 : $\mathtt{acceptorPort2(ch)} := \mathtt{p}$ \\ manually
end

Event *selectAndEstablishbyInitiator* $\widehat{=}$
extends *selectAndEstablishbyInitiator*
any
ch, c \\ manually
where
grd_selectAndEstablishbyInitiator_seq : $\mathtt{ch} \in \mathtt{openAckWithRealCodecs}$
grd_selectAndEstablishbyInitiator : $\mathtt{ch} \notin \mathtt{selectAndEstablishbyInitiator}$
grd1 : $\mathtt{c} \in \mathtt{codecList2(ch)}$ \\ manually
then
act_selectAndEstablishbyInitiator : $\mathtt{selectAndEstablishbyInitiator} :=$
$\mathtt{selectAndEstablishbyInitiator} \cup \{\mathtt{ch}\}$
act1 : $\mathtt{codec(ch)} := \mathtt{c}$ \\ manually
end

Event *modifyCodecBySelector* $\widehat{=}$
refines *modifyCodecBySelector*
any
ch, c \\ manually

```
    where
        grd_modifyCodecBySelector_seq : $ch \in$
            $selectAndEstablishbyAcceptor \cup selectAndEstablishbyInitiator$
        grd_modifyCodecBySelector_loop : $ch \notin closeRequest$
        grd1 : $c \in codecList2(ch)$  \\ manually
    then
        act1 : $codec(ch) := c$  \\ manually
    end
Event  modifyCodecListByDescriptor ≙
refines  modifyCodecListByDescriptor
    any
        ch, cl \\ manually
    where
        grd_modifyCodecListByDescriptor_seq : $ch \in$
            $selectAndEstablishbyAcceptor \cup selectAndEstablishbyInitiator$
        grd_modifyCodecListByDescriptor : $ch \notin modifyCodecListByDescriptor$
        grd_modifyCodecListByDescriptor_loop : $ch \notin closeRequest$
        grd1 : $cl \subseteq CODEC$  \\ manually
        grd2 : $cl \neq \varnothing$  \\ manually
    then
        act_modifyCodecListByDescriptor : $modifyCodecListByDescriptor :=$
            $modifyCodecListByDescriptor \cup \{ch\}$
        act1 : $codecList2(ch) := cl$  \\ manually
    end
Event  respondBySelectorToCodec ≙
refines  respondBySelectorToCodec
    any
        ch, c \\ manually
    where
        grd_respondBySelectorToCodec_seq : $ch \in modifyCodecListByDescriptor$
        grd_respondBySelectorToCodec : $ch \notin respondBySelectorToCodec$
        grd1 : $c \in codecList2(ch)$  \\ manually
        grd2 : $ch \notin closeRequest$  \\ manually, to prove (respondBySelectorToCodec/GRD)
    then
        act_respondBySelectorToCodec : $respondBySelectorToCodec := respondBySelectorToCodec \cup \{ch\}$
        act2 : $codec(ch) := c$  \\ manually
    end
Event  modify_Loop_Reset1 ≙
extends  modify_Loop_Reset1
    any
        ch
    where
        grd_reset : ch ∈ respondBySelectorToCodec
    then
        act_reset_modifyCodecListByDescriptor : modifyCodecListByDescriptor :=
            modifyCodecListByDescriptor \ {ch}
        act_reset_respondBySelectorToCodec : respondBySelectorToCodec :=
            respondBySelectorToCodec \ {ch}
    end
Event  modifyInitiatorPortByDescriptor ≙
refines  modifyInitiatorPortByDescriptor
    any
        ch, p \\ manually
    where
        grd_modifyInitiatorPortByDescriptor_seq : $ch \in$
            $selectAndEstablishbyAcceptor \cup selectAndEstablishbyInitiator$
        grd_modifyInitiatorPortByDescriptor : $ch \notin modifyInitiatorPortByDescriptor$
```

```
            grd_modifyInitiatorPortByDescriptor_loop :  $ch \notin closeRequest$
            grd1 :  $p \neq initiatorPort2(ch)$  \\ manually
      then
            act_modifyInitiatorPortByDescriptor :  $modifyInitiatorPortByDescriptor :=$
                  $modifyInitiatorPortByDescriptor \cup \{ch\}$
            act1 :  $initiatorPort2(ch) := p$  \\ manually
      end
Event   respondBySelectorToInitiatorPort ≙
extends  respondBySelectorToInitiatorPort
      any
            ch
      where
            grd_respondBySelectorToInitiatorPort_seq :  ch ∈ modifyInitiatorPortByDescriptor
            grd_respondBySelectorToInitiatorPort :  ch ∉ respondBySelectorToInitiatorPort
      then
            act_respondBySelectorToInitiatorPort :  respondBySelectorToInitiatorPort :=
                  respondBySelectorToInitiatorPort ∪ {ch}
      end
Event   modify_Loop_Reset2 ≙
extends  modify_Loop_Reset2
      any
            ch
      where
            grd_reset :  ch ∈ respondBySelectorToInitiatorPort
      then
            act_reset_modifyCodecListByDescriptor :  modifyInitiatorPortByDescriptor :=
                  modifyInitiatorPortByDescriptor \ {ch}
            act_reset_respondBySelectorToInitiatorPort :  respondBySelectorToInitiatorPort :=
                  respondBySelectorToInitiatorPort \ {ch}
      end
Event   modifyAcceptorPortByDescriptor ≙
refines  modifyAcceptorPortByDescriptor
      any
            ch, p \\ manually
      where
            grd_modifyAcceptorPortByDescriptor_seq :  $ch \in$
                  $selectAndEstablishbyAcceptor \cup selectAndEstablishbyInitiator$
            grd_modifyAcceptorPortByDescriptor :  $ch \notin modifyAcceptorPortByDescriptor$
            grd_modifyAcceptorPortByDescriptor_loop :  $ch \notin closeRequest$
            grd1 :  $p \neq acceptorPort2(ch)$  \\ manually
      then
            act_modifyAcceptorPortByDescriptor :  $modifyAcceptorPortByDescriptor :=$
                  $modifyAcceptorPortByDescriptor \cup \{ch\}$
            act1 :  $acceptorPort2(ch) := p$  \\ manually
      end
Event   respondBySelectorToAcceptorPort ≙
extends  respondBySelectorToAcceptorPort
      any
            ch
      where
            grd_respondBySelectorToAcceptorPort_seq :  ch ∈ modifyAcceptorPortByDescriptor
            grd_respondBySelectorToAcceptorPort :  ch ∉ respondBySelectorToAcceptorPort
      then
            act_respondBySelectorToAcceptorPort :  respondBySelectorToAcceptorPort :=
                  respondBySelectorToAcceptorPort ∪ {ch}
      end
Event   modify_Loop_Reset3 ≙
```

```
extends  modify_Loop_Reset3
      any
            ch
      where
            grd_reset :  ch ∈ respondBySelectorToAcceptorPort
      then
            act_reset_modifyAcceptorPortByDescriptor :  modifyAcceptorPortByDescriptor :=
                 modifyAcceptorPortByDescriptor \ {ch}
            act_reset_respondBySelectorToAcceptorPort :  respondBySelectorToAcceptorPort :=
                 respondBySelectorToAcceptorPort \ {ch}
      end
Event   closeRequest ≙
      any
            ch
      where
            grd_closeRequest_seq :  ch ∈ selectAndEstablishbyAcceptor ∪ selectAndEstablishbyInitiator
            grd_closeRequest :  ch ∉ closeRequest
      then
            act_closeRequest :  closeRequest := closeRequest ∪ {ch}
      end
Event   closeAck ≙
refines  close
      any
            ch
      where
            grd_closeAck_seq :  ch ∈ closeRequest
            grd_closeAck :  ch ∉ closeAck
      then
            act_closeAck :  closeAck := closeAck ∪ {ch}
      end
END
```

A.5 4th Refinement

A.5.1 Machine: *M4*

```
MACHINE   M4
REFINES   M3
SEES   C2
VARIABLES
      openWithRealCodecs, openAckWithoutCodecs, selectAndEstablishbyAcceptor, openWithoutCodecs, ope-
      nAckWithRealCodecs, selectAndEstablishbyInitiator, modifyCodecBySelector_withInitiator,
      modifyCodecBySelector_withAcceptor, modifyCodecListByDescriptor_withInitiator,
      modifyCodecListByDescriptor_withAcceptor, respondBySelectorToInitiatorCodec, respondBySelectorToAc-
      ceptorCodec, modifyInitiatorPortByDescriptor, respondBySelectorToInitiatorPort, modifyAcceptorPort-
      ByDescriptor, respondBySelectorToAcceptorPort, closeRequest, closeAck, codec \\ manually, initiator-
      Port2 \\ manually, acceptorPort2 \\ manually, codecList2 \\ manually
INVARIANTS
      inv_modifyCodecBySelector_withInitiator_seq :  modifyCodecBySelector_withInitiator ⊆
            selectAndEstablishbyAcceptor ∪ selectAndEstablishbyInitiator
      inv_modifyCodecBySelector_withAcceptor_seq :  modifyCodecBySelector_withAcceptor ⊆
            selectAndEstablishbyAcceptor ∪ selectAndEstablishbyInitiator
      inv_modifyCodecListByDescriptor_withInitiator_seq :  modifyCodecListByDescriptor_withInitiator ⊆
            selectAndEstablishbyAcceptor ∪ selectAndEstablishbyInitiator
```

inv_modifyCodecListByDescriptor_withAcceptor_seq : $modifyCodecListByDescriptor\_withAcceptor \subseteq selectAndEstablishbyAcceptor \cup selectAndEstablishbyInitiator$

inv_respondBySelectorToInitiatorCodec_seq : $respondBySelectorToInitiatorCodec \subseteq modifyCodecListByDescriptor\_withInitiator \cup modifyCodecListByDescriptor\_withAcceptor$

inv_respondBySelectorToAcceptorCodec_seq : $respondBySelectorToAcceptorCodec \subseteq modifyCodecListByDescriptor\_withInitiator \cup modifyCodecListByDescriptor\_withAcceptor$

inv_modifyCodecBySelector_xor_gluing : $partition(modifyCodecBySelector\_withInitiator \cup modifyCodecBySelector\_withAcceptor, modifyCodecBySelector\_withInitiator, modifyCodecBySelector\_withAcceptor)$

inv_modifyCodecListByDescriptor_xor_gluing : $partition(modifyCodecListByDescriptor, modifyCodecListByDescriptor\_withInitiator, modifyCodecListByDescriptor\_withAcceptor)$

inv_respondBySelectorToCodec_xor_gluing : $partition(respondBySelectorToCodec, respondBySelectorToInitiatorCodec, respondBySelectorToAcceptorCodec)$

EVENTS

Initialisation

begin

act_openWithRealCodecs : $openWithRealCodecs := \varnothing$

act_openAckWithoutCodecs : $openAckWithoutCodecs := \varnothing$

act_selectAndEstablishbyAcceptor : $selectAndEstablishbyAcceptor := \varnothing$

act_openWithoutCodecs : $openWithoutCodecs := \varnothing$

act_openAckWithRealCodecs : $openAckWithRealCodecs := \varnothing$

act_selectAndEstablishbyInitiator : $selectAndEstablishbyInitiator := \varnothing$

act_modifyCodecBySelector_withInitiator : $modifyCodecBySelector\_withInitiator := \varnothing$

act_modifyCodecBySelector_withAcceptor : $modifyCodecBySelector\_withAcceptor := \varnothing$

act_modifyCodecListByDescriptor_withInitiator : $modifyCodecListByDescriptor\_withInitiator := \varnothing$

act_modifyCodecListByDescriptor_withAcceptor : $modifyCodecListByDescriptor\_withAcceptor := \varnothing$

act_respondBySelectorToInitiatorCodec : $respondBySelectorToInitiatorCodec := \varnothing$

act_respondBySelectorToAcceptorCodec : $respondBySelectorToAcceptorCodec := \varnothing$

act_modifyInitiatorPortByDescriptor : $modifyInitiatorPortByDescriptor := \varnothing$

act_respondBySelectorToInitiatorPort : $respondBySelectorToInitiatorPort := \varnothing$

act_modifyAcceptorPortByDescriptor : $modifyAcceptorPortByDescriptor := \varnothing$

act_respondBySelectorToAcceptorPort : $respondBySelectorToAcceptorPort := \varnothing$

act_closeRequest : $closeRequest := \varnothing$

act_closeAck : $closeAck := \varnothing$

act1 : $codec := \varnothing$ \\ manually

act2 : $initiatorPort2 := \varnothing$ \\ manually

act3 : $acceptorPort2 := \varnothing$ \\ manually

act4 : $codecList2 := \varnothing$ \\ manually

end

Event $openWithRealCodecs \;\widehat{=}$

extends $openWithRealCodecs$

any

ch

cl manually

p manually

i manually

where

grd_openWithRealCodecs : ch $\notin$ openWithRealCodecs

grd1 : ch $\notin$ openWithoutCodecs

manually, derived from direction(ch) = AtoI, add to prove (inv7/INV)

grd2 : cl $\subseteq$ CODEC \\ manually

grd3 : cl $\neq \varnothing$ \\ manually

grd4 : p $\in$ PORT \\ manually

grd5 : i $\in$ IP \\ manually

grd6 : i $\in$ dom(endpointIp^{-1})

manually - WD

```
        grd7 :  initiator(ch) = endpointIp⁻¹(i)  \\ manually
        grd8 :  direction(ch) = AtoI  \\ manually
    then
        act_openWithRealCodecs :  openWithRealCodecs := openWithRealCodecs ∪ {ch}
        act1 :  codecList2(ch) := cl  \\ manually
        act2 :  initiatorPort2(ch) := p  \\ manually
    end
Event   openAckWithoutCodecs ≙
extends  openAckWithoutCodecs
    any
        ch
        cl     manually
        p      manually
        i      manually
        t      manually, to prove (openAckWithoutCodecs/inv5/INV)
    where
        grd_openAckWithoutCodecs_seq :  ch ∈ openWithRealCodecs
        grd_openAckWithoutCodecs :  ch ∉ openAckWithoutCodecs
        grd1 :  cl ⊆ CODEC  \\ manually
        grd2 :  cl = ∅  \\ manually
        grd3 :  p ∈ PORT  \\ manually
        grd4 :  i ∈ IP  \\ manually
        grd5 :  i ∈ dom(endpointIp⁻¹)
               manually - WD
        grd6 :  acceptor(ch) = endpointIp⁻¹(i)  \\ manually
        grd7 :  t = codecList2(ch)
               manually, to prove (openAckWithoutCodecs/inv5/INV)
    then
        act_openAckWithoutCodecs :  openAckWithoutCodecs := openAckWithoutCodecs ∪ {ch}
        act1 :  acceptorPort2(ch) := p  \\ manually
        act2 :  codecList2(ch) := t
               manually, to prove (openAckWithoutCodecs/inv5/INV)
    end
Event   selectAndEstablishbyAcceptor ≙
extends  selectAndEstablishbyAcceptor
    any
        ch
        c      manually
    where
        grd_selectAndEstablishbyAcceptor_seq :  ch ∈ openAckWithoutCodecs
        grd_selectAndEstablishbyAcceptor :  ch ∉ selectAndEstablishbyAcceptor
        grd1 :  c ∈ codecList2(ch)  \\ manually
    then
        act_selectAndEstablishbyAcceptor :  selectAndEstablishbyAcceptor :=
               selectAndEstablishbyAcceptor ∪ {ch}
        act1 :  codec(ch) := c  \\ manually
    end
Event   openWithoutCodecs ≙
extends  openWithoutCodecs
    any
        ch
        cl     manually
        p      manually
        i      manually
    where
        grd_openWithoutCodecs :  ch ∉ openWithoutCodecs
        grd9 :  ch ∉ openWithRealCodecs
               manually, derived from direction(ch) = ItoA, add to prove (inv7/INV)
```

```
            grd2 : cl ⊆ CODEC  \\ manually
            grd3 : cl = ∅  \\ manually
            grd4 : p ∈ PORT  \\ manually
            grd5 : i ∈ IP  \\ manually
            grd6 : i ∈ dom(endpointIp⁻¹)
                 manually - WD
            grd7 : initiator(ch) = endpointIp⁻¹(i)  \\ manually
            grd8 : direction(ch) = ItoA  \\ manually
      then
            act_openWithoutCodecs : openWithoutCodecs := openWithoutCodecs ∪ {ch}
            act1 : initiatorPort2(ch) := p  \\ manually
      end
Event   openAckWithRealCodecs ≙
extends  openAckWithRealCodecs
      any
            ch
            cl    manually
            p     manually
            i     manually
      where
            grd_openAckWithRealCodecs_seq : ch ∈ openWithoutCodecs
            grd_openAckWithRealCodecs : ch ∉ openAckWithRealCodecs
            grd1 : cl ⊆ CODEC  \\ manually
            grd2 : cl ≠ ∅  \\ manually
            grd3 : p ∈ PORT  \\ manually
            grd4 : i ∈ IP  \\ manually
            grd5 : i ∈ dom(endpointIp⁻¹)
                 manually - WD
            grd6 : acceptor(ch) = endpointIp⁻¹(i)  \\ manually
      then
            act_openAckWithRealCodecs : openAckWithRealCodecs := openAckWithRealCodecs ∪ {ch}
            act1 : codecList2(ch) := cl  \\ manually
            act2 : acceptorPort2(ch) := p  \\ manually
      end
Event   selectAndEstablishbyInitiator ≙
extends  selectAndEstablishbyInitiator
      any
            ch
            c     manually
      where
            grd_selectAndEstablishbyInitiator_seq : ch ∈ openAckWithRealCodecs
            grd_selectAndEstablishbyInitiator : ch ∉ selectAndEstablishbyInitiator
            grd1 : c ∈ codecList2(ch)  \\ manually
      then
            act_selectAndEstablishbyInitiator : selectAndEstablishbyInitiator :=
                 selectAndEstablishbyInitiator ∪ {ch}
            act1 : codec(ch) := c  \\ manually
      end
Event   modifyCodecBySelector_withInitiator ≙
refines  modifyCodecBySelector
      any
            ch
            c     manually
      where
            grd_modifyCodecBySelector_withInitiator_seq : ch ∈
                 selectAndEstablishbyAcceptor ∪ selectAndEstablishbyInitiator
            grd_modifyCodecBySelector_withInitiator : ch ∉ modifyCodecBySelector_withInitiator
```

```
        grd_modifyCodecBySelector_withInitiator_xor : $ch \notin modifyCodecBySelector\_withAcceptor$
        grd_modifyCodecBySelector_withInitiator_loop : $ch \notin closeRequest$
        grd1 : $c \in codecList2(ch)$ \\ manually
        grd2 : $direction(ch) = ItoA$ \\ manually
    then
        act_modifyCodecBySelector_withInitiator : $modifyCodecBySelector\_withInitiator :=$
            $modifyCodecBySelector\_withInitiator \cup \{ch\}$
        act1 : $codec(ch) := c$ \\ manually
    end
Event  modifyCodecBySelector_withAcceptor ≙
refines  modifyCodecBySelector
    any
        ch
        c      manually
    where
        grd_modifyCodecBySelector_withAcceptor_seq : $ch \in selectAndEstablishbyAcceptor \cup$
            $selectAndEstablishbyInitiator$
        grd_modifyCodecBySelector_withAcceptor : $ch \notin modifyCodecBySelector\_withAcceptor$
        grd_modifyCodecBySelector_withAcceptor_xor : $ch \notin modifyCodecBySelector\_withInitiator$
        grd_modifyCodecBySelector_withAcceptor_loop : $ch \notin closeRequest$
        grd1 : $c \in codecList2(ch)$ \\ manually
        grd2 : $direction(ch) = AtoI$ \\ manually
    then
        act_modifyCodecBySelector_withAcceptor : $modifyCodecBySelector\_withAcceptor :=$
            $modifyCodecBySelector\_withAcceptor \cup \{ch\}$
        act1 : $codec(ch) := c$ \\ manually
    end
Event  modify_Loop_Reset0 ≙
    any
        ch
    where
        grd_reset : $ch \in modifyCodecBySelector\_withInitiator \cup modifyCodecBySelector\_withAcceptor$
    then
        act_reset_modifyCodecListByDescriptor : $modifyCodecBySelector\_withInitiator :=$
            $modifyCodecBySelector\_withInitiator \setminus \{ch\}$
        act_reset_modifyCodecBySelector_withAcceptor : $modifyCodecBySelector\_withAcceptor :=$
            $modifyCodecBySelector\_withAcceptor \setminus \{ch\}$
    end
Event  modifyCodecListByDescriptor_withInitiator ≙
refines  modifyCodecListByDescriptor
    any
        ch
        cl
    where
        grd_modifyCodecListByDescriptor_withInitiator_seq : $ch \in$
            $selectAndEstablishbyAcceptor \cup selectAndEstablishbyInitiator$
        grd_modifyCodecListByDescriptor_withInitiator : $ch \notin$
            $modifyCodecListByDescriptor\_withInitiator$
        grd_modifyCodecListByDescriptor_withInitiator_xor : $ch \notin$
            $modifyCodecListByDescriptor\_withAcceptor$
        grd_modifyCodecListByDescriptor_withInitiator_loop : $ch \notin closeRequest$
        grd1 : $cl \subseteq CODEC$
        grd2 : $cl \neq \varnothing$
        grd3 : $direction(ch) = AtoI$
    then
        act_modifyCodecListByDescriptor_withInitiator : $modifyCodecListByDescriptor\_withInitiator$
            $:= modifyCodecListByDescriptor\_withInitiator \cup \{ch\}$
        act1 : $codecList2(ch) := cl$
    end
Event  modifyCodecListByDescriptor_withAcceptor ≙
refines  modifyCodecListByDescriptor
    any
        ch
        cl
    where
        grd_modifyCodecListByDescriptor_withAcceptor_seq : ch ∈
            selectAndEstablishbyAcceptor ∪ selectAndEstablishbyInitiator
        grd_modifyCodecListByDescriptor_withAcceptor : ch ∉
            modifyCodecListByDescriptor_withAcceptor
        grd_modifyCodecListByDescriptor_withAcceptor_xor : ch ∉
            modifyCodecListByDescriptor_withInitiator
        grd_modifyCodecListByDescriptor_withAcceptor_loop : ch ∉ closeRequest
        grd1 : cl ⊆ CODEC
        grd2 : cl ≠ ∅
        grd3 : direction(ch) = ItoA
    then
        act_modifyCodecListByDescriptor_withAcceptor : modifyCodecListByDescriptor_withAcceptor
            := modifyCodecListByDescriptor_withAcceptor ∪ {ch}
        act1 : codecList2(ch) := cl
    end
Event  respondBySelectorToInitiatorCodec ≙
refines  respondBySelectorToCodec
    any
        ch
        c
    where
        grd_respondBySelectorToInitiatorCodec_seq : ch ∈ modifyCodecListByDescriptor_withInitiator
            ∪ modifyCodecListByDescriptor_withAcceptor
        grd_respondBySelectorToInitiatorCodec : ch ∉ respondBySelectorToInitiatorCodec
        grd_respondBySelectorToInitiatorCodec_xor : ch ∉ respondBySelectorToAcceptorCodec
        grd1 : c ∈ codecList2(ch)
        grd2 : direction(ch) = AtoI
        grd3 : ch ∉ closeRequest
            manually, from M3 to prove GRD
    then
        act_respondBySelectortoInitiatorCodec : respondBySelectorToInitiatorCodec :=
            respondBySelectorToInitiatorCodec ∪ {ch}
        act1 : codec(ch) := c
    end
Event  respondBySelectorToAcceptorCodec ≙
refines  respondBySelectorToCodec
    any
        ch
        c
    where
        grd_respondBySelectorToAcceptorCodec_seq : ch ∈ modifyCodecListByDescriptor_withInitiator
            ∪ modifyCodecListByDescriptor_withAcceptor
        grd_respondBySelectorToAcceptorCodec : ch ∉ respondBySelectorToAcceptorCodec
        grd_respondBySelectorToAcceptorCodec_xor : ch ∉ respondBySelectorToInitiatorCodec
        grd1 : c ∈ codecList2(ch)
        grd2 : direction(ch) = ItoA
        grd3 : ch ∉ closeRequest
            manually, from M3 to prove GRD
    then
        act_respondBySelectortoAcceptorCodec : respondBySelectorToAcceptorCodec :=
            respondBySelectorToAcceptorCodec ∪ {ch}
        act1 :  $codec(ch) := c$
    end
Event  modify_Loop_Reset1 ≙
refines  modify_Loop_Reset1
    any
        ch
    where
        grd_reset :  $ch \in respondBySelectorToInitiatorCodec \cup respondBySelectorToAcceptorCodec$
    then
        act_reset_modifyCodecListByDescriptor_withInitiator :
            $modifyCodecListByDescriptor\_withInitiator := modifyCodecListByDescriptor\_withInitiator \setminus \{ch\}$
        act_reset_modifyCodecListByDescriptor_withAcceptor :
            $modifyCodecListByDescriptor\_withAcceptor := modifyCodecListByDescriptor\_withAcceptor \setminus \{ch\}$
        act_reset_respondBySelectorToInitiatorCodec :  $respondBySelectorToInitiatorCodec :=$
            $respondBySelectorToInitiatorCodec \setminus \{ch\}$
        act_reset_respondBySelectorToAcceptorCodec :  $respondBySelectorToAcceptorCodec :=$
            $respondBySelectorToAcceptorCodec \setminus \{ch\}$
    end
Event  modifyInitiatorPortByDescriptor ≙
extends  modifyInitiatorPortByDescriptor
    any
        ch
        p     manually
    where
        grd_modifyInitiatorPortByDescriptor_seq :  ch ∈
            selectAndEstablishbyAcceptor ∪ selectAndEstablishbyInitiator
        grd_modifyInitiatorPortByDescriptor :  ch ∉ modifyInitiatorPortByDescriptor
        grd_modifyInitiatorPortByDescriptor_loop :  ch ∉ closeRequest
        grd1 :  p ≠ initiatorPort2(ch)  \\ manually
    then
        act_modifyInitiatorPortByDescriptor :  modifyInitiatorPortByDescriptor :=
            modifyInitiatorPortByDescriptor ∪ {ch}
        act1 :  initiatorPort2(ch) := p  \\ manually
    end
Event  respondBySelectorToInitiatorPort ≙
extends  respondBySelectorToInitiatorPort
    any
        ch
    where
        grd_respondBySelectorToInitiatorPort_seq :  ch ∈ modifyInitiatorPortByDescriptor
        grd_respondBySelectorToInitiatorPort :  ch ∉ respondBySelectorToInitiatorPort
    then
        act_respondBySelectorToInitiatorPort :  respondBySelectorToInitiatorPort :=
            respondBySelectorToInitiatorPort ∪ {ch}
    end
Event  modify_Loop_Reset2 ≙
extends  modify_Loop_Reset2
    any
        ch
    where
        grd_reset :  ch ∈ respondBySelectorToInitiatorPort
    then
        act_reset_modifyCodecListByDescriptor :  modifyInitiatorPortByDescriptor :=
            modifyInitiatorPortByDescriptor \ {ch}
        act_reset_respondBySelectorToInitiatorPort : respondBySelectorToInitiatorPort :=
            respondBySelectorToInitiatorPort \ {ch}
    end
Event  modifyAcceptorPortByDescriptor ≙
extends  modifyAcceptorPortByDescriptor
    any
        ch
        p      manually
    where
        grd_modifyAcceptorPortByDescriptor_seq : ch ∈
            selectAndEstablishbyAcceptor ∪ selectAndEstablishbyInitiator
        grd_modifyAcceptorPortByDescriptor : ch ∉ modifyAcceptorPortByDescriptor
        grd_modifyAcceptorPortByDescriptor_loop : ch ∉ closeRequest
        grd1 : p ≠ acceptorPort2(ch)  \\ manually
    then
        act_modifyAcceptorPortByDescriptor : modifyAcceptorPortByDescriptor :=
            modifyAcceptorPortByDescriptor ∪ {ch}
        act1 : acceptorPort2(ch) := p  \\ manually
    end
Event  respondBySelectorToAcceptorPort ≙
extends  respondBySelectorToAcceptorPort
    any
        ch
    where
        grd_respondBySelectorToAcceptorPort_seq : ch ∈ modifyAcceptorPortByDescriptor
        grd_respondBySelectorToAcceptorPort : ch ∉ respondBySelectorToAcceptorPort
    then
        act_respondBySelectorToAcceptorPort : respondBySelectorToAcceptorPort :=
            respondBySelectorToAcceptorPort ∪ {ch}
    end
Event  modify_Loop_Reset3 ≙
extends  modify_Loop_Reset3
    any
        ch
    where
        grd_reset : ch ∈ respondBySelectorToAcceptorPort
    then
        act_reset_modifyAcceptorPortByDescriptor : modifyAcceptorPortByDescriptor :=
            modifyAcceptorPortByDescriptor \ {ch}
        act_reset_respondBySelectorToAcceptorPort : respondBySelectorToAcceptorPort :=
            respondBySelectorToAcceptorPort \ {ch}
    end
Event  closeRequest ≙
extends  closeRequest
    any
        ch
    where
        grd_closeRequest_seq : ch ∈ selectAndEstablishbyAcceptor ∪
            selectAndEstablishbyInitiator
        grd_closeRequest : ch ∉ closeRequest
    then
        act_closeRequest : closeRequest := closeRequest ∪ {ch}
    end
Event  closeAck ≙
extends  closeAck
    any
```

ch

where

grd_closeAck_seq : $\mathtt{ch} \in \mathtt{closeRequest}$

grd_closeAck : $\mathtt{ch} \notin \mathtt{closeAck}$

then

act_closeAck : $\mathtt{closeAck} := \mathtt{closeAck} \cup \{\mathtt{ch}\}$

end

END

A.6 5th Refinement

A.6.1 Machine: *M5*

MACHINE M5

REFINES M4

SEES C2

VARIABLES

openWithRealCodecs, openAckWithoutCodecs, selectAndEstablishbyAcceptor, openWithoutCodecs, openAckWithRealCodecs, selectAndEstablishbyInitiator, modifyCodecBySelector_withInitiator, modifyCodecBySelector_withAcceptor, modifyCodecListByDescriptor_withInitiator, modifyCodecListByDescriptor_withAcceptor, respondBySelectorToInitiatorCodec, respondBySelectorToAcceptorCodec, modifyInitiatorPortByDescriptor, respondBySelectorToInitiatorPort, modifyAcceptorPortByDescriptor, respondBySelectorToAcceptorPort, closeRequestAtoI, closeRequestItoA, closeAckAtoI, closeAckItoA, codec \\ manually, initiatorPort2 \\ manually, acceptorPort2 \\ manually, codecList2 \\ manually

INVARIANTS

inv_closeRequestAtoI_seq : $closeRequestAtoI \subseteq selectAndEstablishbyAcceptor \cup selectAndEstablishbyInitiator$

inv_closeRequestItoA_seq : $closeRequestItoA \subseteq selectAndEstablishbyAcceptor \cup selectAndEstablishbyInitiator$

inv_closeAckAtoI_seq : $closeAckAtoI \subseteq closeRequestAtoI \cup closeRequestItoA$

inv_closeAckItoA_seq : $closeAckItoA \subseteq closeRequestAtoI \cup closeRequestItoA$

inv_closeRequest_xor_gluing : $partition(closeRequest, closeRequestAtoI, closeRequestItoA)$

inv_closeAck_xor_gluing : $partition(closeAck, closeAckAtoI, closeAckItoA)$

EVENTS

Initialisation

begin

act_openWithRealCodecs : $openWithRealCodecs := \varnothing$

act_openAckWithoutCodecs : $openAckWithoutCodecs := \varnothing$

act_selectAndEstablishbyAcceptor : $selectAndEstablishbyAcceptor := \varnothing$

act_openWithoutCodecs : $openWithoutCodecs := \varnothing$

act_openAckWithRealCodecs : $openAckWithRealCodecs := \varnothing$

act_selectAndEstablishbyInitiator : $selectAndEstablishbyInitiator := \varnothing$

act_modifyCodecBySelector_withInitiator : $modifyCodecBySelector\_withInitiator := \varnothing$

act_modifyCodecBySelector_withAcceptor : $modifyCodecBySelector\_withAcceptor := \varnothing$

act_modifyCodecListByDescriptor_withInitiator : $modifyCodecListByDescriptor\_withInitiator := \varnothing$

act_modifyCodecListByDescriptor_withAcceptor : $modifyCodecListByDescriptor\_withAcceptor := \varnothing$

act_respondBySelectorToInitiatorCodec : $respondBySelectorToInitiatorCodec := \varnothing$

act_respondBySelectorToAcceptorCodec : $respondBySelectorToAcceptorCodec := \varnothing$

act_modifyInitiatorPortByDescriptor : $modifyInitiatorPortByDescriptor := \varnothing$

act_respondBySelectorToInitiatorPort : $respondBySelectorToInitiatorPort := \varnothing$

```
          act_modifyAcceptorPortByDescriptor : $modifyAcceptorPortByDescriptor := \varnothing$
          act_respondBySelectorToAcceptorPort : $respondBySelectorToAcceptorPort := \varnothing$
          act_closeAckAtoI : $closeAckAtoI := \varnothing$
          act_closeAckItoA : $closeAckItoA := \varnothing$
          act_closeRequestAtoI : $closeRequestAtoI := \varnothing$
          act_closeRequestItoA : $closeRequestItoA := \varnothing$
          act1 : $codec := \varnothing$  \\ manually
          act2 : $initiatorPort2 := \varnothing$  \\ manually
          act3 : $acceptorPort2 := \varnothing$  \\ manually
          act4 : $codecList2 := \varnothing$  \\ manually
     end
Event  openWithRealCodecs ≙
extends  openWithRealCodecs
     any
          ch, cl \\ manually, p \\ manually, i \\ manually
     where
          grd_openWithRealCodecs : $ch \notin openWithRealCodecs$
          grd1 : $ch \notin openWithoutCodecs$
               \\ manually, derived from direction(ch) = AtoI, add to prove (inv7/INV)
          grd2 : $cl \subseteq CODEC$  \\ manually
          grd3 : $cl \neq \varnothing$  \\ manually
          grd4 : $p \in PORT$  \\ manually
          grd5 : $i \in IP$  \\ manually
          grd6 : $i \in dom(endpointIp^{-1})$  \\ manually, WD
          grd7 : $initiator(ch) = endpointIp^{-1}(i)$  \\ manually
          grd8 : $direction(ch) = AtoI$  \\ manually
     then
          act_openWithRealCodecs : $openWithRealCodecs := openWithRealCodecs \cup \{ch\}$
          act1 : $codecList2(ch) := cl$  \\ manually
          act2 : $initiatorPort2(ch) := p$  \\ manually
     end
Event  openAckWithoutCodecs ≙
extends  openAckWithoutCodecs
     any
          ch, cl \\ manually, p \\ manually, i \\ manually, t
          \\ manually, to prove (openAckWithoutCodecs/inv5/INV)
     where
          grd_openAckWithoutCodecs_seq : $ch \in openWithRealCodecs$
          grd_openAckWithoutCodecs : $ch \notin openAckWithoutCodecs$
          grd1 : $cl \subseteq CODEC$  \\ manually
          grd2 : $cl = \varnothing$  \\ manually
          grd3 : $p \in PORT$  \\ manually
          grd4 : $i \in IP$  \\ manually
          grd5 : $i \in dom(endpointIp^{-1})$  \\ manually, WD
          grd6 : $acceptor(ch) = endpointIp^{-1}(i)$  \\ manually
          grd7 : $t = codecList2(ch)$  \\ manually, to prove (openAckWithoutCodecs/inv5/INV)
     then
          act_openAckWithoutCodecs : $openAckWithoutCodecs := openAckWithoutCodecs \cup \{ch\}$
          act1 : $acceptorPort2(ch) := p$  \\ manually, to prove (openAckWithoutCodecs/inv5/INV)
          act2 : $codecList2(ch) := t$  \\ manually, to prove (openAckWithoutCodecs/inv5/INV)
     end
Event  selectAndEstablishbyAcceptor ≙
extends  selectAndEstablishbyAcceptor
     any
          ch, c \\ manually
     where
          grd_selectAndEstablishbyAcceptor_seq : $ch \in openAckWithoutCodecs$
          grd_selectAndEstablishbyAcceptor : $ch \notin selectAndEstablishbyAcceptor$
```

```
            grd1 :  c ∈ codecList2(ch)  \\ manually
      then
            act_selectAndEstablishbyAcceptor :  selectAndEstablishbyAcceptor :=
                 selectAndEstablishbyAcceptor ∪ {ch}
            act1 :  codec(ch) := c  \\ manually
      end
Event   openWithoutCodecs ≙
extends  openWithoutCodecs
      any
            ch, cl \\ manually, p \\ manually, i \\ manually
      where
            grd_openWithoutCodecs :  ch ∉ openWithoutCodecs
            grd9 :  ch ∉ openWithRealCodecs
                 manually, derived from direction(ch) = ItoA, add to prove (inv7/INV)
            grd2 :  cl ⊆ CODEC  \\ manually
            grd3 :  cl = ∅  \\ manually
            grd4 :  p ∈ PORT  \\ manually
            grd5 :  i ∈ IP  \\ manually
            grd6 :  i ∈ dom(endpointIp⁻¹)  \\ manually, WD
            grd7 :  initiator(ch) = endpointIp⁻¹(i)  \\ manually
            grd8 :  direction(ch) = ItoA  \\ manually
      then
            act_openWithoutCodecs :  openWithoutCodecs := openWithoutCodecs ∪ {ch}
            act1 :  initiatorPort2(ch) := p  \\ manually
      end
Event   openAckWithRealCodecs ≙
extends  openAckWithRealCodecs
      any
            ch, cl \\ manually, p \\ manually, i \\ manually
      where
            grd_openAckWithRealCodecs_seq :  ch ∈ openWithoutCodecs
            grd_openAckWithRealCodecs :  ch ∉ openAckWithRealCodecs
            grd1 :  cl ⊆ CODEC  \\ manually
            grd2 :  cl ≠ ∅  \\ manually
            grd3 :  p ∈ PORT  \\ manually
            grd4 :  i ∈ IP  \\ manually
            grd5 :  i ∈ dom(endpointIp⁻¹)  \\ manually, WD
            grd6 :  acceptor(ch) = endpointIp⁻¹(i)  \\ manually
      then
            act_openAckWithRealCodecs :  openAckWithRealCodecs := openAckWithRealCodecs ∪ {ch}
            act1 :  codecList2(ch) := cl  \\ manually
            act2 :  acceptorPort2(ch) := p  \\ manually
      end
Event   selectAndEstablishbyInitiator ≙
extends  selectAndEstablishbyInitiator
      any
            ch, c \\ manually
      where
            grd_selectAndEstablishbyInitiator_seq :  ch ∈ openAckWithRealCodecs
            grd_selectAndEstablishbyInitiator :  ch ∉ selectAndEstablishbyInitiator
            grd1 :  c ∈ codecList2(ch)  \\ manually
      then
            act_selectAndEstablishbyInitiator :  selectAndEstablishbyInitiator :=
                 selectAndEstablishbyInitiator ∪ {ch}
            act1 :  codec(ch) := c  \\ manually
      end
Event   modifyCodecBySelector_withInitiator ≙
refines  modifyCodecBySelector_withInitiator
```

```
    any
        ch, c \\ manually
    where
        grd_modifyCodecBySelector_withInitiator_seq : ch ∈
            selectAndEstablishbyAcceptor ∪ selectAndEstablishbyInitiator
        grd_modifyCodecBySelector_withInitiator : ch ∉ modifyCodecBySelector_withInitiator
        grd_modifyCodecBySelector_withInitiator_xor : ch ∉ modifyCodecBySelector_withAcceptor
        grd_modifyCodecBySelector_withInitiator_loop : ch ∉ closeRequestAtoI ∪ closeRequestItoA
        grd1 : c ∈ codecList2(ch)
        grd2 : direction(ch) = ItoA
    then
        act_modifyCodecBySelector_withInitiator : modifyCodecBySelector_withInitiator :=
            modifyCodecBySelector_withInitiator ∪ {ch}
        act1 : codec(ch) := c
    end
Event   modifyCodecBySelector_withAcceptor ≙
refines modifyCodecBySelector_withAcceptor
    any
        ch, c \\ manually
    where
        grd_modifyCodecBySelector_withAcceptor_seq : ch ∈
            selectAndEstablishbyAcceptor ∪ selectAndEstablishbyInitiator
        grd_modifyCodecBySelector_withAcceptor : ch ∉ modifyCodecBySelector_withAcceptor
        grd_modifyCodecBySelector_withAcceptor_xor : ch ∉ modifyCodecBySelector_withInitiator
        grd_modifyCodecBySelector_withAcceptor_loop : ch ∉ closeRequestAtoI ∪ closeRequestItoA
        grd1 : c ∈ codecList2(ch)
        grd2 : direction(ch) = AtoI
    then
        act_modifyCodecBySelector_withAcceptor : modifyCodecBySelector_withAcceptor :=
            modifyCodecBySelector_withAcceptor ∪ {ch}
        act1 : codec(ch) := c
    end
Event   modify_Loop_Reset0 ≙
extends modify_Loop_Reset0
    any
        ch
    where
        grd_reset : ch ∈ modifyCodecBySelector_withInitiator∪modifyCodecBySelector_withAcceptor
    then
        act_reset_modifyCodecListByDescriptor : modifyCodecBySelector_withInitiator :=
            modifyCodecBySelector_withInitiator \ {ch}
        act_reset_modifyCodecBySelector_withAcceptor : modifyCodecBySelector_withAcceptor :=
            modifyCodecBySelector_withAcceptor \ {ch}
    end
Event   modifyCodecListByDescriptor_withInitiator ≙
refines modifyCodecListByDescriptor_withInitiator
    any
        ch, cl \\ manually
    where
        grd_modifyCodecListByDescriptor_withInitiator_seq : ch ∈
            selectAndEstablishbyAcceptor ∪ selectAndEstablishbyInitiator
        grd_modifyCodecListByDescriptor_withInitiator : ch ∉
            modifyCodecListByDescriptor_withInitiator
        grd_modifyCodecListByDescriptor_withInitiator_xor : ch ∉
            modifyCodecListByDescriptor_withAcceptor
        grd_modifyCodecListByDescriptor_withInitiator_loop : ch ∉
            closeRequestAtoI ∪ closeRequestItoA
        grd1 : cl ⊆ CODEC
        grd2 :  $cl \neq \varnothing$
        grd3 :  $direction(ch) = AtoI$
    then
        act_modifyCodecListByDescriptor_withInitiator :  $modifyCodecListByDescriptor\_withInitiator$
              $:= modifyCodecListByDescriptor\_withInitiator \cup \{ch\}$
        act1 :  $codecList2(ch) := cl$
    end
Event  $modifyCodecListByDescriptor\_withAcceptor \ \widehat{=}$
refines  $modifyCodecListByDescriptor\_withAcceptor$
    any
        ch, cl \\ manually
    where
        grd_modifyCodecListByDescriptor_withAcceptor_seq :  $ch \in$
              $selectAndEstablishbyAcceptor \cup selectAndEstablishbyInitiator$
        grd_modifyCodecListByDescriptor_withAcceptor :  $ch \notin$
              $modifyCodecListByDescriptor\_withAcceptor$
        grd_modifyCodecListByDescriptor_withAcceptor_xor :  $ch \notin$
              $modifyCodecListByDescriptor\_withInitiator$
        grd_modifyCodecListByDescriptor_withAcceptor_loop :  $ch \notin$
              $closeRequestAtoI \cup closeRequestItoA$
        grd1 :  $cl \subseteq CODEC$
        grd2 :  $cl \neq \varnothing$
        grd3 :  $direction(ch) = ItoA$
    then
        act_modifyCodecListByDescriptor_withAcceptor :  $modifyCodecListByDescriptor\_withAcceptor$
              $:= modifyCodecListByDescriptor\_withAcceptor \cup \{ch\}$
        act1 :  $codecList2(ch) := cl$
    end
Event  $respondBySelectorToInitiatorCodec \ \widehat{=}$
refines  $respondBySelectorToInitiatorCodec$
    any
        ch, c \\ manually
    where
        grd_respondBySelectorToInitiatorCodec_seq :  $ch \in$
              $modifyCodecListByDescriptor\_withInitiator \cup modifyCodecListByDescriptor\_withAcceptor$
        grd_respondBySelectorToInitiatorCodec :  $ch \notin respondBySelectorToInitiatorCodec$
        grd_respondBySelectorToInitiatorCodec_xor :  $ch \notin respondBySelectorToAcceptorCodec$
        grd1 :  $c \in codecList2(ch)$
        grd2 :  $direction(ch) = AtoI$
        grd3 :  $ch \notin closeRequestAtoI \cup closeRequestItoA$  \\ manually, from M3 to prove GRD
    then
        act_respondBySelectortoInitiatorCodec :  $respondBySelectorToInitiatorCodec :=$
              $respondBySelectorToInitiatorCodec \cup \{ch\}$
        act1 :  $codec(ch) := c$
    end
Event  $respondBySelectorToAcceptorCodec \ \widehat{=}$
refines  $respondBySelectorToAcceptorCodec$
    any
        ch, c \\ manually
    where
        grd_respondBySelectorToAcceptorCodec_seq :  $ch \in modifyCodecListByDescriptor\_withInitiator$
              $\cup\ modifyCodecListByDescriptor\_withAcceptor$
        grd_respondBySelectorToAcceptorCodec :  $ch \notin respondBySelectorToAcceptorCodec$
        grd_respondBySelectorToAcceptorCodec_xor :  $ch \notin respondBySelectorToInitiatorCodec$
        grd1 :  $c \in codecList2(ch)$
        grd2 :  $direction(ch) = ItoA$
        grd3 :  $ch \notin closeRequestAtoI \cup closeRequestItoA$  \\ manually, from M3 to prove GRD
    then
```

```
            act_respondBySelectortoAcceptorCodec :  $respondBySelectorToAcceptorCodec :=$
                 $respondBySelectorToAcceptorCodec \cup \{ch\}$
            act1 :  $codec(ch) := c$
      end
Event   modify_Loop_Reset1 ≙
extends  modify_Loop_Reset1
      any
            ch
      where
            grd_reset :  ch ∈
                 respondBySelectorToInitiatorCodec ∪ respondBySelectorToAcceptorCodec
      then
            act_reset_modifyCodecListByDescriptor_withInitiator :
                 modifyCodecListByDescriptor_withInitiator :=
                 modifyCodecListByDescriptor_withInitiator \ {ch}
            act_reset_modifyCodecListByDescriptor_withAcceptor :
                 modifyCodecListByDescriptor_withAcceptor :=
                 modifyCodecListByDescriptor_withAcceptor \ {ch}
            act_reset_respondBySelectorToInitiatorCodec :  respondBySelectorToInitiatorCodec
                 := respondBySelectorToInitiatorCodec \ {ch}
            act_reset_respondBySelectorToAcceptorCodec :  respondBySelectorToAcceptorCodec
                 := respondBySelectorToAcceptorCodec \ {ch}
      end
Event   modifyInitiatorPortByDescriptor ≙
refines  modifyInitiatorPortByDescriptor
      any
            ch, p \\ manually
      where
            grd_modifyInitiatorPortByDescriptor_seq :  $ch \in$
                 $selectAndEstablishbyAcceptor \cup selectAndEstablishbyInitiator$
            grd_modifyInitiatorPortByDescriptor :  $ch \notin modifyInitiatorPortByDescriptor$
            grd_modifyInitiatorPortByDescriptor_loop :  $ch \notin closeRequestAtoI \cup closeRequestItoA$
            grd1 :  $p \neq initiatorPort2(ch)$  \\ manually
      then
            act_modifyInitiatorPortByDescriptor :  $modifyInitiatorPortByDescriptor :=$
                 $modifyInitiatorPortByDescriptor \cup \{ch\}$
            act1 :  $initiatorPort2(ch) := p$  \\ manually
      end
Event   respondBySelectorToInitiatorPort ≙
extends  respondBySelectorToInitiatorPort
      any
            ch
      where
            grd_respondBySelectorToInitiatorPort_seq :  ch ∈ modifyInitiatorPortByDescriptor
            grd_respondBySelectorToInitiatorPort :  ch ∉ respondBySelectorToInitiatorPort
      then
            act_respondBySelectorToInitiatorPort :  respondBySelectorToInitiatorPort :=
                 respondBySelectorToInitiatorPort ∪ {ch}
      end
Event   modify_Loop_Reset2 ≙
extends  modify_Loop_Reset2
      any
            ch
      where
            grd_reset :  ch ∈ respondBySelectorToInitiatorPort
      then
```

```
        act_reset_modifyCodecListByDescriptor : modifyInitiatorPortByDescriptor :=
             modifyInitiatorPortByDescriptor \ {ch}
        act_reset_respondBySelectorToInitiatorPort : respondBySelectorToInitiatorPort :=
             respondBySelectorToInitiatorPort \ {ch}
  end
Event   modifyAcceptorPortByDescriptor ≙
refines  modifyAcceptorPortByDescriptor
  any
        ch, p \\ manually
  where
        grd_modifyAcceptorPortByDescriptor_seq : ch ∈
             selectAndEstablishbyAcceptor ∪ selectAndEstablishbyInitiator
        grd_modifyAcceptorPortByDescriptor : ch ∉ modifyAcceptorPortByDescriptor
        grd_modifyAcceptorPortByDescriptor_loop : ch ∉ closeRequestAtoI ∪ closeRequestItoA
        grd1 : p ≠ acceptorPort2(ch)  \\ manually
  then
        act_modifyAcceptorPortByDescriptor : modifyAcceptorPortByDescriptor :=
             modifyAcceptorPortByDescriptor ∪ {ch}
        act1 : acceptorPort2(ch) := p  \\ manually
  end
Event   respondBySelectorToAcceptorPort ≙
extends  respondBySelectorToAcceptorPort
  any
        ch
  where
        grd_respondBySelectorToAcceptorPort_seq : ch ∈ modifyAcceptorPortByDescriptor
        grd_respondBySelectorToAcceptorPort : ch ∉ respondBySelectorToAcceptorPort
  then
        act_respondBySelectorToAcceptorPort : respondBySelectorToAcceptorPort :=
             respondBySelectorToAcceptorPort ∪ {ch}
  end
Event   modify_Loop_Reset3 ≙
extends  modify_Loop_Reset3
  any
        ch
  where
        grd_reset : ch ∈ respondBySelectorToAcceptorPort
  then
        act_reset_modifyAcceptorPortByDescriptor : modifyAcceptorPortByDescriptor :=
             modifyAcceptorPortByDescriptor \ {ch}
        act_reset_respondBySelectorToAcceptorPort : respondBySelectorToAcceptorPort :=
             respondBySelectorToAcceptorPort \ {ch}
  end
Event   closeRequestAtoI ≙
refines  closeRequest
  any
        ch
  where
        grd_closeRequestAtoI_seq : ch ∈ selectAndEstablishbyAcceptor ∪ selectAndEstablishbyInitiator
        grd_closeRequestAtoI : ch ∉ closeRequestAtoI
        grd_closeRequestAtoI_xor : ch ∉ closeRequestItoA
        grd1 : direction(ch) = AtoI  \\ manually
  then
        act_closeRequestAtoI : closeRequestAtoI := closeRequestAtoI ∪ {ch}
  end
Event   closeRequestItoA ≙
refines  closeRequest
```

```
     any
          $ch$
     where
          grd_closeRequestItoA_sequencing : $ch \in$
               $selectAndEstablishbyAcceptor \cup selectAndEstablishbyInitiator$
          grd_closeRequestItoA : $ch \notin closeRequestItoA$
          grd_closeRequestItoA_xor : $ch \notin closeRequestAtoI$
          grd1 : $direction(ch) = ItoA$  \\ manually
     then
          act_closeRequestItoA : $closeRequestItoA := closeRequestItoA \cup \{ch\}$
     end
Event  $closeAckAtoI \widehat{=}$
refines  $closeAck$
     any
          $ch$
     where
          grd_closeAckAtoI_sequencing : $ch \in closeRequestAtoI \cup closeRequestItoA$
          grd_closeAckAtoI : $ch \notin closeAckAtoI$
          grd_closeAckAtoI_xor : $ch \notin closeAckItoA$
          grd1 : $direction(ch) = AtoI$  \\ manually
     then
          act_closeAckAtoI : $closeAckAtoI := closeAckAtoI \cup \{ch\}$
     end
Event  $closeAckItoA \widehat{=}$
refines  $closeAck$
     any
          $ch$
     where
          grd_closeAckItoA_sequencing : $ch \in closeRequestAtoI \cup closeRequestItoA$
          grd_closeAckItoA : $ch \notin closeAckItoA$
          grd_closeAckItoA_xor : $ch \notin closeAckAtoI$
          grd1 : $direction(ch) = ItoA$  \\ manually
     then
          act_closeAckItoA : $closeAckItoA := closeAckItoA \cup \{ch\}$
     end
END
```

Appendix B

The Event-B Model of the BepiColombo System

B.1 Abstract Specification

B.1.1 Context: *C0*

CONTEXT C0

SETS

TC \\ Telecommand, TC_Types_Set

CONSTANTS

SCI_on_TC, HK_off_TC, HK_on_TC, TC_Type, SCI_off_TC

AXIOMS

axm1 : $partition(TC\_Types\_Set, \{HK\_on\_TC\}, \{HK\_off\_TC\}, \{SCI\_on\_TC\}, \{SCI\_off\_TC\})$

axm2 : $TC\_Type \in TC \to TC\_Types\_Set$

END

B.1.2 Machine: *M0*

MACHINE M0

SEES C0

VARIABLES

ReceiveTC, TC_Validation_Ok, TCValid_GenerateData, TCValid_ReplyDataTM, TC_Validation_Fail

INVARIANTS

inv_ReceiveTC : $ReceiveTC \subseteq TC$

inv_TC_Validation_Ok_seq : $TC\_Validation\_Ok \subseteq ReceiveTC$

inv_TCValid_GenerateData_seq : $TCValid\_GenerateData \subseteq TC\_Validation\_Ok$

inv_TCValid_ReplyDataTM_seq : $TCValid\_ReplyDataTM \subseteq TCValid\_GenerateData$

inv_TC_Validation_Fail_seq : $TC\_Validation\_Fail \subseteq ReceiveTC$

inv1 : $TC\_Validation\_Ok \cap TC\_Validation\_Fail = \varnothing$ \\ manually

EVENTS

Initialisation

begin

act_ReceiveTC : $ReceiveTC := \varnothing$

```
        act_TC_Validation_Ok : $TC\_Validation\_Ok := \varnothing$
        act_TCValid_GenerateData : $TCValid\_GenerateData := \varnothing$
        act_TCValid_ReplyDataTM : $TCValid\_ReplyDataTM := \varnothing$
        act_TC_Validation_Fail : $TC\_Validation\_Fail := \varnothing$
    end
Event $ReceiveTC \widehat{=}$
    any
        $tc$
    where
        grd_ReceiveTC : $tc \notin ReceiveTC$
    then
        act_ReceiveTC : $ReceiveTC := ReceiveTC \cup \{tc\}$
    end
Event $TC\_Validation\_Ok \widehat{=}$
    any
        $tc$
    where
        grd_TC_Validation_Ok_seq : $tc \in ReceiveTC$
        grd_TC_Validation_Ok : $tc \notin TC\_Validation\_Ok$
        grd1 : $tc \notin TC\_Validation\_Fail$  \\ manually
    then
        act_TC_Validation_Ok : $TC\_Validation\_Ok := TC\_Validation\_Ok \cup \{tc\}$
    end
Event $TCValid\_GenerateData \widehat{=}$
    any
        $tc$
    where
        grd_TCValid_GenerateData_seq : $tc \in TC\_Validation\_Ok$
        grd_TCValid_GenerateData : $tc \notin TCValid\_GenerateData$
        grd1 : $TC\_Type(tc) \in \{HK\_on\_TC, SCI\_on\_TC\}$  \\ manually
    then
        act_TCValid_GenerateData : $TCValid\_GenerateData := TCValid\_GenerateData \cup \{tc\}$
    end
Event $TCValid\_ReplyDataTM \widehat{=}$
    any
        $tc$
    where
        grd_TCValid_ReplyDataTM_seq : $tc \in TCValid\_GenerateData$
        grd_TCValid_ReplyDataTM : $tc \notin TCValid\_ReplyDataTM$
    then
        act_TCValid_ReplyDataTM : $TCValid\_ReplyDataTM := TCValid\_ReplyDataTM \cup \{tc\}$
    end
Event $TC\_Validation\_Fail \widehat{=}$
    any
        $tc$
    where
        grd_TC_Validation_Fail_seq : $tc \in ReceiveTC$
        grd_TC_Validation_Fail : $tc \notin TC\_Validation\_Fail$
        grd1 : $tc \notin TC\_Validation\_Ok$  \\ manually
    then
        act_TC_Validation_Fail : $TC\_Validation\_Fail := TC\_Validation\_Fail \cup \{tc\}$
    end
END
```

B.2 1st Refinement

B.2.1 Machine: M1

MACHINE M1

REFINES M0

SEES C0

VARIABLES

ReceiveTC, TCCheck_Ok, TCExecute_Ok, TCExecOk_ReplyCtrlTM, TCValid_GenerateData, TCValid_ReplyDataTM, TCCheck_Fail, TCExecute_Fail, TCExecFail_ReplyCtrlTM, TCCheckFail_ReplyCtrlTM

INVARIANTS

```
inv_TCCheck_Ok_seq : $TCCheck\_Ok \subseteq ReceiveTC$
inv_TCExecute_Ok_seq : $TCExecute\_Ok \subseteq TCCheck\_Ok$
inv_TCExecOk_ReplyCtrlTM_seq : $TCExecOk\_ReplyCtrlTM \subseteq TCExecute\_Ok$
inv_TCCheck_Fail_seq : $TCCheck\_Fail \subseteq ReceiveTC$
inv_TCExecute_Fail_seq : $TCExecute\_Fail \subseteq TCCheck\_Ok$
inv_TCExecFail_ReplyCtrlTM_seq : $TCExecFail\_ReplyCtrlTM \subseteq TCExecute\_Fail$
inv_TCCheckFail_ReplyCtrlTM_seq : $TCCheckFail\_ReplyCtrlTM \subseteq TCCheck\_Fail$
inv_TCValid_GenerateData_seq : $TCValid\_GenerateData \subseteq TCExecute\_Ok$ \\ weak seq
inv_TCExecute_Ok_gluing : $TCExecute\_Ok = TC\_Validation\_Ok$
inv_gluing : $TCExecute\_Fail \cup TCCheck\_Fail = TC\_Validation\_Fail$
inv1 : $TCCheck\_Ok \cap TCCheck\_Fail = \varnothing$ \\ manually
inv2 : $TCExecute\_Ok \cap TCExecute\_Fail = \varnothing$ \\ manually
```

EVENTS

Initialisation

```
    begin
        act_ReceiveTC : $ReceiveTC := \varnothing$
        act_TCCheck_Ok : $TCCheck\_Ok := \varnothing$
        act_TCExecute_Ok : $TCExecute\_Ok := \varnothing$
        act_TCExecOk_ReplyCtrlTM : $TCExecOk\_ReplyCtrlTM := \varnothing$
        act_TCValid_GenerateData : $TCValid\_GenerateData := \varnothing$
        act_TCValid_ReplyDataTM : $TCValid\_ReplyDataTM := \varnothing$
        act_TCCheck_Fail : $TCCheck\_Fail := \varnothing$
        act_TCExecute_Fail : $TCExecute\_Fail := \varnothing$
        act_TCExecFail_ReplyCtrlTM : $TCExecFail\_ReplyCtrlTM := \varnothing$
        act_TCCheckFail_ReplyCtrlTM : $TCCheckFail\_ReplyCtrlTM := \varnothing$
    end
```

Event $ReceiveTC \widehat{=}$

refines $ReceiveTC$

```
    any
        $tc$
    where
        grd_ReceiveTC : $tc \notin ReceiveTC$
    then
        act_ReceiveTC : $ReceiveTC := ReceiveTC \cup \{tc\}$
    end
```

Event $TCCheck\_Ok \widehat{=}$

```
    any
        $tc$
    where
        grd_TCCheck_Ok_seq : $tc \in ReceiveTC$ \\ although in both weak and strong seq
        grd_TCCheck_Ok : $tc$
                                        $\notin TCCheck\_Ok$
```

```
            grd1 : $tc \notin TCCheck\_Fail$  \\ manually
        then
            act_TCCheck_Ok : $TCCheck\_Ok := TCCheck\_Ok \cup \{tc\}$
        end
Event   $TCExecute\_Ok \mathrel{\widehat{=}}$
refines   $TC\_Validation\_Ok$
        any
            $tc$
        where
            grd_TCExecute_Ok_seq : $tc \in TCCheck\_Ok$
            grd_TCExecute_Ok : $tc \notin TCExecute\_Ok$
            grd1 : $tc \notin TCExecute\_Fail$  \\ manually
        then
            act_TCExecute_Ok : $TCExecute\_Ok := TCExecute\_Ok \cup \{tc\}$
        end
Event   $TCExecOk\_ReplyCtrlTM \mathrel{\widehat{=}}$
        any
            $tc$
        where
            grd_TCExecOk_ReplyCtrlTM_seq : $tc \in TCExecute\_Ok$
            grd_TCExecOk_ReplyCtrlTM : $tc \notin TCExecOk\_ReplyCtrlTM$
        then
            act_TCExecOk_ReplyCtrlTM : $TCExecOk\_ReplyCtrlTM := TCExecOk\_ReplyCtrlTM \cup \{tc\}$
        end
Event   $TCValid\_GenerateData \mathrel{\widehat{=}}$
refines   $TCValid\_GenerateData$
        any
            $tc$
        where
            grd_TCValid_GenerateData_seq : $tc \in TCExecute\_Ok$
            grd_TCValid_GenerateData : $tc \notin TCValid\_GenerateData$
            grd1 : $TC\_Type(tc) \in \{HK\_on\_TC, SCI\_on\_TC\}$  \\ manually
        then
            act_TCValid_GenerateData : $TCValid\_GenerateData := TCValid\_GenerateData \cup \{tc\}$
        end
Event   $TCValid\_ReplyDataTM \mathrel{\widehat{=}}$
extends   $TCValid\_ReplyDataTM$
        any
            tc
        where
            grd_TCValid_ReplyDataTM_seq : tc ∈ TCValid_GenerateData
            grd_TCValid_ReplyDataTM : tc ∉ TCValid_ReplyDataTM
        then
            act_TCValid_ReplyDataTM : TCValid_ReplyDataTM := TCValid_ReplyDataTM ∪ {tc}
        end
Event   $TCExecute\_Fail \mathrel{\widehat{=}}$
refines   $TC\_Validation\_Fail$
        any
            $tc$
        where
            grd_TCExecute_Fail_seq : $tc \in TCCheck\_Ok$
            grd_TCExecute_Fail : $tc \notin TCExecute\_Ok$
            grd1 : $tc \notin TCExecute\_Fail$
        then
            act_TCExecute_Fail : $TCExecute\_Fail := TCExecute\_Fail \cup \{tc\}$
        end
```

```
Event  $TCExecFail\_ReplyCtrlTM \widehat{=}$
    any
        $tc$
    where
        grd_TCExecFail_ReplyCtrlTM_seq :  $tc \in TCExecute\_Fail$
        grd_TCExecFail_ReplyCtrlTM :  $tc \notin TCExecFail\_ReplyCtrlTM$
    then
        act_TCExecFail_ReplyCtrlTM :  $TCExecFail\_ReplyCtrlTM :=$
            $TCExecFail\_ReplyCtrlTM \cup \{tc\}$
    end
Event  $TCCheck\_Fail \widehat{=}$
refines  $TC\_Validation\_Fail$
    any
        $tc$
    where
        grd_TCCheck_Fail_seq :  $tc \in ReceiveTC$
        grd_TCCheck_Fail :  $tc \notin TCCheck\_Ok$
        grd1 :  $tc \notin TCCheck\_Fail$
    then
        act_TCCheck_Fail :  $TCCheck\_Fail := TCCheck\_Fail \cup \{tc\}$
    end
Event  $TCCheckFail\_ReplyCtrlTM \widehat{=}$
    any
        $tc$
    where
        grd_TCCheckFail_ReplyCtrlTM_seq :  $tc \in TCCheck\_Fail$
        grd_TCCheckFail_ReplyCtrlTM :  $tc \notin TCCheckFail\_ReplyCtrlTM$
    then
        act_TCCheckFail_ReplyCtrlTM :  $TCCheckFail\_ReplyCtrlTM :=$
            $TCCheckFail\_ReplyCtrlTM \cup \{tc\}$
    end
END
```

B.3 2nd Refinement

B.3.1 Context: *C1*

```
CONTEXT  C1
EXTENDS  C0
SETS
    PIDS
CONSTANTS
    PID, csw, sixsp, sixsx, mixst, mixsc
AXIOMS
    axm1 :  $partition(PIDS, \{csw\}, \{mixsc\}, \{mixst\}, \{sixsp\}, \{sixsx\})$
    axm2 :  $PID \in TC \to PIDS$
END
```

B.3.2 Machine: M2

MACHINE M2

REFINES M1

SEES C1

VARIABLES

ReceiveTC, TCCheck_Ok, TCCore_Execute_Ok, TCDevice_Execute_Ok, TCCheck_Fail, TCCore_Execute_Fail, TCDevice_Execute_Fail, TCExecOk_ReplyCtrlTM, TCValid_ReplyDataTM, TCExecFail_ReplyCtrlTM, TCCheckFail_ReplyCtrlTM, TCValid_GenerateData

INVARIANTS

inv_TCCore_Execute_Ok_seq : $TCCore\_Execute\_Ok \subseteq TCCheck\_Ok$

inv_TCDevice_Execute_Ok_seq : $TCDevice\_Execute\_Ok \subseteq TCCheck\_Ok$

inv_TCExecOk_ReplyCtrlTM_seq : $TCExecOk\_ReplyCtrlTM \subseteq TCCore\_Execute\_Ok \cup TCDevice\_Execute\_Ok$

inv_TCCore_Execute_Fail_seq : $TCCore\_Execute\_Fail \subseteq TCCheck\_Ok$

inv_TCDevice_Execute_Fail_seq : $TCDevice\_Execute\_Fail \subseteq TCCheck\_Ok$

inv_TCExecFail_ReplyCtrlTM_seq : $TCExecFail\_ReplyCtrlTM \subseteq TCCore\_Execute\_Fail \cup TCDevice\_Execute\_Fail$

inv_TCValid_GenerateData_seq : $TCValid\_GenerateData \subseteq TCCore\_Execute\_Ok \cup TCDevice\_Execute\_Ok$ \\ weak seq

inv_xor_gluing1 : $partition(TCExecute\_Ok, TCCore\_Execute\_Ok, TCDevice\_Execute\_Ok)$

inv_xor_gluing2 : $partition(TCExecute\_Fail, TCCore\_Execute\_Fail, TCDevice\_Execute\_Fail)$

inv1 : $partition(TCCore\_Execute\_Ok \cup TCCore\_Execute\_Fail \cup TCDevice\_Execute\_Ok \cup TCDevice\_Execute\_Fail, TCCore\_Execute\_Ok, TCCore\_Execute\_Fail, TCDevice\_Execute\_Ok, TCDevice\_Execute\_Fail)$ \\ manually

EVENTS

Initialisation

begin

act_ReceiveTC : $ReceiveTC := \varnothing$

act_TCCheck_Ok : $TCCheck\_Ok := \varnothing$

act_TCCore_Execute_Ok : $TCCore\_Execute\_Ok := \varnothing$

act_TCDevice_Execute_Ok : $TCDevice\_Execute\_Ok := \varnothing$

act_TCCheck_Fail : $TCCheck\_Fail := \varnothing$

act_TCCore_Execute_Fail : $TCCore\_Execute\_Fail := \varnothing$

act_TCDevice_Execute_Fail : $TCDevice\_Execute\_Fail := \varnothing$

act_TCExecOk_ReplyCtrlTM : $TCExecOk\_ReplyCtrlTM := \varnothing$

act_TCValid_GenerateData : $TCValid\_GenerateData := \varnothing$

act_TCValid_ReplyDataTM : $TCValid\_ReplyDataTM := \varnothing$

act_TCExecFail_ReplyCtrlTM : $TCExecFail\_ReplyCtrlTM := \varnothing$

act_TCCheckFail_ReplyCtrlTM : $TCCheckFail\_ReplyCtrlTM := \varnothing$

end

Event $ReceiveTC \;\widehat{=}$

refines $ReceiveTC$

any

tc

where

grd_ReceiveTC : $tc \notin ReceiveTC$

then

act_ReceiveTC : $ReceiveTC := ReceiveTC \cup \{tc\}$

end

Event $TCCheck\_Ok \;\widehat{=}$

refines $TCCheck\_Ok$

any

tc

where

```
          grd_TCCheck_Ok_seq :  tc ∈ ReceiveTC
          grd_TCCheck_Ok :  tc ∉ TCCheck_Ok
          grd1 :  tc ∉ TCCheck_Fail
     then
          act_TCCheck_Ok :  TCCheck_Ok := TCCheck_Ok ∪ {tc}
     end
Event   TCCheck_Fail ≙
refines   TCCheck_Fail
     any
          tc
     where
          grd_TCCheck_Fail_seq :  tc ∈ ReceiveTC
          grd_TCCheck_Fail :  tc ∉ TCCheck_Ok
          grd1 :  tc ∉ TCCheck_Fail
     then
          act_TCCheck_Fail :  TCCheck_Fail := TCCheck_Fail ∪ {tc}
     end
Event   TCCore_Execute_Ok ≙
refines   TCExecute_Ok
     any
          tc
     where
          grd_TCCore_Execute_Ok_seq :  tc ∈ TCCheck_Ok
          grd_TCCore_Execute_Ok :  tc ∉ TCCore_Execute_Ok
          grd_TCCore_Execute_Ok_xor :  tc ∉ TCDevice_Execute_Ok
          grd1 :  tc ∉ TCCore_Execute_Fail  \\ manually
          grd2 :  tc ∉ TCDevice_Execute_Fail  \\ manually
          grd3 :  PID(tc) = csw  \\ manually
     then
          act_TCCore_Execute_Ok :  TCCore_Execute_Ok := TCCore_Execute_Ok ∪ {tc}
     end
Event   TCDevice_Execute_Ok ≙
refines   TCExecute_Ok
     any
          tc
     where
          grd_TCDevice_Execute_Ok_seq :  tc ∈ TCCheck_Ok
          grd_TCDevice_Execute_Ok :  tc ∉ TCDevice_Execute_Ok
          grd_TCDevice_Execute_Ok_xor :  tc ∉ TCCore_Execute_Ok
          grd1 :  tc ∉ TCDevice_Execute_Fail  \\ manually
          grd2 :  tc ∉ TCCore_Execute_Fail  \\ manually
          grd3 :  PID(tc) ∈ {mixsc, mixst, sixsp, sixsx}  \\ manually
     then
          act_TCDevice_Execute_Ok :  TCDevice_Execute_Ok := TCDevice_Execute_Ok ∪ {tc}
     end
Event   TCCore_Execute_Fail ≙
refines   TCExecute_Fail
     any
          tc
     where
          grd_TCCore_Execute_Fail_seq :  tc ∈ TCCheck_Ok
          grd_TCCore_Execute_Fail :  tc ∉ TCCore_Execute_Fail
          grd_TCCore_Execute_Fail_xor :  tc ∉ TCDevice_Execute_Fail
          grd1 :  tc ∉ TCCore_Execute_Ok  \\ manually
          grd2 :  tc ∉ TCDevice_Execute_Ok  \\ manually
          grd3 :  PID(tc) = csw  \\ manually
     then
```

```
        act_TCCore_Execute_Fail :  $TCCore\_Execute\_Fail := TCCore\_Execute\_Fail \cup \{tc\}$
    end
Event   $TCDevice\_Execute\_Fail \widehat{=}$
refines  $TCExecute\_Fail$
    any
        $tc$
    where
        grd_TCDevice_Execute_Fail_seq :  $tc \in TCCheck\_Ok$
        grd_TCDevice_Execute_Fail :  $tc \notin TCDevice\_Execute\_Fail$
        grd_TCDevice_Execute_Fail_xor :  $tc \notin TCCore\_Execute\_Fail$
        grd1 :  $tc \notin TCDevice\_Execute\_Ok$  \\ manually
        grd2 :  $tc \notin TCCore\_Execute\_Ok$  \\ manually
        grd3 :  $PID(tc) \in \{mixsc, mixst, sixsp, sixsx\}$  \\ manually
    then
        act_TCDevice_Execute_Fail :  $TCDevice\_Execute\_Fail := TCDevice\_Execute\_Fail \cup \{tc\}$
    end
Event   $TCValid\_GenerateData \widehat{=}$
refines  $TCValid\_GenerateData$
    any
        $tc$
    where
        grd_TCValid_GenerateData_seq :  $tc \in TCCore\_Execute\_Ok \cup TCDevice\_Execute\_Ok$
        grd_TCValid_GenerateData :  $tc \notin TCValid\_GenerateData$
        grd1 :  $TC\_Type(tc) \in \{HK\_on\_TC, SCI\_on\_TC\}$  \\ manually
        grd2 :  $PID(tc) \in \{mixsc, mixst, sixsp, sixsx\}$  \\ manually, it limits to tc : TCDevice_Execute_Ok
    then
        act_TCValid_GenerateData :  $TCValid\_GenerateData := TCValid\_GenerateData \cup \{tc\}$
    end
Event   $TCValid\_ReplyDataTM \widehat{=}$
extends  $TCValid\_ReplyDataTM$
    any
        tc
    where
        grd_TCValid_ReplyDataTM_seq :  tc ∈ TCValid_GenerateData
        grd_TCValid_ReplyDataTM :  tc ∉ TCValid_ReplyDataTM
    then
        act_TCValid_ReplyDataTM :  TCValid_ReplyDataTM := TCValid_ReplyDataTM ∪ {tc}
    end
Event   $TCExecFail\_ReplyCtrlTM \widehat{=}$
refines  $TCExecFail\_ReplyCtrlTM$
    any
        $tc$
    where
        grd_TCExecFail_ReplyCtrlTM_seq :  $tc \in TCCore\_Execute\_Fail \cup TCDevice\_Execute\_Fail$
        grd_TCExecFail_ReplyCtrlTM :  $tc \notin TCExecFail\_ReplyCtrlTM$
    then
        act_TCExecFail_ReplyCtrlTM :  $TCExecFail\_ReplyCtrlTM := TCExecFail\_ReplyCtrlTM \cup \{tc\}$
    end
Event   $TCCheckFail\_ReplyCtrlTM \widehat{=}$
refines  $TCCheckFail\_ReplyCtrlTM$
    any
        $tc$
    where
        grd_TCCheckFail_ReplyCtrlTM_seq :  $tc \in TCCheck\_Fail$
        grd_TCCheckFail_ReplyCtrlTM :  $tc \notin TCCheckFail\_ReplyCtrlTM$
    then
```

act_TCCheckFail_ReplyCtrlTM : $TCCheckFail\_ReplyCtrlTM :=$
$TCCheckFail\_ReplyCtrlTM \cup \{tc\}$

end

Event $TCExecOk\_ReplyCtrlTM \;\widehat{=}$

refines $TCExecOk\_ReplyCtrlTM$

any

tc

where

grd_TCExecOk_ReplyCtrlTM_seq : $tc \in TCCore\_Execute\_Ok \cup TCDevice\_Execute\_Ok$

grd_TCExecOk_ReplyCtrlTM : $tc \notin TCExecOk\_ReplyCtrlTM$

then

act1 : $TCExecOk\_ReplyCtrlTM := TCExecOk\_ReplyCtrlTM \cup \{tc\}$

end

END

B.4 3rd Refinement

B.4.1 Context: *C2*

CONTEXT C2

EXTENDS C1

SETS

DATA

END

B.4.2 Machine: M3

MACHINE M3

REFINES M2

SEES C2

VARIABLES

ReceiveTC, TCCheck_Ok, TCCore_Execute_Ok, SendTC_Core_to_Device, CheckTC_in_Device_Ok, SendOkTC_Device_to_Core, TCCheck_Fail, TCCore_Execute_Fail, CheckTC_in_Device_Fail, SendFailTC_Device_to_Core, TC_GenerateData_in_Device, TC_TransferData_Device_to_Core, TCValid_ReplyDataTM, TCExecOk_ReplyCtrlTM, TCExecFail_ReplyCtrlTM, TCCheckFail_ReplyCtrlTM

INVARIANTS

inv_SendTC_Core_to_Device_seq : $SendTC\_Core\_to\_Device \subseteq TCCheck\_Ok$

inv_CheckTC_in_Device_Ok_seq : $CheckTC\_in\_Device\_Ok \subseteq SendTC\_Core\_to\_Device$

inv_CheckTC_in_Device_Fail : $CheckTC\_in\_Device\_Fail \subseteq SendTC\_Core\_to\_Device$

inv_SendOkTC_Device_to_Core_seq : $SendOkTC\_Device\_to\_Core \subseteq CheckTC\_in\_Device\_Ok$

inv_SendFailTC_Device_to_Core_seq : $SendFailTC\_Device\_to\_Core \subseteq$
$CheckTC\_in\_Device\_Fail$

inv_TCExecOk_ReplyCtrlTM_seq : $TCExecOk\_ReplyCtrlTM \subseteq$
$TCCore\_Execute\_Ok \cup SendOkTC\_Device\_to\_Core$

linv_TCExec_ReplyCtrlTM_seq : $TCExecFail\_ReplyCtrlTM \subseteq$
$TCCore\_Execute\_Fail \cup SendFailTC\_Device\_to\_Core$

inv_TC_GenerateData_in_Device : $TC\_GenerateData\_in\_Device \subseteq TC \times DATA$

inv_TC_GenerateData_in_Device_seq : $dom(TC\_GenerateData\_in\_Device) \subseteq$
$TCCore\_Execute\_Ok \cup SendOkTC\_Device\_to\_Core$ \\ weak seq

inv_TC_TransferData_Device_to_Core_seq : $TC\_TransferData\_Device\_to\_Core \subseteq$
$dom(TC\_GenerateData\_in\_Device)$

inv_xor1 : $partition(TCCore\_Execute\_Ok \cup CheckTC\_in\_Device\_Ok,$
$TCCore\_Execute\_Ok, CheckTC\_in\_Device\_Ok)$

inv_xor2 : $partition(TCCore\_Execute\_Fail \cup CheckTC\_in\_Device\_Fail,$
$TCCore\_Execute\_Fail, CheckTC\_in\_Device\_Fail)$

inv_CheckTC_in_Device_Ok_gluing : $CheckTC\_in\_Device\_Ok = TCDevice\_Execute\_Ok$

inv_CheckTC_in_Device_Fail_gluing : $CheckTC\_in\_Device\_Fail = TCDevice\_Execute\_Fail$

inv_TC_TransferData_Device_to_Core_gluing : $TC\_TransferData\_Device\_to\_Core =$
$TCValid\_GenerateData$

inv2 : $partition(TCCore\_Execute\_Ok \cup TCCore\_Execute\_Fail \cup SendTC\_Core\_to\_Device,$
$TCCore\_Execute\_Ok, TCCore\_Execute\_Fail, SendTC\_Core\_to\_Device)$ \\\\ manually

inv5 : $CheckTC\_in\_Device\_Ok \cap CheckTC\_in\_Device\_Fail = \varnothing$ \\\\ manually

inv6 : $\forall tc \cdot (tc \in dom(TC\_GenerateData\_in\_Device) \Rightarrow$
$TC\_Type(tc) \in \{HK\_on\_TC, SCI\_on\_TC\})$
\\\\ manually, proving (TransferData_Device_to_Core/GRD)

inv7 : $\forall tc \cdot (tc \in dom(TC\_GenerateData\_in\_Device) \Rightarrow PID(tc) \in \{mixsc, mixst, sixsp, sixsx\})$
\\\\ manually, proving (TransferData_Device_to_Core/GRD)

inv8 : $\forall tc \cdot (tc \in SendTC\_Core\_to\_Device \Rightarrow PID(tc) \in \{mixsc, mixst, sixsp, sixsx\})$
\\\\ manually, proving (CheckTC_in_Device_Fail/GRD)

EVENTS

Initialisation

begin

act_ReceiveTC : $ReceiveTC := \varnothing$
act_TCCheck_Ok : $TCCheck\_Ok := \varnothing$
act_TCCore_Execute_Ok : $TCCore\_Execute\_Ok := \varnothing$
act_SendTC_Core_to_Device : $SendTC\_Core\_to\_Device := \varnothing$
act_CheckTC_in_Device_Ok : $CheckTC\_in\_Device\_Ok := \varnothing$
act_SendOkTC_Device_to_Core : $SendOkTC\_Device\_to\_Core := \varnothing$
act_TCCheck_Fail : $TCCheck\_Fail := \varnothing$
act_TCCore_Execute_Fail : $TCCore\_Execute\_Fail := \varnothing$
act_CheckTC_in_Device_Fail : $CheckTC\_in\_Device\_Fail := \varnothing$
act_SendFailTC_Device_to_Core : $SendFailTC\_Device\_to\_Core := \varnothing$
act_TC_GenerateData_in_Device : $TC\_GenerateData\_in\_Device := \varnothing$
act_TC_TransferData_Device_to_Core : $TC\_TransferData\_Device\_to\_Core := \varnothing$
act_TCValid_ReplyDataTM : $TCValid\_ReplyDataTM := \varnothing$
act_TCExecOk_ReplyCtrlTM : $TCExecOk\_ReplyCtrlTM := \varnothing$
act_TCExecFail_ReplyCtrlTM : $TCExecFail\_ReplyCtrlTM := \varnothing$
act_TCCheckFail_ReplyCtrlTM : $TCCheckFail\_ReplyCtrlTM := \varnothing$

end

Event $ReceiveTC \mathrel{\widehat{=}}$

extends $ReceiveTC$

any

tc

where

grd_ReceiveTC : $\mathtt{tc} \notin \mathtt{ReceiveTC}$

then

act_ReceiveTC : $\mathtt{ReceiveTC} := \mathtt{ReceiveTC} \cup \{\mathtt{tc}\}$

end

Event $TCCheck\_Ok \mathrel{\widehat{=}}$

extends $TCCheck\_Ok$

any

tc

where

grd_TCCheck_Ok_seq : $\mathtt{tc} \in \mathtt{ReceiveTC}$
grd_TCCheck_Ok : $\mathtt{tc} \notin \mathtt{TCCheck\_Ok}$
grd1 : $\mathtt{tc} \notin \mathtt{TCCheck\_Fail}$

then

```
          act_TCCheck_Ok :  TCCheck_Ok := TCCheck_Ok ∪ {tc}
      end
Event   TCCore_Execute_Ok ≙
refines   TCCore_Execute_Ok
      any
          tc
      where
          grd_TCCore_Execute_Ok_seq :  tc ∈ TCCheck_Ok
          grd_TCCore_Execute_Ok :  tc ∉ TCCore_Execute_Ok
          grd_TCCore_Execute_Ok_xor :  tc ∉ SendTC_Core_to_Device
          grd2 :  tc ∉ TCCore_Execute_Fail  \\ manually
          grd3 :  PID(tc) = csw  \\ manually
      then
          act_TCCore_Execute_Ok :  TCCore_Execute_Ok := TCCore_Execute_Ok ∪ {tc}
      end
Event   SendTC_Core_to_Device ≙
      any
          tc
      where
          grd_SendTC_Core_to_Device_seq :  tc ∈ TCCheck_Ok
          grd_SendTC_Core_to_Device :  tc ∉ SendTC_Core_to_Device
          grd_SendTC_Core_to_Device_xor1 :  tc ∉ TCCore_Execute_Ok
          grd_SendTC_Core_to_Device_xor2 :  tc ∉ TCCore_Execute_Fail
          grd1 :  PID(tc) ∈ {mixsc, mixst, sixsp, sixsx}
      then
          act_SendTC_Core_to_Device :  SendTC_Core_to_Device := SendTC_Core_to_Device ∪ {tc}
      end
Event   CheckTC_in_Device_Ok ≙
refines   TCDevice_Execute_Ok
      any
          tc
      where
          grd_CheckTC_in_Device_Ok_seq :  tc ∈ SendTC_Core_to_Device
          grd_CheckTC_in_Device_Ok :  tc ∉ CheckTC_in_Device_Ok
          grd1 :  tc ∉ CheckTC_in_Device_Fail
      then
          act_CheckTC_in_Device_Ok :  CheckTC_in_Device_Ok :=
                CheckTC_in_Device_Ok ∪ {tc}
      end
Event   SendOkTC_Device_to_Core ≙
      any
          tc
      where
          grd_SendOkTC_Device_to_Core_seq :  tc ∈ CheckTC_in_Device_Ok
          grd_SendOkTC_Device_to_Core :  tc ∉ SendOkTC_Device_to_Core
      then
          act_SendOkTC_Device_to_Core :  SendOkTC_Device_to_Core :=
                SendOkTC_Device_to_Core ∪ {tc}
      end
Event   TCCore_Execute_Fail ≙
refines   TCCore_Execute_Fail
      any
          tc
      where
          grd_TCCore_Execute_Fail_seq :  tc ∈ TCCheck_Ok
          grd_TCCore_Execute_Fail :  tc ∉ TCCore_Execute_Fail
          grd_TCCore_Execute_Fail_xor :  tc ∉ SendTC_Core_to_Device
```

```
        grd2 : $tc \notin TCCore\_Execute\_Ok$  \\ manually
        grd3 : $PID(tc) = csw$  \\ manually
    then
        act_TCCore_Execute_Fail : $TCCore\_Execute\_Fail := TCCore\_Execute\_Fail \cup \{tc\}$
    end
Event  $CheckTC\_in\_Device\_Fail \;\widehat{=}$
refines  $TCDevice\_Execute\_Fail$
    any
        $tc$
    where
        grd_CheckTC_in_Device_Fail_seq : $tc \in SendTC\_Core\_to\_Device$
        grd_CheckTC_in_Device_Fail : $tc \notin CheckTC\_in\_Device\_Fail$
        grd1 : $tc \notin CheckTC\_in\_Device\_Ok$
    then
        act_CheckTC_in_Device_Fail : $CheckTC\_in\_Device\_Fail :=$
            $CheckTC\_in\_Device\_Fail \cup \{tc\}$
    end
Event  $SendFailTC\_Device\_to\_Core \;\widehat{=}$
    any
        $tc$
    where
        grd_SendFailTC_Device_to_Core_seq : $tc \in CheckTC\_in\_Device\_Fail$
        grd_SendFailTC_Device_to_Core : $tc \notin SendFailTC\_Device\_to\_Core$
    then
        act_SendFailTC_Device_to_Core : $SendFailTC\_Device\_to\_Core :=$
            $SendFailTC\_Device\_to\_Core \cup \{tc\}$
    end
Event  $TCCheck\_Fail \;\widehat{=}$
extends  $TCCheck\_Fail$
    any
        tc
    where
        grd_TCCheck_Fail_seq : tc $\in$ ReceiveTC
        grd_TCCheck_Fail : tc $\notin$ TCCheck_Ok
        grd1 : tc $\notin$ TCCheck_Fail
    then
        act_TCCheck_Fail : TCCheck_Fail := TCCheck_Fail $\cup$ {tc}
    end
Event  $TC\_GenerateData\_in\_Device \;\widehat{=}$
    any
        $tc$
        $d$
    where
        grd_TC_GenerateData_in_Device_seq : $tc \in$
            $TCCore\_Execute\_Ok \cup SendOkTC\_Device\_to\_Core$
        grd_TC_GenerateData_in_Device : $tc \mapsto d \notin TC\_GenerateData\_in\_Device$
        grd1 : $TC\_Type(tc) \in \{HK\_on\_TC, SCI\_on\_TC\}$  \\ manually
        grd2 : $PID(tc) \in \{mixsc, mixst, sixsp, sixsx\}$
            manually, it limits to tc : SendOkTC_Device_to_Core
    then
        act_TC_GenerateData_in_Device : $TC\_GenerateData\_in\_Device :=$
            $TC\_GenerateData\_in\_Device \cup \{tc \mapsto d\}$
    end
Event  $TC\_TransferData\_Device\_to\_Core \;\widehat{=}$
refines  $TCValid\_GenerateData$
    any
        $tc$
```

```
          data
      where
          grd_TC_TransferData_Device_to_Core_seq : tc ∈ dom(TC_GenerateData_in_Device)
          grd_TC_TransferData_Device_to_Core : tc ∉ TC_TransferData_Device_to_Core
          grd1 : data = TC_GenerateData_in_Device[{tc}]  \\ manually
      then
          act_TC_TransferData_Device_to_Core : TC_TransferData_Device_to_Core :=
              TC_TransferData_Device_to_Core ∪ {tc}
      end
Event   TCValid_ReplyDataTM ≙
refines  TCValid_ReplyDataTM
      any
          tc
      where
          grd_TCValid_ReplyDataTM_seq : tc ∈ TC_TransferData_Device_to_Core
          grd_TCValid_ReplyDataTM : tc ∉ TCValid_ReplyDataTM
      then
          act_TCValid_ReplyDataTM : TCValid_ReplyDataTM := TCValid_ReplyDataTM ∪ {tc}
      end
Event   TCExecOk_ReplyCtrlTM ≙
refines  TCExecOk_ReplyCtrlTM
      any
          tc
      where
          grd_TCExecOk_ReplyCtrlTM_seq : tc ∈
              TCCore_Execute_Ok ∪ SendOkTC_Device_to_Core  \\ weak seq
          grd_TCExecOk_ReplyCtrlTM : tc ∉ TCExecOk_ReplyCtrlTM
      then
          act_TCExecOk_ReplyCtrlTM : TCExecOk_ReplyCtrlTM :=
              TCExecOk_ReplyCtrlTM ∪ {tc}
      end
Event   TCExecFail_ReplyCtrlTM ≙
refines  TCExecFail_ReplyCtrlTM
      any
          tc
      where
          grd_TCExecFail_ReplyCtrlTM_seq : tc ∈
              TCCore_Execute_Fail ∪ SendFailTC_Device_to_Core
          grd_TCExecFail_ReplyCtrlTM : tc ∉ TCExecFail_ReplyCtrlTM
      then
          act_TCExecFail_ReplyCtrlTM : TCExecFail_ReplyCtrlTM :=
              TCExecFail_ReplyCtrlTM ∪ {tc}
      end
Event   TCCheckFail_ReplyCtrlTM ≙
extends  TCCheckFail_ReplyCtrlTM
      any
          tc
      where
          grd_TCCheckFail_ReplyCtrlTM_seq : tc ∈ TCCheck_Fail
          grd_TCCheckFail_ReplyCtrlTM : tc ∉ TCCheckFail_ReplyCtrlTM
      then
          act_TCCheckFail_ReplyCtrlTM : TCCheckFail_ReplyCtrlTM :=
              TCCheckFail_ReplyCtrlTM ∪ {tc}
      end
END
```

B.5 Core Sub-model

B.5.1 Context: *Context_M3*

CONTEXT Context_M3

SETS

TC, PIDS, TC_Types_Set

CONSTANTS

PID, csw, mixsc, mixst, sixsp, sixsx, TC_Type, HK_on_TC, SCI_on_TC

AXIOMS

`typing_PID :` $PID \in \mathbb{P}(TC \times PIDS)$

`typing_csw :` $csw \in PIDS$

`typing_mixsc :` $mixsc \in PIDS$

`typing_mixst :` $mixst \in PIDS$

`typing_sixsp :` $sixsp \in PIDS$

`typing_sixsx :` $sixsx \in PIDS$

`typing_TC_Type :` $TC\_Type \in \mathbb{P}(TC \times TC\_Types\_Set)$

`typing_HK_on_TC :` $HK\_on\_TC \in TC\_Types\_Set$

`typing_SCI_on_TC :` $SCI\_on\_TC \in TC\_Types\_Set$

`C0_axm2 :` $TC\_Type \in TC \rightarrow TC\_Types\_Set$

`C1_axm1 :` $partition(PIDS, \{csw\}, \{mixsc\}, \{mixst\}, \{sixsp\}, \{sixsx\})$

`C1_axm2 :` $PID \in TC \rightarrow PIDS$

END

B.5.2 Machine: M3

MACHINE M3

SEES Context_M3

VARIABLES

ReceiveTC, TCCheck_Ok, TCCore_Execute_Ok, SendOkTC_Device_to_Core, TCCheck_Fail, TCCore_Execute_Fail, SendFailTC_Device_to_Core, TC_TransferData_Device_to_Core, TCValid_ReplyDataTM, TCExecOk_ReplyCtrlTM, TCExecFail_ReplyCtrlTM, TCCheckFail_ReplyCtrlTM

INVARIANTS

`typing_TCCheck_Fail :` $TCCheck\_Fail \in \mathbb{P}(TC)$

`typing_TCCheck_Ok :` $TCCheck\_Ok \in \mathbb{P}(TC)$

`typing_TCCheckFail_ReplyCtrlTM :` $TCCheckFail\_ReplyCtrlTM \in \mathbb{P}(TC)$

`typing_ReceiveTC :` $ReceiveTC \in \mathbb{P}(TC)$

`typing_TCValid_ReplyDataTM :` $TCValid\_ReplyDataTM \in \mathbb{P}(TC)$

`typing_TCCore_Execute_Ok :` $TCCore\_Execute\_Ok \in \mathbb{P}(TC)$

`typing_SendFailTC_Device_to_Core :` $SendFailTC\_Device\_to\_Core \in \mathbb{P}(TC)$

`typing_TCCore_Execute_Fail :` $TCCore\_Execute\_Fail \in \mathbb{P}(TC)$

`typing_TC_TransferData_Device_to_Core :` $TC\_TransferData\_Device\_to\_Core \in \mathbb{P}(TC)$

`typing_SendOkTC_Device_to_Core :` $SendOkTC\_Device\_to\_Core \in \mathbb{P}(TC)$

`typing_TCExecOk_ReplyCtrlTM :` $TCExecOk\_ReplyCtrlTM \in \mathbb{P}(TC)$

`typing_TCExecFail_ReplyCtrlTM :` $TCExecFail\_ReplyCtrlTM \in \mathbb{P}(TC)$

`M0_inv_ReceiveTC :` $ReceiveTC \subseteq TC$

`M1_inv_TCCheck_Ok_seq :` $TCCheck\_Ok \subseteq ReceiveTC$

`M1_inv_TCCheck_Fail_seq :` $TCCheck\_Fail \subseteq ReceiveTC$

`M1_inv_TCCheckFail_ReplyCtrlTM_seq :` $TCCheckFail\_ReplyCtrlTM \subseteq TCCheck\_Fail$

`M1_inv1 :` $TCCheck\_Ok \cap TCCheck\_Fail = \varnothing$

M2_inv_TCCore_Execute_Ok_sequencing : $TCCore\_Execute\_Ok \subseteq TCCheck\_Ok$

M2_inv_TCCore_Execute_Fail_sequencing : $TCCore\_Execute\_Fail \subseteq TCCheck\_Ok$

M3_inv_TCExecOk_ReplyCtrlTM_seq : $TCExecOk\_ReplyCtrlTM \subseteq TCCore\_Execute\_Ok \cup SendOkTC\_Device\_to\_Core$

M3_linv_TCExec_ReplyCtrlTM_seq : $TCExecFail\_ReplyCtrlTM \subseteq TCCore\_Execute\_Fail \cup SendFailTC\_Device\_to\_Core$

EVENTS

Initialisation

begin

act_ReceiveTC : $ReceiveTC := \varnothing$
act_TCCheck_Ok : $TCCheck\_Ok := \varnothing$
act_TCCore_Execute_Ok : $TCCore\_Execute\_Ok := \varnothing$
act_SendOkTC_Device_to_Core : $SendOkTC\_Device\_to\_Core := \varnothing$
act_TCCheck_Fail : $TCCheck\_Fail := \varnothing$
act_TCCore_Execute_Fail : $TCCore\_Execute\_Fail := \varnothing$
act_SendFailTC_Device_to_Core : $SendFailTC\_Device\_to\_Core := \varnothing$
act_TC_TransferData_Device_to_Core : $TC\_TransferData\_Device\_to\_Core := \varnothing$
act_TCValid_ReplyDataTM : $TCValid\_ReplyDataTM := \varnothing$
act_TCExecOk_ReplyCtrlTM : $TCExecOk\_ReplyCtrlTM := \varnothing$
act_TCExecFail_ReplyCtrlTM : $TCExecFail\_ReplyCtrlTM := \varnothing$
act_TCCheckFail_ReplyCtrlTM : $TCCheckFail\_ReplyCtrlTM := \varnothing$

end

Event $ReceiveTC \mathrel{\widehat{=}}$

any

tc

where

typing_tc : $tc \in TC$
grd_ReceiveTC : $tc \notin ReceiveTC$

then

act_ReceiveTC : $ReceiveTC := ReceiveTC \cup \{tc\}$

end

Event $TCCheck\_Ok \mathrel{\widehat{=}}$

any

tc

where

typing_tc : $tc \in TC$
grd_TCCheck_Ok_seq : $tc \in ReceiveTC$
grd_TCCheck_Ok : $tc \notin TCCheck\_Ok$
grd1 : $tc \notin TCCheck\_Fail$

then

act_TCCheck_Ok : $TCCheck\_Ok := TCCheck\_Ok \cup \{tc\}$

end

Event $TCCore\_Execute\_Ok \mathrel{\widehat{=}}$

any

tc

where

typing_tc : $tc \in TC$
grd_TCCore_Execute_Ok_seq : $tc \in TCCheck\_Ok$
grd_TCCore_Execute_Ok : $tc \notin TCCore\_Execute\_Ok$
grd2 : $tc \notin TCCore\_Execute\_Fail$
grd3 : $PID(tc) = csw$

then

act_TCCore_Execute_Ok : $TCCore\_Execute\_Ok := TCCore\_Execute\_Ok \cup \{tc\}$

end

Event $SendTC\_Core\_to\_Device \mathrel{\widehat{=}}$

any

tc

```
    where
        typing_tc :  $tc \in TC$
        grd_SendTC_Core_to_Device_seq :  $tc \in TCCheck\_Ok$
        grd_SendTC_Core_to_Device_xor1 :  $tc \notin TCCore\_Execute\_Ok$
        grd_SendTC_Core_to_Device_xor2 :  $tc \notin TCCore\_Execute\_Fail$
        grd1 :  $PID(tc) \in \{mixsc, mixst, sixsp, sixsx\}$
    then
        skip
    end
Event  $SendOkTC\_Device\_to\_Core \triangleq$
    any
        $tc$
    where
        typing_tc :  $tc \in TC$
        grd_SendOkTC_Device_to_Core :  $tc \notin SendOkTC\_Device\_to\_Core$
    then
        act_SendOkTC_Device_to_Core :  $SendOkTC\_Device\_to\_Core :=$
            $SendOkTC\_Device\_to\_Core \cup \{tc\}$
    end
Event  $TCCore\_Execute\_Fail \triangleq$
    any
        $tc$
    where
        typing_tc :  $tc \in TC$
        grd_TCCore_Execute_Fail_seq :  $tc \in TCCheck\_Ok$
        grd_TCCore_Execute_Fail :  $tc \notin TCCore\_Execute\_Fail$
        grd2 :  $tc \notin TCCore\_Execute\_Ok$
        grd3 :  $PID(tc) = csw$
    then
        act_TCCore_Execute_Fail :  $TCCore\_Execute\_Fail := TCCore\_Execute\_Fail \cup \{tc\}$
    end
Event  $SendFailTC\_Device\_to\_Core \triangleq$
    any
        $tc$
    where
        typing_tc :  $tc \in TC$
        grd_SendFailTC_Device_to_Core :  $tc \notin SendFailTC\_Device\_to\_Core$
    then
        act_SendFailTC_Device_to_Core :  $SendFailTC\_Device\_to\_Core :=$
            $SendFailTC\_Device\_to\_Core \cup \{tc\}$
    end
Event  $TCCheck\_Fail \triangleq$
    any
        $tc$
    where
        typing_tc :  $tc \in TC$
        grd_TCCheck_Fail_seq :  $tc \in ReceiveTC$
        grd_TCCheck_Fail :  $tc \notin TCCheck\_Ok$
        grd1 :  $tc \notin TCCheck\_Fail$
    then
        act_TCCheck_Fail :  $TCCheck\_Fail := TCCheck\_Fail \cup \{tc\}$
    end
Event  $TC\_GenerateData\_in\_Device \triangleq$
    any
        $tc$
    where
        typing_tc :  $tc \in TC$
        grd_TC_GenerateData_in_Device_seq : $tc \in TCCore\_Execute\_Ok \cup SendOkTC\_Device\_to\_Core$
        grd1 : $TC\_Type(tc) \in \{HK\_on\_TC, SCI\_on\_TC\}$
        grd2 : $PID(tc) \in \{mixsc, mixst, sixsp, sixsx\}$
    then
        skip
    end
Event $TC\_TransferData\_Device\_to\_Core$ $\widehat{=}$
    any
        $tc$
    where
        typing_tc : $tc \in TC$
        grd_TC_TransferData_Device_to_Core : $tc \notin TC\_TransferData\_Device\_to\_Core$
    then
        act_TC_TransferData_Device_to_Core : $TC\_TransferData\_Device\_to\_Core :=$
            $TC\_TransferData\_Device\_to\_Core \cup \{tc\}$
    end
Event $TCValid\_ReplyDataTM$ $\widehat{=}$
    any
        $tc$
    where
        typing_tc : $tc \in TC$
        grd_TCValid_ReplyDataTM_seq : $tc \in TC\_TransferData\_Device\_to\_Core$
        grd_TCValid_ReplyDataTM : $tc \notin TCValid\_ReplyDataTM$
    then
        act_TCValid_ReplyDataTM : $TCValid\_ReplyDataTM := TCValid\_ReplyDataTM \cup \{tc\}$
    end
Event $TCExecOk\_ReplyCtrlTM$ $\widehat{=}$
    any
        $tc$
    where
        typing_tc : $tc \in TC$
        grd_TCExecOk_ReplyCtrlTM_seq : $tc \in$
            $TCCore\_Execute\_Ok \cup SendOkTC\_Device\_to\_Core$
        grd_TCExecOk_ReplyCtrlTM : $tc \notin TCExecOk\_ReplyCtrlTM$
    then
        act_TCExecOk_ReplyCtrlTM : $TCExecOk\_ReplyCtrlTM :=$
            $TCExecOk\_ReplyCtrlTM \cup \{tc\}$
    end
Event $TCExecFail\_ReplyCtrlTM$ $\widehat{=}$
    any
        $tc$
    where
        typing_tc : $tc \in TC$
        grd_TCExecFail_ReplyCtrlTM_seq : $tc \in$
            $TCCore\_Execute\_Fail \cup SendFailTC\_Device\_to\_Core$
        grd_TCExecFail_ReplyCtrlTM : $tc \notin TCExecFail\_ReplyCtrlTM$
    then
        act_TCExecFail_ReplyCtrlTM : $TCExecFail\_ReplyCtrlTM :=$
            $TCExecFail\_ReplyCtrlTM \cup \{tc\}$
    end
Event $TCCheckFail\_ReplyCtrlTM$ $\widehat{=}$
    any
        $tc$
    where
        typing_tc : $tc \in TC$
        grd_TCCheckFail_ReplyCtrlTM_seq : $tc \in TCCheck\_Fail$
        grd_TCCheckFail_ReplyCtrlTM : $tc \notin TCCheckFail\_ReplyCtrlTM$
```

then

`act_TCCheckFail_ReplyCtrlTM :` $TCCheckFail\_ReplyCtrlTM :=$
$TCCheckFail\_ReplyCtrlTM \cup \{tc\}$

end

END

B.5.3 1st Refinement

B.5.3.1 Context: *Context_M4*

CONTEXT Context_M4

EXTENDS Context_M3

SETS

`TM`

END

B.5.3.2 Machine: M4

MACHINE M4

REFINES M3

SEES Context_M4

VARIABLES

ReceiveTC, TCCheck_Ok, TCCore_Execute_Ok, SendOkTC_Device_to_Core, TCCheck_Fail, TCCore_Execute_Fail, SendFailTC_Device_to_Core, TC_TransferData_Device_to_Core, TCValid_ProcessCtrlTM, TCValid_CompleteCtrlTM, TCExecOk_ProcessCtrlTM, TCExecOk_CompleteCtrlTM, TCExecFail_ProcessCtrlTM, TCExecFail_CompleteCtrlTM, TCCheckFail_ProcessCtrlTM, TCCheckFail_CompleteCtrlTM

INVARIANTS

`inv_TCValid_ProcessCtrlTM :` $TCValid\_ProcessCtrlTM \subseteq TC \times TM$

`inv_TCValid_ProcessCtrlTM_seq :` $dom(TCValid\_ProcessCtrlTM) \subseteq$
$TC\_TransferData\_Device\_to\_Core$

`inv_TCValid_CompleteCtrlTM_seq :` $TCValid\_CompleteCtrlTM \subseteq$
$dom(TCValid\_ProcessCtrlTM)$

`inv_TCExecOk_ProcessCtrlTM :` $TCExecOk\_ProcessCtrlTM \subseteq TC \times TM$

`inv_TCExecOk_ProcessCtrlTM_seq :` $dom(TCExecOk\_ProcessCtrlTM) \subseteq$
$TCCore\_Execute\_Ok \cup SendOkTC\_Device\_to\_Core$

`inv1 :` $\forall tc \cdot finite(TCExecOk\_ProcessCtrlTM[\{tc\}])$
manually, to prove (inv_TCExecOk_ProcessCtrlTM_one/WD)

`inv_TCExecOk_ProcessCtrlTM_one :` $\forall tc \cdot card(TCExecOk\_ProcessCtrlTM[\{tc\}]) \leq 1$

`inv_TCExecOk_CompleteCtrlTM_seq :` $TCExecOk\_CompleteCtrlTM \subseteq dom(TCExecOk\_ProcessCtrlTM)$

`inv_TCExecFail_ProcessCtrlTM :` $TCExecFail\_ProcessCtrlTM \subseteq TC \times TM$

`inv_TCExecFail_ProcessCtrlTM_seq :` $dom(TCExecFail\_ProcessCtrlTM) \subseteq$
$TCCore\_Execute\_Fail \cup SendFailTC\_Device\_to\_Core$

`inv2 :` $\forall tc \cdot finite(TCExecFail\_ProcessCtrlTM[\{tc\}])$ \\ manually

`inv_TCExecFail_ProcessCtrlTM_one :` $\forall tc \cdot card(TCExecFail\_ProcessCtrlTM[\{tc\}]) \leq 1$

`inv_TCExecFail_CompleteCtrlTM_seq :` $TCExecFail\_CompleteCtrlTM \subseteq$
$dom(TCExecFail\_ProcessCtrlTM)$

`inv_TCCheckFail_ProcessCtrlTM :` $TCCheckFail\_ProcessCtrlTM \subseteq TC \times TM$

`inv_TCCheckFail_ProcessCtrlTM_seq :` $dom(TCCheckFail\_ProcessCtrlTM) \subseteq TCCheck\_Fail$

`inv3 :` $\forall tc \cdot finite(TCCheckFail\_ProcessCtrlTM[\{tc\}])$ \\ manually

`inv_TCCheckFail_ProcessCtrlTM_one :` $\forall tc \cdot card(TCCheckFail\_ProcessCtrlTM[\{tc\}]) \leq 1$

```
        inv_TCCheckFail_CompleteCtrlTM_seq : TCCheckFail_CompleteCtrlTM ⊆
                dom(TCCheckFail_ProcessCtrlTM)
        inv_TCValid_CompleteCtrlTM_gluing : TCValid_CompleteCtrlTM = TCValid_ReplyDataTM
        inv_TCExecOk_CompleteCtrlTM_gluing : TCExecOk_CompleteCtrlTM = TCExecOk_ReplyCtrlTM
        inv_TCExecFail_CompleteCtrlTM_gluing : TCExecFail_CompleteCtrlTM =
                TCExecFail_ReplyCtrlTM
        inv_TCCheckFail_CompleteCtrlTM_gluing : TCCheckFail_CompleteCtrlTM =
                TCCheckFail_ReplyCtrlTM
EVENTS
Initialisation
        begin
                act_ReceiveTC : ReceiveTC := ∅
                act_TCCheck_Ok : TCCheck_Ok := ∅
                act_TCCore_Execute_Ok : TCCore_Execute_Ok := ∅
                act_SendOkTC_Device_to_Core : SendOkTC_Device_to_Core := ∅
                act_TCCheck_Fail : TCCheck_Fail := ∅
                act_TCCore_Execute_Fail : TCCore_Execute_Fail := ∅
                act_SendFailTC_Device_to_Core : SendFailTC_Device_to_Core := ∅
                act_TC_TransferData_Device_to_Core : TC_TransferData_Device_to_Core := ∅
                act_TCValid_ProcessCtrlTM : TCValid_ProcessCtrlTM := ∅
                act_TCValid_CompleteCtrlTM : TCValid_CompleteCtrlTM := ∅
                act_TCExecOk_ProcessCtrlTM : TCExecOk_ProcessCtrlTM := ∅
                act_TCExecOk_CompleteCtrlTM : TCExecOk_CompleteCtrlTM := ∅
                act_TCExecFail_ProcessCtrlTM : TCExecFail_ProcessCtrlTM := ∅
                act_TCExecFail_CompleteCtrlTM : TCExecFail_CompleteCtrlTM := ∅
                act_TCCheckFail_ProcessCtrlTM : TCCheckFail_ProcessCtrlTM := ∅
                act_TCCheckFail_CompleteCtrlTM : TCCheckFail_CompleteCtrlTM := ∅
        end
Event   ReceiveTC ≙
extends ReceiveTC
        any
                tc
        where
                typing_tc : tc ∈ TC
                grd_ReceiveTC : tc ∉ ReceiveTC
        then
                act_ReceiveTC : ReceiveTC := ReceiveTC ∪ {tc}
        end
Event   TCCheck_Ok ≙
extends TCCheck_Ok
        any
                tc
        where
                typing_tc : tc ∈ TC
                grd_TCCheck_Ok_seq : tc ∈ ReceiveTC
                grd_TCCheck_Ok : tc ∉ TCCheck_Ok
                grd1 : tc ∉ TCCheck_Fail
        then
                act_TCCheck_Ok : TCCheck_Ok := TCCheck_Ok ∪ {tc}
        end
Event   TCCore_Execute_Ok ≙
extends TCCore_Execute_Ok
        any
                tc
        where
                typing_tc : tc ∈ TC
        grd_TCCore_Execute_Ok_seq : tc ∈ TCCheck_Ok
        grd_TCCore_Execute_Ok : tc ∉ TCCore_Execute_Ok
        grd2 : tc ∉ TCCore_Execute_Fail
        grd3 : PID(tc) = csw
    then
        act_TCCore_Execute_Ok : TCCore_Execute_Ok := TCCore_Execute_Ok ∪ {tc}
    end
Event  SendTC_Core_to_Device ≙
extends  SendTC_Core_to_Device
    any
        tc
    where
        typing_tc : tc ∈ TC
        grd_SendTC_Core_to_Device_seq : tc ∈ TCCheck_Ok
        grd_SendTC_Core_to_Device_xor1 : tc ∉ TCCore_Execute_Ok
        grd_SendTC_Core_to_Device_xor2 : tc ∉ TCCore_Execute_Fail
        grd1 : PID(tc) ∈ {mixsc, mixst, sixsp, sixsx}
    then
        skip
    end
Event  SendOkTC_Device_to_Core ≙
extends  SendOkTC_Device_to_Core
    any
        tc
    where
        typing_tc : tc ∈ TC
        grd_SendOkTC_Device_to_Core : tc ∉ SendOkTC_Device_to_Core
    then
        act_SendOkTC_Device_to_Core : SendOkTC_Device_to_Core :=
            SendOkTC_Device_to_Core ∪ {tc}
    end
Event  TCCore_Execute_Fail ≙
extends  TCCore_Execute_Fail
    any
        tc
    where
        typing_tc : tc ∈ TC
        grd_TCCore_Execute_Fail_seq : tc ∈ TCCheck_Ok
        grd_TCCore_Execute_Fail : tc ∉ TCCore_Execute_Fail
        grd2 : tc ∉ TCCore_Execute_Ok
        grd3 : PID(tc) = csw
    then
        act_TCCore_Execute_Fail : TCCore_Execute_Fail := TCCore_Execute_Fail ∪ {tc}
    end
Event  SendFailTC_Device_to_Core ≙
extends  SendFailTC_Device_to_Core
    any
        tc
    where
        typing_tc : tc ∈ TC
        grd_SendFailTC_Device_to_Core : tc ∉ SendFailTC_Device_to_Core
    then
        act_SendFailTC_Device_to_Core : SendFailTC_Device_to_Core := SendFailTC_Device_to_Core
            ∪ {tc}
    end
Event  TCCheck_Fail ≙
extends  TCCheck_Fail
```

```
        any
              tc
        where
              typing_tc :  $tc \in TC$
              grd_TCCheck_Fail_seq :  $tc \in ReceiveTC$
              grd_TCCheck_Fail :  $tc \notin TCCheck\_Ok$
              grd1 :  $tc \notin TCCheck\_Fail$
        then
              act_TCCheck_Fail :  $TCCheck\_Fail := TCCheck\_Fail \cup \{tc\}$
        end
Event   TC_GenerateData_in_Device ≙
extends   TC_GenerateData_in_Device
        any
              tc
        where
              typing_tc :  $tc \in TC$
              grd_TC_GenerateData_in_Device_seq :  $tc \in TCCore\_Execute\_Ok \cup SendOkTC\_Device\_to\_Core$
              grd1 :  $TC\_Type(tc) \in \{HK\_on\_TC, SCI\_on\_TC\}$
              grd2 :  $PID(tc) \in \{mixsc, mixst, sixsp, sixsx\}$
        then
              skip
        end
Event   TC_TransferData_Device_to_Core ≙
extends   TC_TransferData_Device_to_Core
        any
              tc
        where
              typing_tc :  $tc \in TC$
              grd_TC_TransferData_Device_to_Core :  $tc \notin TC\_TransferData\_Device\_to\_Core$
        then
              act_TC_TransferData_Device_to_Core :  $TC\_TransferData\_Device\_to\_Core :=$
                    $TC\_TransferData\_Device\_to\_Core \cup \{tc\}$
        end
Event   TCValid_ProcessCtrlTM ≙
        any
              tc
              tm
        where
              grd_TCValid_ProcessCtrlTM_seq :  $tc \in TC\_TransferData\_Device\_to\_Core$
              grd_TCValid_ProcessCtrlTM :  $tc \mapsto tm \notin TCValid\_ProcessCtrlTM$
        then
              act_TCValid_ProcessCtrlTM :  $TCValid\_ProcessCtrlTM := TCValid\_ProcessCtrlTM \cup \{tc \mapsto tm\}$
        end
Event   TCValid_CompleteCtrlTM ≙
refines   TCValid_ReplyDataTM
        any
              tc
        where
              grd_TCValid_CompleteCtrlTM_seq :  $tc \in dom(TCValid\_ProcessCtrlTM)$
              grd_TCValid_CompleteCtrlTM :  $tc \notin TCValid\_CompleteCtrlTM$
        then
              act_TCValid_CompleteCtrlTM :  $TCValid\_CompleteCtrlTM := TCValid\_CompleteCtrlTM \cup \{tc\}$
        end
Event   TCExecOk_ProcessCtrlTM ≙
        any
              tc
              tm
        where
              grd_TCExecOk_ProcessCtrlTM_seq :  $tc \in TCCore\_Execute\_Ok \cup SendOkTC\_Device\_to\_Core$
              grd_TCExecOk_ProcessCtrlTM :  $tc \mapsto tm \notin TCExecOk\_ProcessCtrlTM$
              grd_TCExecOk_ProcessCtrlTM_one :  $tc \notin dom(TCExecOk\_ProcessCtrlTM)$
        then
              act_TCExecOk_ProcessCtrlTM :  $TCExecOk\_ProcessCtrlTM :=$
                    $TCExecOk\_ProcessCtrlTM \cup \{tc \mapsto tm\}$
        end
Event   $TCExecOk\_CompleteCtrlTM \;\widehat{=}$
refines  $TCExecOk\_ReplyCtrlTM$
        any
              $tc$
        where
              grd_TCExecOk_CompleteCtrlTM_seq :  $tc \in dom(TCExecOk\_ProcessCtrlTM)$
              grd_TCExecOk_CompleteCtrlTM :  $tc \notin TCExecOk\_CompleteCtrlTM$
        then
              act_TCExecOk_CompleteCtrlTM :  $TCExecOk\_CompleteCtrlTM := TCExecOk\_CompleteCtrlTM \cup$
                    $\{tc\}$
        end
Event   $TCExecFail\_ProcessCtrlTM \;\widehat{=}$
        any
              $tc$
              $tm$
        where
              grd_TCExecFail_ProcessCtrlTM_seq :  $tc \in TCCore\_Execute\_Fail \cup SendFailTC\_Device\_to\_Core$
              grd_TCExecFail_ProcessCtrlTM :  $tc \mapsto tm \notin TCExecFail\_ProcessCtrlTM$
              grd_TCExecFail_ProcessCtrlTM_one :  $tc \notin dom(TCExecFail\_ProcessCtrlTM)$
        then
              act_TCExecFail_ProcessCtrlTM :  $TCExecFail\_ProcessCtrlTM := TCExecFail\_ProcessCtrlTM \cup$
                    $\{tc \mapsto tm\}$
        end
Event   $TCExecFail\_CompleteCtrlTM \;\widehat{=}$
refines  $TCExecFail\_ReplyCtrlTM$
        any
              $tc$
        where
              grd_TCExecFail_CompleteCtrlTM_seq :  $tc \in dom(TCExecFail\_ProcessCtrlTM)$
              grd_TCExecFail_CompleteCtrlTM :  $tc \notin TCExecFail\_CompleteCtrlTM$
        then
              act_TCExecFail_CompleteCtrlTM :  $TCExecFail\_CompleteCtrlTM :=$
                    $TCExecFail\_CompleteCtrlTM \cup \{tc\}$
        end
Event   $TCCheckFail\_ProcessCtrlTM \;\widehat{=}$
        any
              $tc$
              $tm$
        where
              grd_TCCheckFail_ProcessCtrlTM_seq :  $tc \in TCCheck\_Fail$
              grd_TCCheckFail_ProcessCtrlTM :  $tc \mapsto tm \notin TCCheckFail\_ProcessCtrlTM$
              grd_TCCheckFail_ProcessCtrlTM_one :  $tc \notin dom(TCCheckFail\_ProcessCtrlTM)$
        then
              act_TCCheckFail_ProcessCtrlTM :  $TCCheckFail\_ProcessCtrlTM :=$
                    $TCCheckFail\_ProcessCtrlTM$
                    $\cup \{tc \mapsto tm\}$
        end
Event   $TCCheckFail\_CompleteCtrlTM \;\widehat{=}$
refines  $TCCheckFail\_ReplyCtrlTM$
```

any
  tc
where
  grd_TCCheckFail_CompleteCtrlTM_seq : $tc \in dom(TCCheckFail\_ProcessCtrlTM)$
  grd_TCCheckFail_CompleteCtrlTM : $tc \notin TCCheckFail\_CompleteCtrlTM$
then
  act_TCCheckFail_CompleteCtrlTM : $TCCheckFail\_CompleteCtrlTM :=$
    $TCCheckFail\_CompleteCtrlTM \cup \{tc\}$
end

END

B.5.4 2nd Refinement

B.5.4.1 Context: *Context_M5*

CONTEXT Context_M5

EXTENDS Context_M4

SETS
  TM_Types_Set

CONSTANTS
  Exec_nok_TM, Exec_ok_TM, SCI_TM, HK_TM, TM_Type, Check_nok_TM

AXIOMS
  axm3 : $TM\_Types\_Set = \{Check\_nok\_TM, Exec\_ok\_TM, Exec\_nok\_TM, HK\_TM, SCI\_TM\}$
  axm4 : $TM\_Type \in TM \rightarrow TM\_Types\_Set$

END

B.5.4.2 Machine: M5

MACHINE M5

REFINES M4

SEES Context_M5

VARIABLES
  ReceiveTC, TCCheck_Ok, TCCore_Execute_Ok, SendOkTC_Device_to_Core, TCCheck_Fail, TCCore_Execute_Fail, SendFailTC_Device_to_Core, TC_TransferData_Device_to_Core, Produce_DataTM, Send_DataTM, TCValid_CompleteCtrlTM, Produce_ExecOkTM, Send_ExecOkTM, TCExecOk_CompleteCtrlTM, Produce_ExecFailTM, Send_ExecFailTM, TCExecFail_CompleteCtrlTM, Produce_CheckFailTM, Send_CheckFailTM, TCCheckFail_CompleteCtrlTM

INVARIANTS
  inv_Produce_DataTM : $Produce\_DataTM \subseteq TC \times TM$
  inv_Produce_DataTM_seq : $dom(Produce\_DataTM) \subseteq TC\_TransferData\_Device\_to\_Core$
  inv_Send_DataTM_seq : $Send\_DataTM \subseteq Produce\_DataTM$
  inv_TCValid_CompleteCtrlTM_seq : $TCValid\_CompleteCtrlTM \subseteq dom(Produce\_DataTM)$
  inv_Produce_ExecOkTM : $Produce\_ExecOkTM \subseteq TC \times TM$
  inv_Produce_ExecOkTM_seq : $dom(Produce\_ExecOkTM) \subseteq$
    $TCCore\_Execute\_Ok \cup SendOkTC\_Device\_to\_Core$
  inv1 : $\forall tc \cdot finite(Produce\_ExecOkTM[\{tc\}])$
  inv_Produce_ExecOkTM_one : $\forall tc \cdot card(Produce\_ExecOkTM[\{tc\}]) \leq 1$
  inv_Send_ExecOkTM_seq : $Send\_ExecOkTM \subseteq Produce\_ExecOkTM$
  inv_TCExecOk_CompleteCtrlTM_seq : $TCExecOk\_CompleteCtrlTM \subseteq$
    $dom(Produce\_ExecOkTM)$
  inv_Produce_ExecFailTM : $Produce\_ExecFailTM \subseteq TC \times TM$

inv_Produce_ExecFailTM_seq : $dom(Produce\_ExecFailTM) \subseteq$
$TCCore\_Execute\_Fail \cup SendFailTC\_Device\_to\_Core$

inv2 : $\forall tc \cdot finite(Produce\_ExecFailTM[\{tc\}])$

inv_Produce_ExecFailTM_one : $\forall tc \cdot card(Produce\_ExecFailTM[\{tc\}]) \leq 1$

inv_Send_ExecFailTM_seq : $Send\_ExecFailTM \subseteq Produce\_ExecFailTM$

inv_TCExecFail_CompleteCtrlTM_seq : $TCExecFail\_CompleteCtrlTM \subseteq dom(Produce\_ExecFailTM)$

inv_Produce_CheckFailTM : $Produce\_CheckFailTM \subseteq TC \times TM$

inv_Produce_CheckFailTM_seq : $dom(Produce\_CheckFailTM) \subseteq TCCheck\_Fail$

inv3 : $\forall tc \cdot finite(Produce\_CheckFailTM[\{tc\}])$

inv_Produce_CheckFailTM_one : $\forall tc \cdot card(Produce\_CheckFailTM[\{tc\}]) \leq 1$

inv_Send_CheckFailTM_seq : $Send\_CheckFailTM \subseteq Produce\_CheckFailTM$

inv_TCCheckFail_CompleteCtrlTM_seq : $TCCheckFail\_CompleteCtrlTM \subseteq$
$dom(Produce\_CheckFailTM)$

inv_Produce_DataTM_gluing : $Produce\_DataTM = TCValid\_ProcessCtrlTM$

inv_Produce_ExecOkTM_gluing : $Produce\_ExecOkTM = TCExecOk\_ProcessCtrlTM$

inv_Produce_ExecFailTM_gluing : $Produce\_ExecFailTM = TCExecFail\_ProcessCtrlTM$

inv_Produce_CheckFailTM_gluing : $Produce\_CheckFailTM = TCCheckFail\_ProcessCtrlTM$

EVENTS

Initialisation

begin

act_ReceiveTC : $ReceiveTC := \varnothing$
act_TCCheck_Ok : $TCCheck\_Ok := \varnothing$
act_TCCore_Execute_Ok : $TCCore\_Execute\_Ok := \varnothing$
act_SendOkTC_Device_to_Core : $SendOkTC\_Device\_to\_Core := \varnothing$
act_TCCheck_Fail : $TCCheck\_Fail := \varnothing$
act_TCCore_Execute_Fail : $TCCore\_Execute\_Fail := \varnothing$
act_SendFailTC_Device_to_Core : $SendFailTC\_Device\_to\_Core := \varnothing$
act_TC_TransferData_Device_to_Core : $TC\_TransferData\_Device\_to\_Core := \varnothing$
act_Produce_DataTM : $Produce\_DataTM := \varnothing$
act_Send_DataTM : $Send\_DataTM := \varnothing$
act_TCValid_CompleteCtrlTM : $TCValid\_CompleteCtrlTM := \varnothing$
act_Produce_ExecOkTM : $Produce\_ExecOkTM := \varnothing$
act_Send_ExecOkTM : $Send\_ExecOkTM := \varnothing$
act_TCExecOk_CompleteCtrlTM : $TCExecOk\_CompleteCtrlTM := \varnothing$
act_Produce_ExecFailTM : $Produce\_ExecFailTM := \varnothing$
act_Send_ExecFailTM : $Send\_ExecFailTM := \varnothing$
act_TCExecFail_CompleteCtrlTM : $TCExecFail\_CompleteCtrlTM := \varnothing$
act_Produce_CheckFailTM : $Produce\_CheckFailTM := \varnothing$
act_TCCheckFail_CompleteCtrlTM : $TCCheckFail\_CompleteCtrlTM := \varnothing$
act_Send_CheckFailTM : $Send\_CheckFailTM := \varnothing$

end

Event $ReceiveTC \widehat{=}$

extends $ReceiveTC$

any

tc

where

typing_tc : tc ∈ TC
grd_ReceiveTC : tc ∉ ReceiveTC

then

act_ReceiveTC : ReceiveTC := ReceiveTC ∪ {tc}

end

Event $TCCheck\_Ok \widehat{=}$

extends $TCCheck\_Ok$

any

tc

```
        where
              typing_tc :  tc ∈ TC
              grd_TCCheck_Ok_seq :  tc ∈ ReceiveTC
              grd_TCCheck_Ok :  tc ∉ TCCheck_Ok
              grd1 :  tc ∉ TCCheck_Fail
        then
              act_TCCheck_Ok :  TCCheck_Ok := TCCheck_Ok ∪ {tc}
        end
Event    TCCore_Execute_Ok ≙
extends  TCCore_Execute_Ok
        any
              tc
        where
              typing_tc :  tc ∈ TC
              grd_TCCore_Execute_Ok_seq :  tc ∈ TCCheck_Ok
              grd_TCCore_Execute_Ok :  tc ∉ TCCore_Execute_Ok
              grd2 :  tc ∉ TCCore_Execute_Fail
              grd3 :  PID(tc) = csw
        then
              act_TCCore_Execute_Ok :  TCCore_Execute_Ok := TCCore_Execute_Ok ∪ {tc}
        end
Event    SendTC_Core_to_Device ≙
extends  SendTC_Core_to_Device
        any
              tc
        where
              typing_tc :  tc ∈ TC
              grd_SendTC_Core_to_Device_seq :  tc ∈ TCCheck_Ok
              grd_SendTC_Core_to_Device_xor1 :  tc ∉ TCCore_Execute_Ok
              grd_SendTC_Core_to_Device_xor2 :  tc ∉ TCCore_Execute_Fail
              grd1 :  PID(tc) ∈ {mixsc, mixst, sixsp, sixsx}
        then
              skip
        end
Event    SendOkTC_Device_to_Core ≙
extends  SendOkTC_Device_to_Core
        any
              tc
        where
              typing_tc :  tc ∈ TC
              grd_SendOkTC_Device_to_Core :  tc ∉ SendOkTC_Device_to_Core
        then
              act_SendOkTC_Device_to_Core :  SendOkTC_Device_to_Core := SendOkTC_Device_to_Core ∪ {tc}
        end
Event    TCCore_Execute_Fail ≙
extends  TCCore_Execute_Fail
        any
              tc
        where
              typing_tc :  tc ∈ TC
              grd_TCCore_Execute_Fail_seq :  tc ∈ TCCheck_Ok
              grd_TCCore_Execute_Fail :  tc ∉ TCCore_Execute_Fail
              grd2 :  tc ∉ TCCore_Execute_Ok
              grd3 :  PID(tc) = csw
        then
              act_TCCore_Execute_Fail :  TCCore_Execute_Fail := TCCore_Execute_Fail ∪ {tc}
        end
```

```
Event   SendFailTC_Device_to_Core ≙
extends SendFailTC_Device_to_Core
     any
          tc
     where
          typing_tc : tc ∈ TC
          grd_SendFailTC_Device_to_Core : tc ∉ SendFailTC_Device_to_Core
     then
          act_SendFailTC_Device_to_Core : SendFailTC_Device_to_Core := SendFailTC_Device_to_Core∪
               {tc}
     end
Event   TCCheck_Fail ≙
extends TCCheck_Fail
     any
          tc
     where
          typing_tc : tc ∈ TC
          grd_TCCheck_Fail_seq : tc ∈ ReceiveTC
          grd_TCCheck_Fail : tc ∉ TCCheck_Ok
          grd1 : tc ∉ TCCheck_Fail
     then
          act_TCCheck_Fail : TCCheck_Fail := TCCheck_Fail ∪ {tc}
     end
Event   TC_GenerateData_in_Device ≙
extends TC_GenerateData_in_Device
     any
          tc
     where
          typing_tc : tc ∈ TC
          grd_TC_GenerateData_in_Device_seq : tc ∈ TCCore_Execute_Ok ∪ SendOkTC_Device_to_Core
          grd1 : TC_Type(tc) ∈ {HK_on_TC, SCI_on_TC}
          grd2 : PID(tc) ∈ {mixsc, mixst, sixsp, sixsx}
     then
          skip
     end
Event   TC_TransferData_Device_to_Core ≙
extends TC_TransferData_Device_to_Core
     any
          tc
     where
          typing_tc : tc ∈ TC
          grd_TC_TransferData_Device_to_Core : tc ∉ TC_TransferData_Device_to_Core
     then
          act_TC_TransferData_Device_to_Core : TC_TransferData_Device_to_Core :=
               TC_TransferData_Device_to_Core ∪ {tc}
     end
Event   Produce_DataTM ≙
refines TCValid_ProcessCtrlTM
     any
          tc
          tm
     where
          grd_Produce_DataTM_seq : tc ∈ TC_TransferData_Device_to_Core
          grd_Produce_DataTM : tc ↦ tm ∉ Produce_DataTM
          grd1 : TM_Type(tm) ∈ {HK_TM, SCI_TM}
     then
          act_Produce_DataTM : Produce_DataTM := Produce_DataTM ∪ {tc ↦ tm}
```

```
        end
Event   Send_DataTM ≙
        any
                tc
                tm
        where
                grd_Send_DataTM_sequencing : tc ↦ tm ∈ Produce_DataTM \ Send_DataTM
        then
                act_Send_DataTM : Send_DataTM := Send_DataTM ∪ {tc ↦ tm}
        end
Event   TCValid_CompleteCtrlTM ≙
refines TCValid_CompleteCtrlTM
        any
                tc
        where
                grd_TCValid_CompleteCtrlTM_seq : tc ∈ dom(Produce_DataTM)
                grd_TCValid_CompleteCtrlTM : tc ∉ TCValid_CompleteCtrlTM
        then
                act_TCValid_CompleteCtrlTM : TCValid_CompleteCtrlTM := TCValid_CompleteCtrlTM ∪ {tc}
        end
Event   Produce_ExecOkTM ≙
refines TCExecOk_ProcessCtrlTM
        any
                tc
                tm
        where
                grd_Produce_ExecOkTM_seq : tc ∈ TCCore_Execute_Ok ∪ SendOkTC_Device_to_Core
                grd_Produce_ExecOkTM : tc ↦ tm ∉ Produce_ExecOkTM
                grd_Produce_ExecOkTM_one : tc ∉ dom(Produce_ExecOkTM)
                grd1 : TM_Type(tm) = Exec_ok_TM
        then
                act_Produce_ExecOkTM : Produce_ExecOkTM := Produce_ExecOkTM ∪ {tc ↦ tm}
        end
Event   Send_ExecOkTM ≙
        any
                tc
                tm
        where
                grd_Send_ExecOkTM_seq : tc ↦ tm ∈ Produce_ExecOkTM \ Send_ExecOkTM
                grd_Send_ExecOkTM : tc ↦ tm ∉ Send_ExecOkTM
        then
                act_Send_ExecOkTM : Send_ExecOkTM := Send_ExecOkTM ∪ {tc ↦ tm}
        end
Event   TCExecOk_CompleteCtrlTM ≙
refines TCExecOk_CompleteCtrlTM
        any
                tc
        where
                grd_TCExecOk_CompleteCtrlTM_seq : tc ∈ dom(Produce_ExecOkTM)
                grd_TCExecOk_CompleteCtrlTM : tc ∉ TCExecOk_CompleteCtrlTM
        then
                act_TCExecOk_CompleteCtrlTM : TCExecOk_CompleteCtrlTM := TCExecOk_CompleteCtrlTM ∪
                        {tc}
        end
Event   Produce_ExecFailTM ≙
refines TCExecFail_ProcessCtrlTM
```

any
tc
tm
where
`grd_Produce_ExecFailTM_seq :` $tc \in TCCore\_Execute\_Fail \cup SendFailTC\_Device\_to\_Core$
`grd_Produce_ExecFailTM :` $tc \mapsto tm \notin Produce\_ExecFailTM$
`grd_Produce_ExecFailTM_one :` $tc \notin dom(Produce\_ExecFailTM)$
`grd1 :` $TM\_Type(tm) = Exec\_nok\_TM$
then
`act_Produce_ExecFailTM :` $Produce\_ExecFailTM := Produce\_ExecFailTM \cup \{tc \mapsto tm\}$
end

Event $Send\_ExecFailTM \mathrel{\widehat{=}}$
any
tc
tm
where
`grd_Send_ExecFailTM_seq :` $tc \mapsto tm \in Produce\_ExecFailTM$
`grd_Send_ExecFailTM :` $tc \mapsto tm \notin Send\_ExecFailTM$
then
`act_Send_ExecFailTM :` $Send\_ExecFailTM := Send\_ExecFailTM \cup \{tc \mapsto tm\}$
end

Event $TCExecFail\_CompleteCtrlTM \mathrel{\widehat{=}}$
refines $TCExecFail\_CompleteCtrlTM$
any
tc
where
`grd_TCExecFail_CompleteCtrlTM_seq :` $tc \in dom(Produce\_ExecFailTM)$
`grd_TCExecFail_CompleteCtrlTM :` $tc \notin TCExecFail\_CompleteCtrlTM$
then
`act_TCExecFail_CompleteCtrlTM :` $TCExecFail\_CompleteCtrlTM :=$
$TCExecFail\_CompleteCtrlTM \cup \{tc\}$
end

Event $Produce\_CheckFailTM \mathrel{\widehat{=}}$
refines $TCCheckFail\_ProcessCtrlTM$
any
tc
tm
where
`grd_Produce_CheckFailTM_seq :` $tc \in TCCheck\_Fail$
`grd_Produce_CheckFailTM :` $tc \mapsto tm \notin Produce\_CheckFailTM$
`grd_Produce_CheckFailTM_one :` $tc \notin dom(Produce\_CheckFailTM)$
`grd1 :` $TM\_Type(tm) = Check\_nok\_TM$
then
`act_Produce_CheckFailTM :` $Produce\_CheckFailTM := Produce\_CheckFailTM \cup \{tc \mapsto tm\}$
end

Event $Send\_CheckFailTM \mathrel{\widehat{=}}$
any
tc
tm
where
`grd_Send_CheckFailTM_seq :` $tc \mapsto tm \in Produce\_CheckFailTM$
`grd_Send_CheckFailTM_sequencing :` $tc \mapsto tm \notin Send\_CheckFailTM$
then
`act_Send_CheckFailTM :` $Send\_CheckFailTM := Send\_CheckFailTM \cup \{tc \mapsto tm\}$
end

Event $TCCheckFail\_CompleteCtrlTM \mathrel{\widehat{=}}$
refines $TCCheckFail\_CompleteCtrlTM$

```
    any
        $tc$
    where
        grd_TCCheckFail_CompleteCtrlTM_seq : $tc \in dom(Produce\_CheckFailTM)$
        grd_TCCheckFail_CompleteCtrlTM : $tc \notin TCCheckFail\_CompleteCtrlTM$
    then
        act_TCCheckFail_CompleteCtrlTM : $TCCheckFail\_CompleteCtrlTM :=$
            $TCCheckFail\_CompleteCtrlTM \cup \{tc\}$
    end
END
```

B.6 Device Sub-model

B.6.1 Context: *Context_M3*

```
CONTEXT  Context_M3
SETS
    TC, DATA, TC_Types_Set, PIDS
CONSTANTS
    TC_Type, HK_on_TC, SCI_on_TC, PID, mixsc, mixst, sixsp, sixsx, csw
AXIOMS
    typing_TC_Type : $TC\_Type \in \mathbb{P}(TC \times TC\_Types\_Set)$
    typing_HK_on_TC : $HK\_on\_TC \in TC\_Types\_Set$
    typing_SCI_on_TC : $SCI\_on\_TC \in TC\_Types\_Set$
    typing_PID : $PID \in \mathbb{P}(TC \times PIDS)$
    typing_mixsc : $mixsc \in PIDS$
    typing_mixst : $mixst \in PIDS$
    typing_sixsp : $sixsp \in PIDS$
    typing_sixsx : $sixsx \in PIDS$
    typing_csw : $csw \in PIDS$
    C0_axm2 : $TC\_Type \in TC \rightarrow TC\_Types\_Set$
    C1_axm1 : $partition(PIDS, \{csw\}, \{mixsc\}, \{mixst\}, \{sixsp\}, \{sixsx\})$
    C1_axm2 : $PID \in TC \rightarrow PIDS$
END
```

B.6.2 Machine: M3

```
MACHINE  M3
SEES  Context_M3
VARIABLES
    CheckTC_in_Device_Ok, CheckTC_in_Device_Fail, TC_GenerateData_in_Device,
    SendTC_Core_to_Device
INVARIANTS
    typing_TC_GenerateData_in_Device : $TC\_GenerateData\_in\_Device \in \mathbb{P}(TC \times DATA)$
    typing_CheckTC_in_Device_Ok : $CheckTC\_in\_Device\_Ok \in \mathbb{P}(TC)$
    typing_SendTC_Core_to_Device : $SendTC\_Core\_to\_Device \in \mathbb{P}(TC)$
    typing_CheckTC_in_Device_Fail : $CheckTC\_in\_Device\_Fail \in \mathbb{P}(TC)$
    M3_inv_CheckTC_in_Device_Ok_seq : $CheckTC\_in\_Device\_Ok \subseteq SendTC\_Core\_to\_Device$
    M3_inv_CheckTC_in_Device_Fail : $CheckTC\_in\_Device\_Fail \subseteq SendTC\_Core\_to\_Device$
    M3_inv_TC_GenerateData_in_Device : $TC\_GenerateData\_in\_Device \subseteq TC \times DATA$
    M3_inv5 : $CheckTC\_in\_Device\_Ok \cap CheckTC\_in\_Device\_Fail = \varnothing$
    M3_inv6 : $\forall tc \cdot (tc \in dom(TC\_GenerateData\_in\_Device) \Rightarrow$
            $TC\_Type(tc) \in \{HK\_on\_TC, SCI\_on\_TC\})$
    M3_inv7 : $\forall tc \cdot (tc \in dom(TC\_GenerateData\_in\_Device) \Rightarrow$
            $PID(tc) \in \{mixsc, mixst, sixsp, sixsx\})$
    M3_inv8 : $\forall tc \cdot (tc \in SendTC\_Core\_to\_Device \Rightarrow PID(tc) \in \{mixsc, mixst, sixsp, sixsx\})$
EVENTS
Initialisation
    begin
        act_SendTC_Core_to_Device : $SendTC\_Core\_to\_Device := \varnothing$
        act_CheckTC_in_Device_Ok : $CheckTC\_in\_Device\_Ok := \varnothing$
        act_CheckTC_in_Device_Fail : $CheckTC\_in\_Device\_Fail := \varnothing$
        act_TC_GenerateData_in_Device : $TC\_GenerateData\_in\_Device := \varnothing$
    end
Event   $TCCore\_Execute\_Ok \mathrel{\widehat{=}}$
    any
        $tc$
    where
        typing_tc : $tc \in TC$
        grd_TCCore_Execute_Ok_xor : $tc \notin SendTC\_Core\_to\_Device$
        grd3 : $PID(tc) = csw$
    then
        skip
    end
Event   $SendTC\_Core\_to\_Device \mathrel{\widehat{=}}$
    any
        $tc$
    where
        typing_tc : $tc \in TC$
        grd_SendTC_Core_to_Device : $tc \notin SendTC\_Core\_to\_Device$
        grd1 : $PID(tc) \in \{mixsc, mixst, sixsp, sixsx\}$
    then
        act_SendTC_Core_to_Device : $SendTC\_Core\_to\_Device := SendTC\_Core\_to\_Device \cup \{tc\}$
    end
Event   $CheckTC\_in\_Device\_Ok \mathrel{\widehat{=}}$
    any
        $tc$
    where
        typing_tc : $tc \in TC$
        grd_CheckTC_in_Device_Ok_seq : $tc \in SendTC\_Core\_to\_Device$
        grd_CheckTC_in_Device_Ok : $tc \notin CheckTC\_in\_Device\_Ok$
        grd1 : $tc \notin CheckTC\_in\_Device\_Fail$
    then
        act_CheckTC_in_Device_Ok : $CheckTC\_in\_Device\_Ok := CheckTC\_in\_Device\_Ok \cup \{tc\}$
    end
Event   $SendOkTC\_Device\_to\_Core \mathrel{\widehat{=}}$
    any
        $tc$
    where
        typing_tc : $tc \in TC$
        grd_SendOkTC_Device_to_Core_seq : $tc \in CheckTC\_in\_Device\_Ok$
    then
        skip
    end
Event   $TCCore\_Execute\_Fail \mathrel{\widehat{=}}$
    any
        $tc$
        where
              typing_tc :  $tc \in TC$
              grd_TCCore_Execute_Fail_xor :  $tc \notin SendTC\_Core\_to\_Device$
              grd3 :  $PID(tc) = csw$
        then
              skip
        end
Event    $CheckTC\_in\_Device\_Fail \;\widehat{=}$
        any
              $tc$
        where
              typing_tc :  $tc \in TC$
              grd_CheckTC_in_Device_Fail_seq :  $tc \in SendTC\_Core\_to\_Device$
              grd_CheckTC_in_Device_Fail :  $tc \notin CheckTC\_in\_Device\_Fail$
              grd1 :  $tc \notin CheckTC\_in\_Device\_Ok$
        then
              act_CheckTC_in_Device_Fail :  $CheckTC\_in\_Device\_Fail := CheckTC\_in\_Device\_Fail \cup \{tc\}$
        end
Event    $SendFailTC\_Device\_to\_Core \;\widehat{=}$
        any
              $tc$
        where
              typing_tc :  $tc \in TC$
              grd_SendFailTC_Device_to_Core_seq :  $tc \in CheckTC\_in\_Device\_Fail$
        then
              skip
        end
Event    $TC\_GenerateData\_in\_Device \;\widehat{=}$
        any
              $tc$
              $d$
        where
              typing_d :  $d \in DATA$
              typing_tc :  $tc \in TC$
              grd_TC_GenerateData_in_Device :  $tc \mapsto d \notin TC\_GenerateData\_in\_Device$
              grd1 :  $TC\_Type(tc) \in \{HK\_on\_TC, SCI\_on\_TC\}$
              grd2 :  $PID(tc) \in \{mixsc, mixst, sixsp, sixsx\}$
        then
              act_TC_GenerateData_in_Device :  $TC\_GenerateData\_in\_Device :=$
                    $TC\_GenerateData\_in\_Device \cup \{tc \mapsto d\}$
        end
Event    $TC\_TransferData\_Device\_to\_Core \;\widehat{=}$
        any
              $tc$
              $data$
        where
              typing_tc :  $tc \in TC$
              typing_data :  $data \in \mathbb{P}(DATA)$
              grd_TC_TransferData_Device_to_Core_seq :  $tc \in dom(TC\_GenerateData\_in\_Device)$
              grd1 :  $data = TC\_GenerateData\_in\_Device[\{tc\}]$
        then
              skip
        end
END
```

References

[1] Fathabadi, A. S. & Butler, M. J. Applying Event-B Atomicity Decomposition to a Multi Media Protocol. In *FMCO Formal Methods for Components and Objects*, 89–104 (2010). URL http://eprints.ecs.soton.ac.uk/21261/.

[2] Fathabadi, A. S., Rezazadeh, A. & Butler, M. J. Applying Atomicity and Model Decomposition to a Space Craft System in Event-B. In *NASA Formal Methods*, 328–342 (2011). URL http://eprints.ecs.soton.ac.uk/22048/.

[3] Abrial, J.-R. Formal Methods: Theory Becoming Practice. *J. UCS* **13**, 619–628 (2007).

[4] Abrial, J.-R. *Modeling in Event-B: System and Software Engineering* (Cambridge University Press, 2008).

[5] Jacky, J. *The Way of Z: Practical Programming with Formal Methods* (Cambridge University Press).

[6] Bowen, J. *Formal Specification and Documentation Using Z: A Case Study Approach* (International Thomson Computer Press).

[7] Jones, C. B. *Systematic software development using VDM (2nd ed.)* (Prentice-Hall, Inc., 1990). URL http://portal.acm.org/citation.cfm?id=94062.

[8] de Roever, W. P. & Engelhardt, K. *Data Refinement: Model-oriented Proof Theories and their Comparison*, vol. 46 of *Cambridge Tracts in Theoretical Computer Science* (Cambridge University Press, 1998).

[9] Abrial, J.-R. *The B-book: assigning programs to meanings* (Cambridge University Press, New York, NY, USA, 1996).

[10] Butler, M. J., Leuschel, M. & Snook, C. F. Tools for System Validation with B Abstract Machines. In *Abstract State Machines*, 57–69 (2005).

[11] Cansell, D. & Mery, D. Foundations of the B Method. *Computers and Artificial Intelligence* **22** (2003).

[12] Cansell & Mery. Tutorial on the event-based B method : Concepts and Case Studies. In *Logics of Formal Software Specification Languages - LFSL'2004* (2004). URL http://hal.inria.fr/inria-00100065/en/.

[13] Abrial, J.-R., Butler, M. J., Hallerstede, S. & Voisin, L. An Open Extensible Tool Environment for Event-B. In *ICFEM*, 588–605 (2006).

[14] Hallerstede, S. Justifications for the Event-B Modelling Notation. In *B*, 49–63 (2007).

[15] Abrial, J.-R. *et al.* Rodin: An Open Toolset for Modelling and Reasoning in Event-B (2009). URL http://deploy-eprints.ecs.soton.ac.uk/130/.

[16] Butler, M. J. & Hallerstede, S. The Rodin Formal Modelling Tool. In *BCS-FACS Christmas 2007 Meeting - Formal Methods In Industry* (2007). URL http://deploy-eprints.ecs.soton.ac.uk/4/.

[17] Rodin Project [Online]. http://rodin.cs.ncl.ac.uk/.

[18] Wiki. ProB [Online]. http://wiki.event-b.org/index.php/ProB.

[19] Wiki. UML-B [Online]. http://wiki.event-b.org/index.php/UML-B.

[20] Wiki. B2Latex [Online]. http://wiki.event-b.org/index.php/B2Latex.

[21] Wiki. Decomposition Plug-in User Guide [Online]. http://wiki.event-b.org/index.php/Decomposition_Plug-in_User_Guide.

[22] Badeau, F. & Amelot, A. Using B as a High Level Programming Language in an Industrial Project: Roissy VAL. In *ZB*, 334–354 (2005).

[23] Behm, P., Benoit, P., Faivre, A. & Meynadier, J.-M. Meteor: A Successful Application of B in a Large Project. In *World Congress on Formal Methods*, 369–387 (1999).

[24] Butler, M. J. Decomposition Structures for Event-B. In *Integrated Formal Methods iFM2009* (2009). URL http://deploy-eprints.ecs.soton.ac.uk/51/.

[25] Abrial, J.-R. Refinement, Decomposition and Instantiation of Discrete Models. In *Abstract State Machines*, 17–40 (2005).

[26] Butler, M. J. Incremental Design of Distributed Systems with Event-B. In *Marktoberdorf Summer School 2008 Lecture Notes* (IoS, 2008). URL http://deploy-eprints.ecs.soton.ac.uk/49/.

[27] Silva, R. & Butler, M. J. Supporting Reuse of Event-B Developments through Generic Instantiation. In *International Conference on Formal Engineering Methods (ICFEM 09)* (2009). URL http://eprints.soton.ac.uk/68737/.

[28] Holloway, C. M. Why Engineers Should Consider Formal Methods. Technical Report (1997).

[29] Butler, R. E. What is Formal Methods. Technical Report (2001). URL http://shemesh.larc.nasa.gov/fm/fm-what.html.

[30] Rangarajan, M. Formal Methods: Analysis of complex systems to ensure correctness and reduce cost [Online]. http://www51.honeywell.com/aero/technology/common/documents/formal-methods.pdf.

[31] Back, R.-J. & von Wright, J. *Refinement Calculus: A Systematic Introduction* (Springer-Verlag, 1998). Graduate Texts in Computer Science.

[32] Morgan, C. *Programming from Specifications* (1990).

[33] Abrial, J.-R., Schuman, S. A. & Meyer, B. Specification Language. In *On the Construction of Programs*, 343–410 (1980).

[34] Liu, X., Yang, H. & Zedan, H. Formal Methods for the Re-Engineering of Computing Systems: A Comparison. In *COMPSAC*, 409– (1997).

[35] Steria. Atelier B, User and Reference Manuals [Online]. http://www.atelierb.societe.com/indexuk.html (1996).

[36] Abdallah, A. E., Jones, C. B. & Sanders, J. W. (eds.). *Communicating Sequential Processes: The First 25 Years, Symposium on the Occasion of 25 Years of CSP, London, UK, July 7-8, 2004, Revised Invited Papers*, vol. 3525 of *Lecture Notes in Computer Science* (Springer, 2005).

[37] Hoare, C. A. R. *Communicating Sequential Processes* (Prentice-Hall, 1985).

[38] Butler, M. J. A CSP Approach to Action Systems (1992). PhD thesis.

[39] Back, R.-J. & Kurki-Suonio, R. Distributed Cooperation with Action Systems. *ACM Trans. Program. Lang. Syst.* **10**, 513–554 (1988).

[40] Back, R.-J. Refinement Calculus, Part II: Parallel and Reactive Programs. In *REX Workshop*, 67–93 (1989).

[41] Liu, X., Chen, Z., Yang, H., Zedan, H. & Chu, W. C. A Design Framework for System Re-Engineering. *Asia-Pacific Software Engineering Conference* **0**, 342 (1997).

[42] Nami, M. R. & Hassani, F. A comparative evaluation of the Z, CSP, RSL, and VDM languages. *ACM SIGSOFT Software Engineering Notes* **34**, 1–4 (2009).

[43] Bolognesi, T. A Conceptual Framework for State-Based and Event-Based Formal Behavioural Specification Languages. In *ICECCS*, 107–116 (2004).

[44] Butler, M. J. & Yadav, D. An incremental development of the Mondex system in Event-B. *Formal Asp. Comput.* **20**, 61–77 (2008).

[45] Abrial, J.-R., Cansell, D. & Mery, D. Refinement and Reachability in Event-B. In *ZB*, 222–241 (2005).

[46] Rezazadeh, A., Butler, M. J. & Evans, N. Redevelopment of an Industrial Case Study Using Event-B and Rodin (2007). URL http://deploy-eprints.ecs.soton.ac.uk/9/.

[47] Hallerstede, S. On the Purpose of Event-B Proof Obligations. In *ABZ*, 125–138 (2008).

[48] Abrial, J.-R. Summary of Event-B Proof Obligations. http://www.docstoc.com/docs/7055755/Summary-of-Event-BProof-Obligations (2008).

[49] Event-B [Online]. http://www.event-b.org/.

[50] Wiki. Event Model Decomposition [Online]. http://wiki.event-b.org/index.php/Event_Model_Decomposition.

[51] Abrial, J.-R. Extending b without changing it (for developing distributed systems). *In H. Habrias, editor, 1st Conference on the B method* 169–190 (1996).

[52] Abrial, J.-R. *Event model decomposition* (2009). URL http://deploy-eprints.ecs.soton.ac.uk/109/1/626.pdf. Technical Report, ETH Zurich, Private communication.

[53] Silva, R. Supporting Development of Event-B Models (2009). A mini-thesis submitted for transfer from MPhil to PhD.

[54] Pascal, C. & Silva, R. Event-B Model Decomposition. In *DEPLOY Plenary Technical Workshop* (2009). URL http://eprints.soton.ac.uk/69664/.

[55] Butler, M. J. An Approach to the Design of Distributed Systems with B AMN. In *ZUM*, 223–241 (1997).

[56] Silva, R., Pascal, C., Hoang, T. & Butler, M. J. Decomposition Tool for Event-B. In *ABZ* (2010). URL http://eprints.ecs.soton.ac.uk/18427/.

[57] C., M. & Abrial, J.-R. *Event-B Language* (2005). URL http://rodin.cs.ncl.ac.uk/deliverables/D7.pdf. Deliverable 3.2, EU Project IST-511599 - RODIN.

[58] Abrial, J.-R. & Hallerstede, S. Refinement, Decomposition, and Instantiation of Discrete Models: Application to Event-B. *Fundam. Inform.* **77**, 1–28 (2007).

[59] Butler, M. J. *Synchronisation-based Decomposition for Event-B* (2006). In: RODIN Deliverable D19 Intermediate report on methodology.

[60] Ball, E. *An Incremental Process for the Development of Multi-agent Systems in Event-B*. Ph.D. thesis, University of Southampton, UK (2008).

[61] Said, M. Y., Butler, M. J. & Snook, C. F. Language and Tool Support for Class and State Machine Refinement in UML-B. In *FM*, 579–595 (2009).

[62] Zeyda, F. & Cavalcanti, A. Mechanised Translation of Control Law Diagrams into Circus. In *IFM*, 151–166 (2009).

[63] Woodcock, J. & Cavalcanti, A. The Semantics of Circus. In *ZB*, 184–203 (2002).

[64] Woodcock, J. & Davies, J. *Using Z—Specification, Refinement, and Proof*. Series in Computer Science (Prentice Hall International, 1996).

[65] Schneider, S. & Treharne, H. Verifying Controlled Components. In *IFM*, 87–107 (2004).

[66] Butler, M. J. csp2B: A Practical Approach to Combining CSP and B. *Formal Asp. Comput.* **12**, 182–198 (2000).

[67] Schneider, S., Treharne, H. & Wehrheim, H. A CSP Approach to Control in Event-B. In *IFM*, 260–274 (2010).

[68] Snook, C. F. & Butler, M. J. UML-B and Event-B: an integration of languages and tools. In *The IASTED International Conference on Software Engineering* (2008).

[69] Iliasov, A. Tutorial on the Flow plugin for Event-B. In *Workshop on B Dissemination [WOBD] Satellite event of SBMF, Natal, Brazil* (2010).

[70] Iliasov, A. *On Event-B and Control Flow* (2009). URL http://deploy-eprints.ecs.soton.ac.uk/144/1/flows-paper..pdf. School of Computing Science, Newcastle University.

[71] Crocker, D. & Overell, P. Augmented BNF for Syntax Specifications: ABNF. plain text (2008). STD 68, RFC 5234.

[72] Eclipse [Online]. http://www.eclipse.org.

[73] Steinberg, D., Budinsky, F., Paternostro, M. & Merks, E. *EMF: Eclipse Modeling Framework* (Addison-Wesley Professional), second edn.

[74] Kolovos, D., Rose, L. & Paige, R. *THE epsilon BOOK* (2008).

[75] Wiki. Eclipse [Online]. http://wiki.eclipse.org.

[76] Zave, P. & Cheung, E. Compositional Control of IP Media. *IEEE Trans. Software Eng.* **35**, 46–66 (2009).

[77] ESA Media Center, Space Science. Factsheet: Bepicolombo. http://www.esa.int/esaSC/SEMNEM3MDAF_0_spk.html.

[78] Deploy Project [Online]. http://www.deploy-project.eu/.

[79] Ltd, S. S. F. *BepiColombo - Modelling Approach* (2008). Technical Report, DEPLOY Project.

[80] Ltd, S. S. F. *Industrial deployment of advanced system engineering methods for high productivity and dependability* (2010). Report on Pilot Deployment in the Space Sector, DEPLOY Project.

[81] Tindell, K., Burns, A. & Wellings, A. Calculating Controller Area Network (CAN) Message Response Times. *Control Engineering Practice* **3**, 1163–1169 (1995).

[82] Yeganehfard, S. From Requirement Document to Formal Modelling and Decomposition of Control Systems. Chapter 7, mini-thesis Report, University of Southampton (2012).

[83] Graphical Modelling Framework (GMF) [Online]. http://www.eclipse.org/modeling/gmp/.

[84] EuGENia [Online]. http://www.eclipse.org/gmt/epsilon/doc/eugenia/.