UNIVERSITY OF SOUTHAMPTON

FACULTY OF PHYSICAL SCIENCES AND ENGINEERING
Electronics and Computer Science

Extending the ERS Approach for Workflow Modelling in Event-B

by

Dana Dghaym

Thesis for the degree of Doctor of Philosophy

April 2017
The Event Refinement Structures (ERS) approach augments the Event-B formal method with hierarchical diagrams, providing explicit support for control flow and refinement relationships. ERS was originally designed to decompose the atomicity of the events in Event-B and later enriched with control flow combinators.

Combining graphical workflow approaches with formal methods has been a subject of interest in both industry and academia, resulting in a diversity of approaches. In this thesis, we present an approach for workflow modelling that addresses both control flow and data handling. ERS is used for control flow, while Event-B mathematical notation supports the data handling. This separation simplifies the modelling by avoiding an extensive number of patterns, though separation does not mean the independence of control flow from data handling. The dependency is achieved by the ERS semantics, which are acquired by transforming the diagrams to Event-B. This combination not only benefits from the verification capabilities of Event-B and the graphical nature of ERS, but also supports incremental modelling through refinement and hierarchy.

Our studies resulted in extending the ERS approach to support more flexible behaviour like unbounded replication and exception handling. Unbounded replication is needed when the number of instances of a flow to be executed is unknown and additional instances can be initiated during execution. We also enhance some of the existing ERS combinators such as the loop. We validate our approach and extensions by applying them to two complex workflows, the fire dispatch system and the travel agency booking system. Finally, we extend the ERS formal language with new translation rules to support our new ERS extensions. We formally define the new translation rules of ERS to Event-B, using the Augmented Backus-Naur Form (ABNF), to be easily integrated in the ERS plug-in. The ERS plug-in is a tool providing automatic generation of part of the Event-B model representing types and sequencing. We also evaluate the ERS combinators in control flow modelling against already published criteria.
Contents

Declaration of Authorship xix
Acknowledgements xxi

1 Introduction 1
  1.1 Aims and Contributions 2
  1.2 Research and Modelling Methodology 4
  1.3 Thesis Organisation 5

2 Workflow Modelling Approaches and Specification Techniques 7
  2.1 Introduction 7
  2.2 Jackson Structure Diagrams (JSD) 7
  2.3 Unified Modelling Language (UML) 8
     2.3.1 Activity Diagrams 9
     2.3.2 State Machines 10
  2.4 Business Process Model and Notation (BPMN) 11
  2.5 Petri Nets 13
  2.6 Yet Another Workflow Language (YAWL) 13
  2.7 Formal Modelling of Workflows 15
     2.7.1 Formalising Control Flow Patterns Using Abstract State Machine (ASM) 15
     2.7.2 Other Work on Formalising Workflows 16
  2.8 Conclusion 17

3 Formal Modelling Approaches 19
  3.1 Introduction 19
  3.2 Formal Methods 19
     3.2.1 Overview of Some Formal Methods 20
  3.3 Event-B 22
     3.3.1 Structure 22
     3.3.2 Refinement 23
     3.3.3 Proof Obligation Rules 25
     3.3.4 Rodin: An Event-B Modelling Tool 25
     3.3.5 UML-B 26
  3.4 Event Refinement Structures Approach (ERS) 27
     3.4.1 Event Refinement Structures (ERS) Combinators 29
        3.4.1.1 And Combinator 31
        3.4.1.2 Or Combinator 31
## Contents

3.4.1.3 Xor Combinator ........................................... 32
3.4.1.4 Loop Combinator ........................................ 34
3.4.1.5 All Combinator ........................................... 36
3.4.1.6 Some Combinator ........................................ 38
3.4.1.7 One Combinator .......................................... 38
3.4.2 Event Refinement Structures (ERS) Constraints ................. 40
3.5 Conclusion ...................................................... 42

### 4 Extending ERS by Dynamic Workflow Modelling

4.1 Introduction .................................................... 43
4.2 Limitations of the Original ERS Approach in Control Flow Modelling .................................. 43
4.2.1 Data Dependent Workflow Limitations ................. 43
4.2.2 Loop Limitations ........................................... 44
4.3 Advantages of ERS in Control Flow Modelling ......................... 45
4.4 Parameter Extension ........................................... 48
4.5 A Dynamic Replicator: Par-Replicator .......................... 49
4.5.1 Par-Replicator Semantics .................................. 51
4.6 Non-Deterministic Loop ........................................ 55
4.6.1 Non-Deterministic Loop Semantics ....................... 56
4.7 Conclusion ...................................................... 58

### 5 Fire Dispatch Case Study

5.1 Introduction .................................................... 61
5.2 Fire Dispatch System Overview ................................ 61
5.3 Fire Dispatch System Requirements ........................... 62
5.4 Incremental Development of the Fire Dispatch Model .......... 64
5.4.1 Abstract and First Refinement: Duplicate and Non-Duplicate .......... 65
5.4.2 Second and Third Refinement: Incident Management .......... 66
5.4.3 Fourth Refinement: Incident Details ....................... 68
5.4.4 Fifth, Sixth and Seventh Refinement: Physical Resources ....... 71
5.5 Evaluation of the Fire Dispatch Case Study .................... 74
5.6 Conclusion ...................................................... 77

### 6 Extending ERS by Exception Handling

6.1 Introduction .................................................... 79
6.2 Exception Handling Combinators ................................ 80
6.2.1 The Interrupt Combinator Semantics ....................... 81
6.2.2 The Retry Combinator Semantics ............................ 84
6.3 Application to the Fire Dispatch Workflow ...................... 88
6.3.1 Distinction Between Duplicate and Non-Duplicate Incidents Using the Interrupt Combinator ............... 90
6.3.2 Introducing the Priority of the Incident .................... 92
6.3.3 Handling the Lost Allocated Resources with the Retry Combinator .... 94
6.3.4 Handling Allocated Resources to Duplicate Incidents .......... 95
6.3.5 Extending the Model with State Machines .................. 98
6.3.6 Evaluation of the Updated Fire Dispatch Case Study ........ 101
6.4 Comparison with Other Work on Exception Handling .......... 102
CONTENTS

8.4.4.5 TR_retry_reset: Mapping the Retry Combinator to a Reset Event .......................... 155
8.5 ERS Tool Support ........................................................................................................... 156
8.5.1 User Interface ............................................................................................................. 160
8.6 Conclusion ....................................................................................................................... 162

9 Conclusions ....................................................................................................................... 165
9.1 Contributions ................................................................................................................... 165
9.2 Limitations ....................................................................................................................... 166
9.3 Lessons Learned ............................................................................................................. 167
9.3.1 Separation of Control Flow and Data Handling .......................................................... 167
9.3.2 Integration of ERS with Other Event-B Approaches ................................................. 167
9.3.3 Proper Behaviour of the Model ................................................................................. 168
9.3.4 Modelling New Behaviours and Features ................................................................. 169
9.3.5 Role of Invariants ....................................................................................................... 170
9.3.6 Role of ERS Diagrams in Refinements and Proofs .................................................... 170
9.4 Comparing ERS with Other Workflow Modelling Approaches ..................................... 171
9.4.1 Workflow Control Flow Patterns .............................................................................. 171
9.4.2 UML Activity Diagram ............................................................................................. 175
9.4.3 iUML-B Statemachines ............................................................................................. 177
9.5 Other Work on Integration of Formal Methods and Graphical Representations ............. 178
9.5.1 Tree-Like Structures Integrated with Formal Methods ............................................. 178
9.5.2 BPMN and Formal Methods ..................................................................................... 179
9.5.3 UML Activity Diagrams and Formal Methods ......................................................... 179
9.5.4 Other UML Diagrams and Formal Methods .............................................................. 181
9.6 Overview of ERS in Comparison to Other Approaches ................................................ 182
9.7 Conclusion and Future Work .......................................................................................... 183

Application to Case Studies: ............................................................................................... 184
Timing Patterns: .................................................................................................................... 184
Decomposition: ..................................................................................................................... 184
Transformation Correctness: ............................................................................................... 184

A Event-B Model of the Fire Dispatch Workflow .................................................................. 187
A.1 Abstract Level .................................................................................................................. 187
A.1.1 Context C0 ................................................................................................................ 187
A.1.2 ERS Diagram: M0 ..................................................................................................... 187
A.1.3 Machine M0 .............................................................................................................. 188
A.2 First Refinement .............................................................................................................. 189
A.2.1 ERS Diagrams: M1 .................................................................................................. 189
A.2.2 Machine M1 ............................................................................................................. 189
A.3 Second Refinement ......................................................................................................... 192
A.3.1 ERS Diagram: M2 .................................................................................................... 192
A.3.2 Machine M2 ............................................................................................................ 192
A.4 Third Refinement ............................................................................................................ 196
A.4.1 Context C1 .............................................................................................................. 196
A.4.2 ERS Diagram: M3 .................................................................................................... 196
| A.4.3 | Machine M3 | 198 |
| A.5 | Fourth Refinement | 202 |
| A.5.1 | Context C2 | 202 |
| A.5.2 | ERS Diagram: M4 | 203 |
| A.5.3 | Machine M4 | 205 |
| A.6 | Fifth Refinement | 211 |
| A.6.1 | ERS Diagram: M5 | 211 |
| A.6.2 | Machine M5 | 213 |
| A.7 | Sixth Refinement | 220 |
| A.7.1 | Context C3 | 220 |
| A.7.2 | Machine M6 | 221 |
| A.8 | Seventh Refinement | 229 |
| A.8.1 | Context C4 | 229 |
| A.8.2 | Machine M7 | 229 |

**B Fire Dispatch Model with Exception Handling**

| B.1 | Static Part of the Model: Contexts | 239 |
| B.1.1 | Context: C0 | 239 |
| B.1.2 | Context: C1 | 239 |
| B.1.3 | Context: C2 | 239 |
| B.1.4 | Context: C3 | 240 |
| B.1.5 | Context: C4 | 240 |
| B.1.6 | Context: C5 | 241 |
| B.1.7 | Context: M9_implicitContext | 241 |
| B.2 | Model Refinements | 242 |
| B.2.1 | Abstract Level: M0 | 242 |
| B.2.2 | First Refinement: M1 | 244 |
| B.2.3 | Second Refinement: M2 | 246 |
| B.2.4 | Third Refinement: M3 | 250 |
| B.2.5 | Fourth Refinement: M4 | 255 |
| B.2.6 | Fifth Refinement: M5 | 261 |
| B.2.7 | Sixth Refinement: M6 | 267 |
| B.2.8 | Seventh Refinement: M7 | 275 |
| B.2.9 | Eighth Refinement: M8 | 285 |
| B.2.10 | Ninth Refinement: M9 | 297 |
| B.2.11 | Tenth Refinement: M10 | 311 |

**C Event-B Model of the Travel Agency Workflow**

| C.1 | Static Part of the Model: Contexts | 327 |
| C.1.1 | Context: C0 | 327 |
| C.1.2 | Context: C1 | 327 |
| C.2 | Model Refinements | 328 |
| C.2.1 | Abstract Level: M0 | 328 |
| C.2.2 | First Refinement: M1 | 329 |
| C.2.3 | Second Refinement: M2 | 331 |
| C.2.4 | Third Refinement: M3 | 335 |
| C.2.5 | Fourth Refinement: M4 | 342 |
D Functions Used in Formalising the ERS Translation Rules 351
  D.1 Predecessor and Successor Functions 351
  D.2 Traversing Functions 352
  D.3 Sequencing Functions 352

References 359
## List of Figures

2.1 Example of Jackson Structure Diagram (Jackson, 1983) .................. 8  
2.2 Example of Premature Termination Using Jackson Structure Diagram  (Jackson, 1983) ......................................................... 8  
2.3 Some Constructs of UML Activity Diagrams ................................. 10  
2.4 "Order" Activity: Simple Example of Activity Diagram .................. 10  
2.5 Composite State With Regions (omg.org, 2012) .......................... 11  
2.6 BPMN Diagram of an Order Process (Object Management Group, 2010) 13  
2.7 YAWL Symbols (van der Aalst and ter Hofstede, 2005) ................. 14  
2.8 Examples of Multiple Instances Tasks Using YAWL (van der Aalst and ter Hofstede, 2005) ....................................................... 15  
3.1 Example of Z Schema Notation (Bowen, 2003) ............................ 20  
3.3 Simple Event-B Model .......................................................... 23  
3.4 Machine and Context Relationships ........................................... 24  
3.5 A Basic Structure for an ERS Diagram Using Single Instance (SI) Modelling and its Event-B Semantics ....................................... 28  
3.6 A Basic Structure for an ERS Diagram Using Multiple Instances (MI) Modelling and its Event-B Semantics .................................. 28  
3.7 ERS Representation of Sequential and Non-Sequential Behaviour .......... 30  
3.8 ERS Example Diagram .......................................................... 30  
3.9 ERS \texttt{and-combinator} (Single Instance Case) ........................... 31  
3.10 ERS \texttt{and-combinator} (Multiple Instances Case) ....................... 32  
3.11 ERS \texttt{or-combinator} (Single Instance Case) ............................ 33  
3.12 ERS \texttt{or-combinator} (Multiple Instances Case) ........................ 33  
3.13 ERS \texttt{xor-combinator} (Single Instance Case) .......................... 34  
3.14 ERS \texttt{xor-combinator} (Multiple Instances Case) ....................... 35  
3.15 ERS \texttt{loop-combinator} (Single Instance Case) .......................... 35  
3.16 ERS \texttt{loop-combinator} (Multiple Instances Case) ....................... 36  
3.17 ERS \texttt{all-combinator} (Single Instance Case) ............................ 37  
3.18 ERS \texttt{all-combinator} (Multiple Instances Case) ........................ 37  
3.19 ERS \texttt{some-combinator} (Single Instance Case) .......................... 38  
3.20 ERS \texttt{some-combinator} (Multiple Instances Case) ....................... 39  
3.21 ERS \texttt{one-combinator} (Single Instance Case) ........................... 39  
3.22 ERS \texttt{one-combinator} (Multiple Instances Case) ....................... 40  
3.23 General Pattern for Introducing Replicator ................................ 41  
4.1 ERS Diagram of a Fire Dispatch Workflow .................................. 44
4.2 Invalid Representation of Non-Deterministic Loop Events .......................... 45
4.3 Correct Representation of Non-Deterministic Loop Events in the Original ERS ................................................................................................. 46
4.4 Generalisation of the Loop ........................................................................ 46
4.5 Generalised ERS Diagram ......................................................................... 47
4.6 Home Care System Example ...................................................................... 49
4.7 Diagrammatic Representation of the par-replicator .................................. 50
4.8 Parallel Producer-Consumer Pattern ......................................................... 51
4.9 Single Instance Semantics of the par-replicator ....................................... 52
4.10 Multiple Instances Semantics of the par-replicator .................................. 53
4.11 Refinement of the par-replicator Event ..................................................... 54
4.12 The par-replicator Refining Event ............................................................ 55
4.13 Diagrammatic Representation of the Generalised Loop ......................... 56
4.14 Single Instance Semantics of the Loop ..................................................... 56
4.15 Multiple Instances Semantics of the Loop .............................................. 57
4.16 Refinement of a Loop Child ...................................................................... 58
5.1 The Control Operator Workstation ............................................................. 62
5.2 ERS Diagram for the Fire Dispatch Abstract Level .................................. 65
5.3 Refinement of Non-Duplicate and Duplicate Events .................................. 65
5.4 Application of the Parallel Producer-Consumer Pattern for Incident Man-
agement ........................................................................................................ 66
5.5 Setting the Action Plan (ap) in SetInitialAP Event Using Event-B .......... 67
5.6 Event-B Specifications of the Producer Events ........................................ 68
5.7 Event-B Specifications of the Consumer Events ....................................... 69
5.8 Refinement of GatherInfo Introducing Type and Location ....................... 70
5.9 Refinement of the Initial Action Plan According to Type and Location ...... 70
5.10 Different Steps of Incident Handling ......................................................... 71
5.11 Allocation and Deallocation of a Physical Resource pr in Event-B .......... 72
5.12 Reallocation of a Quicker Resource to the Incident ................................. 73
5.13 Effect of ERS Invariant in Discharging Proof Obligations ....................... 77
5.14 Proof Details of the DeallocateRes/GRD ............................................... 78
5.15 Fire Dispatch Model Statistics ................................................................. 78
6.1 Generalised Form of the interrupt and the retry Combinators ................... 80
6.2 Single Instance Semantics of the Interrupt Combinator ......................... 82
6.3 Multiple Instances Semantics of the Interrupt Combinator ..................... 83
6.4 Invalid Refinement of the Interrupt Combinator ..................................... 84
6.5 Refinement Semantics of the normal sub-flow B of Figure 6.2 ................. 85
6.6 Single Instance Semantics of the Retry Combinator ................................. 86
6.7 Multiple Instances Semantics of the Retry Combinator ........................... 87
6.8 Behaviour of retry (Figure 6.6) at Abstract Level ..................................... 87
6.9 Introducing Exception Handling at Two Levels ........................................ 88
6.10 Refinement Semantics of the normal sub-flow B of Figure 6.7 ................ 89
6.11 Distinction Between Duplicate and Non-Duplicate Incidents ................ 91
6.12 Invariants of the First Refinement Level (M1) ........................................ 91
6.13 Event-B Specification of some Events of the Refinement Level (M1) ...... 92
6.14 Introducing Priority of the Incident Using Weak Sequencing .......................... 93
6.15 Incident Management Using the Retry Combinator ................................. 94
6.16 Event-B Specification of the Allocation of Resources and its Cancellation at the Refinement Level (M8) ................................................................. 96
6.17 Event-B Specification of RemoveBrokenRes and the Resetting Event at the Refinement Level (M8) ................................................................. 97
6.18 Refining the IsDuplicate Interrupting Event to Handle Allocated Resources 97
6.19 State Machine of the Status Change of a Physical Resource ...................... 98
6.20 Properties Overview of the Physical Resource State Machine .................. 99
6.21 Event-B Specification of Some of the Events Updating the Resource Status 100
6.22 ReallocateQuickerRes Extended Manually with the Resource Status Update 101
6.23 Activity Diagrams Exception Handling Mechanisms (Object Management Group (OMG), 2015) ................................................................. 102
6.24 Interruptible Activity Region (Object Management Group (OMG), 2015) 103
6.25 Exception Handling Example Using BPMN (Object Management Group, 2011) ................................................................. 104
6.26 Possible ERS Representation of Figure 6.25 ............................................... 105
6.27 Little-JIL Step (Wise et al., 2000) ................................................................. 106

7.1 ERS Diagram for the Abstract and First Refinement Levels of the Travel Agency System ................................................................................... 115
7.2 ERS Diagram for the Second and Third Refinement Levels of the Travel Agency System ................................................................................... 115
7.3 Event-B Specification of the Events Updating the Flight Itinerary .............. 116
7.4 CompFlightItin Event ...................................................................................... 117
7.5 Refinement of Quit to Remove Booked Flights, Hotels and Cars from Itinerary ................................................................................... 117
7.6 Another Possible Handling of Booked Itinerary ........................................... 118
7.7 Setting the Time of an Itinerary Item ............................................................ 119
7.8 Extending the Flight Events with Time ........................................................ 119
7.9 Statistics of the Travel Agency Model .......................................................... 119
7.10 Flight Itinerary Using the Retry Combinator .............................................. 120

8.1 An Example of an ERS Diagram ................................................................. 125
8.2 list_child_leaves Function ........................................................................... 130
8.3 list_first_child_leaves Function ................................................................... 131
8.4 build_par_ref grd Function .......................................................................... 131
8.5 par_ref grd Function .................................................................................... 132
8.6 build_succ_SI inv Function .......................................................................... 132
8.7 Translation Rules Tabular Format .................................................................. 134
8.8 TR_rangeExpression: Mapping a replicator to (a) guard(s) ....................... 135
8.9 Application of TR_rangeExpression to the par-rep and all ......................... 136
8.10 TR_rep: Mapping a replicator leaf to an invariant ....................................... 137
8.11 Application of TR_rep ................................................................................. 138
8.12 TR_par1: Mapping a par-rep to (a) guard(s) .............................................. 139
8.13 Application of TR_par1 ................................................................................. 140
8.14 TR_par2: Mapping a refined par-rep to (a) guard(s) in the successor event(s) ................................................................. 141
8.15 Application of TR_par2 ................................................................................. 142
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.16</td>
<td>TR_par3: Mapping a refined par-rep to an invariant</td>
<td>142</td>
</tr>
<tr>
<td>8.17</td>
<td>Application of TR_par3</td>
<td>143</td>
</tr>
<tr>
<td>8.18</td>
<td>TR_loop1: Mapping a loop to (a) guard(s)</td>
<td>144</td>
</tr>
<tr>
<td>8.19</td>
<td>Application of TR_loop1</td>
<td>145</td>
</tr>
<tr>
<td>8.20</td>
<td>TR_loop2: Mapping a loop to (a) guard(s) in the successor child</td>
<td>146</td>
</tr>
<tr>
<td>8.21</td>
<td>Application of TR_loop2</td>
<td>147</td>
</tr>
<tr>
<td>8.22</td>
<td>TR_loop3: Mapping a loop to (a) resetting event(s)</td>
<td>148</td>
</tr>
<tr>
<td>8.23</td>
<td>Application of TR_loop3</td>
<td>149</td>
</tr>
<tr>
<td>8.24</td>
<td>TR_loop4: Mapping a loop to (an) invariant(s)</td>
<td>150</td>
</tr>
<tr>
<td>8.25</td>
<td>Application of TR_loop4</td>
<td>151</td>
</tr>
<tr>
<td>8.26</td>
<td>TR_interrupt: Mapping the interrupt/retry to Guard(s) in the Normal Flow</td>
<td>152</td>
</tr>
<tr>
<td>8.27</td>
<td>Application of TR_interrupt to the interrupt-combinator</td>
<td>153</td>
</tr>
<tr>
<td>8.28</td>
<td>TR_disable_interrupt: Mapping the interrupt/retry to Guard(s) in the</td>
<td>154</td>
</tr>
<tr>
<td></td>
<td>Interrupting Flow</td>
<td></td>
</tr>
<tr>
<td>8.29</td>
<td>Application of TR_disable_interrupt</td>
<td>155</td>
</tr>
<tr>
<td>8.30</td>
<td>TR_interrupt_inv: Mapping the interrupt/retry to an Invariant</td>
<td>156</td>
</tr>
<tr>
<td>8.31</td>
<td>Application of TR_interrupt_inv</td>
<td>157</td>
</tr>
<tr>
<td>8.32</td>
<td>TR_retry: Mapping the retry Combinator to Resetting Actions of the</td>
<td>158</td>
</tr>
<tr>
<td></td>
<td>Normal Flow</td>
<td></td>
</tr>
<tr>
<td>8.33</td>
<td>Application of TR_retry</td>
<td>159</td>
</tr>
<tr>
<td>8.34</td>
<td>TR_retry_reset: Mapping the retry Combinator to a Resetting Event to</td>
<td>160</td>
</tr>
<tr>
<td></td>
<td>Reset the Interrupting Flow</td>
<td></td>
</tr>
<tr>
<td>8.35</td>
<td>Application of TR_retry_reset</td>
<td>161</td>
</tr>
<tr>
<td>8.36</td>
<td>TR_par1 Rule: enabled and dependenciesOK Methods</td>
<td>162</td>
</tr>
<tr>
<td>8.37</td>
<td>TR_par1 Rule: fire Method</td>
<td>163</td>
</tr>
<tr>
<td>8.38</td>
<td>An Example ERS Diagram at Abstract Level in the ERS Tool</td>
<td>163</td>
</tr>
<tr>
<td>8.39</td>
<td>Refinement in the ERS Tool</td>
<td>164</td>
</tr>
<tr>
<td>9.1</td>
<td>Machine M Generated by Applying Different Approaches</td>
<td>168</td>
</tr>
<tr>
<td>9.2</td>
<td>Integration of ERS with Other Approaches</td>
<td>169</td>
</tr>
<tr>
<td>9.3</td>
<td>Example of Inconsistent Integration of Diagrams</td>
<td>169</td>
</tr>
<tr>
<td>9.4</td>
<td>ERS Refinement Example from Fire Dispatch Workflow</td>
<td>171</td>
</tr>
<tr>
<td>9.5</td>
<td>Graphical Comparison of the non-replicator Combinators of ERS and Activity</td>
<td>176</td>
</tr>
<tr>
<td></td>
<td>Diagrams</td>
<td></td>
</tr>
<tr>
<td>9.6</td>
<td>Structured Loop Node in UML 2.0 and Its Equivalent Representation</td>
<td>176</td>
</tr>
<tr>
<td>9.7</td>
<td>Expansion Region (omg.org, 2012)</td>
<td>176</td>
</tr>
<tr>
<td>9.8</td>
<td>Safe Transportation (van Lamsweerde, 2008): Example of KAOS Goal Model</td>
<td>178</td>
</tr>
<tr>
<td>A.1</td>
<td>ERS Diagram at Abstract Level: M0</td>
<td>187</td>
</tr>
<tr>
<td>A.2</td>
<td>ERS Diagram at First Refinement: M1</td>
<td>189</td>
</tr>
<tr>
<td>A.3</td>
<td></td>
<td>193</td>
</tr>
<tr>
<td>A.4</td>
<td>ERS Diagram at Third Refinement: M3</td>
<td>197</td>
</tr>
<tr>
<td>A.5</td>
<td>ERS Diagram at Fourth Refinement: M4</td>
<td>204</td>
</tr>
<tr>
<td>A.6</td>
<td>ERS Diagram at Fifth Refinement Level: M5</td>
<td>212</td>
</tr>
<tr>
<td>D.1</td>
<td>Updated predecessor Function</td>
<td>351</td>
</tr>
</tbody>
</table>
D.2 Updated successor Function ........................................ 352
D.3 Updated conjunction_of_leaves Function .......................... 353
D.4 Updated disjunction_of_leaves Function ........................... 354
D.5 Updated union_of_leaves Function .................................. 355
D.6 Updated build_seq_grd Function .................................... 356
D.7 Sequencing between Two Leaf Events (Salehi Fathabadi, 2012) .... 357
D.8 Original seq_grd Function (Salehi Fathabadi, 2012) ........ 358
# List of Tables


4.1 Disabling and Enabling Conditions for ERS Patterns .......................... 47
4.2 Disabling and Enabling Conditions for the `par-replicator` .................. 51

5.1 Incident Information Requirements ................................................. 63
5.2 Incident Handling Requirements ..................................................... 63
5.3 Resource Manipulation Requirements .............................................. 63
5.4 Incident Closure Requirements ...................................................... 64
5.5 Requirements Tracing of the Fire Dispatch Case Study ........................ 74

6.1 Disabling and Enabling Conditions for `interrupt` and `retry` ................. 81

8.1 Original ERS Syntax ........................................................................ 124
8.2 Extended ERS Syntax ..................................................................... 124
8.3 ABNF Representation of the ERS Example in Figure 8.1 ..................... 126
8.4 ERS Leaf-Related Translation Rules ................................................ 133
8.5 ERS Combinator-Related Translation Rules .................................... 133
8.6 New ERS Translation Rules ............................................................. 162

9.1 Evaluation of ERS Patterns Against the Generic Control Flow Patterns . 173
Declaration of Authorship

I, Dana Dghaym, declare that the thesis entitled Extending the ERS Approach for Workflow Modelling in Event-B 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: (Dghaym et al., 2013), (Dghaym et al., 2016)

Signed:...........................................................................................................................

Date:...................................................................................................................................
Acknowledgements

I would like to express my sincere gratitude to my supervisor Professor Michael Butler, for all his guidance and support during my PhD. I cannot thank him enough for all what I have learned and will continue to learn from him, and for his great understanding of my family commitments. I would also like to thank my second supervisor Dr Asieh Salehi Fathabadi, who was not only a supervisor but also a great friend.

I would like to thank Dr Corina Cirstea for all her valuable feedback and comments. I would also like to thank Matheus Garay Trindade and Dr Colin Snook for all their help in development with the iUML-B framework.

I would also like to express my gratitude to my former colleagues at the Hampshire Fire and Rescue Service for their help with my case study requirements.

Special thanks to all my friends in the ESS group for all their encouragement, support and great company during my PhD.

I cannot thank enough my parents, sisters and brothers, who despite all the distance were always beside me and gave me the strength and encouragement to pursue all my dreams.

Lastly, I would like to thank my beautiful little family. My husband and soulmate for all his love, support and encouragement. My angel, Celine who is not only just my little daughter, but also my best friend and the most amazing little helper. My forever baby, Adam for all the laughter and smiles that enlightened my days.
Chapter 1

Introduction

Formal methods are mathematical techniques used for the systematic specification, verification and development of both hardware and software systems. However, these techniques have been mainly criticised for their difficulty (Wing, 1990; Hall, 1990; Bowen and Hinchey, 1995; Abrial, 2007). On the other hand, semi-formal notations are graphical representations of the system, that lack a precise or formal semantics. Despite all the criticisms about lacking precise semantics, semi-formal notations, such as the Unified Modelling Language (UML) (Rumbaugh et al., 1998), are very popular and widely used in industry. Probably, the main reason for the popularity of semi-formal representation is the use of both textual and graphical notions, where many studies have shown that such integration of graphical and textual representations can improve the comprehensibility of problems (Mayer et al., 1996; Razali et al., 2007).

The thesis builds on the Event-B formalism (Abrial, 2010). Event-B is a formal method based on set theory and first order logic. Refinement and decomposition are important concepts in Event-B, for solving the complexity of formal modelling (Abrial and Haller-stede, 2007). Refinement is the gradual building of a model, where details of the model are added gradually at different refinements levels. Decomposition is dividing the model into sub-models, where each can be refined independently. Writing Event-B models and verifying them is supported by a tool called, Rodin (Abrial et al., 2005, 2010).

The integration of formal methods with semi-formal representations can be the solution to the complexity of formal modelling, on one hand; and the lack of formality in semi-formal methods, on the other hand. Many researchers have studied such integration with a variety of formalisms. Of direct relevance to our work is UML-B, which is a UML-like representation based on Event-B (Snook and Butler, 2008; Said et al., 2009).

The Event Refinement Structures (ERS) approach, introduced by Butler (Butler, 2009a) is a hierarchical graphical representation that aims at structuring Event-B refinements and explicitly describe the ordering of events. The control flow of events is done implicitly.
in Event-B using guards, which are event enabling conditions. Semantics is given to an ERS diagram by transforming it into an Event-B model using some transformation rules (Salehi Fathabadi et al., 2012).

1.1 Aims and Contributions

During recent years workflow modelling with formal semantics has been a subject of interest in both industry and academia, resulting in a diversity of workflow modelling approaches. Nowadays, BPMN (Chinosi and Trombetta, 2012), for Business Process Model and Notation, is adopted as a de-facto language for business process modelling. UML (Rumbaugh et al., 1998) diagrams such as activity and state-machine diagrams have also been used widely for workflow modelling. However, both BPMN and UML have been criticised for their lack of formal semantics. This resulted in many attempts to give these modelling languages precise semantics by integrating them with different formal methods, such as the work in Börger and Sörensen (2011); Storrle (2004).

Petri Nets (Murata, 1989), is a formal method which has also been used for workflow modelling due to its graphical nature, its state-based approach, in addition to the abundance of its analysis techniques (van der Aalst, 1998). YAWL (van der Aalst and ter Hofstede, 2005), for Yet Another Workflow Language, inspired by Petri Nets, was introduced to address the Petri Nets limitations in workflow modelling, which are limitations in multiple instances modelling, complex synchronisations and cancellation patterns. However, YAWL is still criticised for many reasons such as its complex formal semantics (Börger, 2012).

All the previously mentioned workflow modelling approaches have limitations in some aspects. When modelling a complex workflow, we are looking for an approach that supports the following criteria:

1. A range of control structures covering sequencing, choice and parallelism with precise semantics.
2. Flexible control flow that supports dynamic changes in the degree of concurrency.
3. Data handling, due to the influence data can have on the course of execution of activities.
4. Exception handling mechanisms.
5. Incremental modelling and verification to reduce the modelling complexity and ensure the consistency of the workflow models.
6. Graphical representation to facilitate the understanding of the workflow models and for easier communication.
In this thesis, we show how ERS, an approach originally designed for decomposing the atomicity of events in further refinements, can also be used at a high level to model workflows and help overcome some of the challenges of workflow modelling. Our workflow modelling approach uses ERS to model control flow, while data handling is done using Event-B.

**Why Event-B?** Event-B (Abrial, 2010), is a formal method with a formal modelling language and tool support for formal verification. Using Event-B supports the incremental modelling of workflows through refinement, and also supports the verification of invariants and refinement correctness. In addition, various plug-ins extend the Event-B tool, Rodin (Abrial et al., 2006), with several validation and verification techniques. The language of Event-B, which is based on set theory and first order logic, provides comprehensive support for data handling.

**Why ERS?** An Event-B model consists of a collection of guarded atomic actions (i.e. events). In Event-B, event execution is interleaved and there is no explicit control flow. ERS (Fathabadi et al., 2014) augments Event-B with explicit control flow structures, in addition to graphical representation of control flow. The ERS control flow combinators support sequencing, iteration, choice, synchronisation and concurrency.

In order to address our workflow modelling criteria, we introduce two types of extensions to the original ERS approach:

- **First:** Extensions related to dynamic workflow modelling, where we provide support to data-dependent workflows in which case, data changes are possible. The extensions include introducing a new combinator called the *par-replicator* that support replication of behaviour with the possibility of providing new data values while executing. Extending the parameter definition to allow parameter ranges to be defined by expressions on the state, which replaces the original ERS parameter which only uses static sets. We also generalise the *loop*, which allows zero or more executions of an activity, to allow non-deterministic executions of events.

- **Second:** The exception handling extension, where we introduce two new combinators (*interrupt* and *retry*). These two new combinators provide the flexibility to allow one flow to interrupt and block the execution of another flow, while *retry* provides an additional functionality that permits handling the exception by executing the expected normal flow again.

In order to validate our workflow modelling approach and ERS extensions, we apply them to two complex workflows, the fire dispatch workflow and the travel agency booking system. The case studies require treatment of exceptions. To an extent ERS already deals with exceptional behaviour by using *xor* (a mutual exclusive constructor) or by using separate diagrams. However, we want something more general like the ability to
allow one flow to interrupt another and represent changes to on-going workflows such as:

- Ability to change the action plan and respond to those changes after the action plan starts execution.
- Ability to model one workflow instance impacting another workflow instance, e.g., re-allocation of resources from a low priority incident to a high priority incident.
- Ability to interrupt the execution of the workflow due to an exception, and how to handle such exceptions.

In our case studies, the fire dispatch system and the travel agency booking system, we show how ERS is used to model the control flow and how we use Event-B to handle data. We also illustrate how we can combine ERS with other graphical approaches like state machines.

The ERS approach gets its semantics by translating the visual representation of the model to the Event-B language. For this reason, we present formally the translation rules of the new ERS extensions to Event-B. Some of these translation rules are already integrated to the ERS plug-in.

1.2 Research and Modelling Methodology

Our research focuses at facilitating the Event-B modelling and the methodology followed in developing our approach can be summarised by the following activities which are repeated in an iterative manner:

1. Model case study using Event-B and original ERS.
2. Identify limitations.
3. Propose extensions based on limitations and literature review.
4. Revisit case study by applying extensions.
5. Extend ERS language and include extensions to ERS tool.
6. Validate extensions by applying them to new case studies.

In modelling the case studies we adopt the following modelling methodology:
1. Represent the control-flow requirements using ERS diagrams and analyse the Event-B refinement levels resulting from the ERS diagrams to come up with a final version of an ERS representation of the case study. This stage is merely a diagrammatic representation of the model and does not involve the application of the ERS tool to generate Event-B models.

2. During the analysis stage of the ERS diagrams, make modelling decisions about where (i.e., at which refinement level) to introduce some data related requirements and what are the possible ERS combinators that can be applied.

3. Using the ERS tool, add the ERS diagram and generate the Event-B model at each refinement level of the ERS representation of the model.

4. At each refinement level, after generating the Event-B model from the ERS diagram, add manually using the Event-B editor any required data elements that are not part of the ERS control flow, before moving to the next refinement level.

5. After each Event-B refinement, verify any proof obligation that is not automatically discharged by the Rodin Provers. This can sometimes lead to some changes to the Event-B models in order to discharge the Event-B proof obligations. However, the Event-B parts that are generated by the ERS tool cannot be modified, except the events can allow the addition of new guards and actions.

6. Finally, we validate the Event-B models using ProB (Leuschel and Butler, 2003), which is an animator and model checker for the B-Method and it also supports Event-B. In modelling the case studies, we used ProB to check for any inconsistencies in the models, which may also result in some modelling changes.

### 1.3 Thesis Organisation

This thesis is divided into the following chapters:

Chapter 2 gives an overview of some of the popular specification techniques and graphical approaches for workflow modelling.

Chapter 3 provides the background information needed to understand the work done in this thesis, which includes formal methods, Event-B, and the ERS approach.

The following chapters represent the main contributions of this thesis:

Chapter 4 presents the dynamic extensions we suggest to enhance the ERS approach for workflow modelling. These extensions include adding a new pattern called the par-replicator to the ERS combinators, generalising the loop combinator to include non-determinism in execution and finally extending the parameter definition to allow the use
of variables. We also provide a concise tabular summary of the enabling and disabling conditions related to the ERS combinators.

Chapter 5 introduces the fire dispatch case study and how we applied our workflow approach to formally model the case study. In this case study, we apply our dynamic extensions to ERS, where we highlight some limitations in ERS, which we address in Chapter 6.

Chapter 6 addresses the ERS limitations in exception handling and introduces two new combinators to address these limitations, which are also applied to the fire dispatch case study, introduced in Chapter 5.

In Chapter 7 we apply our workflow modelling to the on-line booking system of a travel agency, where we also apply some of our ERS extensions.

In Chapter 8, we formalise the extensions introduced in chapters 4 and 6. We use the formal syntax of “Augmented Backus–Naur Form” (ABNF) in the form of translation rules mapping the ERS elements to Event-B elements. We also explain how these new translations are added to the improved ERS plug-in.

Finally, Chapter 9 evaluates our workflow approach against published workflow criteria and presents our conclusions and plans for future work. We also compare ERS with other workflow modelling approaches and other techniques integrating the formal methods with semi-formal representations.
2.1 Introduction

Workflow Modelling can be associated with different concepts like requirements engineering, system specification, formal modelling, business management, and graphical representations. All these concepts aim at better understanding and analysis of the requirements in order to produce a system that conforms with the customers expectations and requirements.

In this chapter we will present some of the existing graphical workflow modelling approaches and specification techniques that are related to our work.

2.2 Jackson Structure Diagrams (JSD)

Jackson Structure Diagrams (JSD) (Jackson, 1983), are tree-like diagrams that aim at showing the ordering of an entity’s actions in a hierarchical way. The diagram can describe sequencing, iteration and selection. JSD diagrams also have a special notation for null actions.

Figure 2.1 shows an example of a structure diagram. The root of the tree “Order” represents the entity’s name and the leaves of the tree must be actions of that entity. The sequencing is read from left to right. The asterisk in “Delay” indicates iteration. The circle in “Allocate” indicates selection, where exactly one child can be selected. The dash represents a null or empty action where nothing happens. The null action must be part of a selection and it must be a leaf node.
In Jackson structure diagrams, concurrency cannot be represented in a single diagram. It can be done using more than one diagram with common actions. Similarly, a premature termination of an entity’s life cannot be easily represented in structure diagrams. To express this kind of behaviour, a special kind of selection is required. Figure 2.2 shows an example of premature termination as a result of natural death in a soldier’s career promotion (Jackson, 1983). The normal structure of a complete career of a soldier is augmented by a choice at the beginning between a normal complete career and a prematurely terminated career. There is also an additional feature represented by an iteration over the normal events (Career Event) in order to avoid representing the several options of a prematurely terminated career.

Unified Modelling Language (UML) (Rumbaugh et al., 1998), is a graphical representation for modelling object-oriented systems. UML consists of many graphical types representing objects, behaviours, and states. Version 2.0. of UML includes thirteen diagrams divided into two semantics groups (OMG, 2013):
• Structure Semantics: Describes the structure of the model and this category includes six diagrams: Class Diagram, Object Diagram, Component Diagram, Composite Structure Diagram, Package Diagram, and Deployment Diagram.

• Behaviour Semantics: Describes the behaviour or the dynamics of the model and it can be divided into two sub-categories. The first is Behaviour Diagrams that include three types: Use Case Diagram, Activity Diagram, and State Machine Diagram. The second is Interaction Diagrams, which include four diagram types: Sequence Diagram, Communication Diagram, Timing Diagram, and Interaction Overview Diagram.

In the following sections, we will present two UML diagrams with behavioural semantics, activity diagrams and state machine diagrams.

2.3.1 Activity Diagrams

A UML activity diagram is one of the UML behaviour diagrams. An activity diagram is a directed graph in which the direction of the arrows suggests the flow of activities. In older versions of UML, activity diagrams were considered as a type of state machine diagram, but since the introduction of version 2.0, activity diagram semantics is widely enriched, based on token flow inspired by Petri Nets, described in more detail in Section 2.5. Despite this fact, UML activity diagrams still lack a complete formal definition and that is why many researchers have tried to define a formal semantics of UML activity diagrams in different ways (omg.org, 2012).

Figure 2.3 describes some of the notation of UML activity diagrams. Activities and actions are connected through a range of flow types including decision points, merges, forks and joins. Activities may be nested hierarchically. Figure 2.4 shows a simple example of an order activity using some of these notation. The start of the order activity is indicated through the use of the initial node with a customer making an order. The directed arrow indicates that the next action will be receiving the order in the order department. After receiving the order, a decision should be taken to whether accept it or not. If it is accepted, then it will continue to the fork-node, which means the execution of the actions Send Invoice followed by Receive Payment, in parallel with the execution of Ship Order Action. The join-node waits for both Receive Payment and Ship Order to finish, to proceed to Close Order action, and finally closing the order activity using the activity final node. In the case of rejection decision, the order will be closed immediately. The use of the merge-node before the Close Order action, means this action will execute if any of the previously described scenarios take place.
2.3.2 State Machines

A State machine (omg.org, 2012) is another UML behavioural diagram that describes the sequence of states an individual object goes through. States and transitions are important concepts in a state machine. A state models a situation in an object’s life,
for example a telephone in an idle state represents the situation when a telephone is not in use. A transition is a directed arc connecting two vertices, where a vertex can be a state or a pseudostate such as a choice or fork etc. A transition can indicate that an object in a source state may perform certain actions resulting in changing the state of the object to the target state as a result of an event. The source and target states of a transition might be the same, in this case it is called self-transition. A transition can be also associated with guards which are boolean conditions that must evaluate to true before a transition can execute.

States can be one of the following three types:

- **Simple State**: Has no internal sub-states or transitions.
- **Composite State**: can be composed of one or more regions. A region is delineated by a dashed line to describe the behaviour of a fragment that may execute concurrently with its orthogonal regions (The term orthogonal refers to regions owned by the same state or state machine). Any state within a region of a composite state is called a sub-state.
- **Submachine State**: is an entire state machine nested within a state.

Initial and final states are special states that respectively indicate the start and completion of a state machine. Figure 2.5 from omg.org (2012) shows an example of composite states, where “CourseAttempt” is a composite state with a single region and “Studying” is a composite state with three orthogonal regions that may execute in parallel.

![Figure 2.5: Composite State With Regions (omg.org, 2012)](image)

2.4 Business Process Model and Notation (BPMN)

BPMN (Object Management Group, 2011), refers to “Business Process Model and Notation” or “Business Process Modeling Notation”. BPMN is a graphical notation which
is nowadays adopted as the de facto for business process modelling (Chinosi and Trombettta, 2012).

BPMN aims at modelling control flow rather than data flow, although it supports flow of messages and associations between activities and data. The main goal of BPMN is to provide a standard notation that can be easily understood by the different parties involved in the business process. Early versions of BPMN were inspired by UML activity diagrams, but lacked a well defined semantics. Then came BPMN 2.0 to extend the notation and provide a formal execution semantics (Object Management Group, 2011; Ouyang et al., 2009). However, BPMN is still criticised for its ambiguous and imprecise semantics (Börger, 2012).

BPMN has five categories of elements: Flow Objects, Data, Connecting Objects, Swimlanes and Artifacts. The Flow Objects category is the main category in any business process, and can be further categorised into three elements (Object Management Group, 2011):

1. **Events**: Can be a “Start”, an “Intermediate” or an “End” event. An event represents something that **happens** during a process and it is represented as a circle.

2. **Activities**: Can be atomic or non-atomic. An atomic activity is called a **task** and it represents the work to be performed within the business process. An activity is represented by a rectangle with round corners. A task can be also marked by any one or two types of markers which are: Loop, Compensation and Multi-Instance. “Compensation” is an important aspect of BPMN which is concerned with undoing the steps that successfully happened and are later cancelled.

3. **Gateways**: Control the divergence or convergence of flows and are represented by diamond shapes.

Figure 2.6 from Object Management Group (2010) shows a BPMN representation of an “order” process. In this diagram the circle with the thin border represents a start event and the circle with a thick dark border represents an end event. The “Approver Order” activity, containing the activities “Approve Customer” and “Approve Product”, shows an example of a non-atomic activity or a sub-process. The blank diamond shape represents an exclusive gateway or a decision gateway. The diamond with a plus sign in the middle represents a parallel gateway, the first parallel gateway is a parallel fork and the second one is a parallel join.
Petri Nets (Murata, 1989) are directed graphs used mainly for modelling concurrent and asynchronous systems. Petri Nets consist of two types of disjoint nodes called places and transitions, connected to each other by edges called arcs, which can only connect nodes of different types. Transitions denoted by solid bars or rectangles, can be interpreted as actions or events. Places are conditions for actions, and they are represented as circles. The dynamics of the system can be described using tokens, which are represented by solid dots in places. The presence of a token in a place represents the enabling condition of a transition. It is also possible to describe the behaviour of the system using mathematical models (Murata, 1989).

Petri Nets have been used for workflow modelling for three main reasons (van der Aalst, 1998; van der Aalst and ter Hofstede, 2005):

1. It has formal semantics in addition to the graphical nature.
2. It is a state-based approach, where the states of a system are explicitly modelled.
3. There are model checking techniques available for Petri Nets.

There are many extensions to Petri Nets, usually referred to as high level Petri Nets. For example, Coloured Petri Nets (Jensen, 1997), which extends the basic Petri Nets by allowing the tokens to carry data values of different types.

2.6 Yet Another Workflow Language (YAWL)

YAWL (van der Aalst and ter Hofstede, 2005) stands for “Yet Another Workflow Language”, is a workflow language as its name suggests. YAWL was introduced to cover the twenty workflow patterns described in van der Aalst et al. (2003). YAWL is inspired by “Petri Nets”, but extended to overcome the limitations of Petri Nets in workflow modelling. Although, YAWL is inspired by Petri Nets, it is considered as a different language with different semantics.
The three limitations in Petri Nets that YAWL tries to solve are limitations in complex synchronisation, limitations in multiple instances modelling and finally limitations in cancellation patterns.

The twenty workflow patterns that YAWL tries to cover can be categorised into six categories (van der Aalst and ter Hofstede, 2005):

1. Basic control flow patterns
2. Advanced branching and synchronisation patterns
3. Structural patterns
4. Pattern involving multiple instances
5. State-based patterns
6. Cancellation patterns

Only one of the twenty patterns is not supported by YAWL which is “implicit termination”, which falls under the structural patterns category.

Figure 2.7 from van der Aalst and ter Hofstede (2005), shows the modelling notations used by YAWL. The modelling elements of YAWL include conditions; AND, XOR, OR splits and joins, atomic and composite tasks in both single instance and multiple instances cases; in addition to “remove tokens” which can be used to model cancellation.

YAWL defines four additional parameters to multiple instances tasks: lower bound, upper bound, threshold and static/dynamic parameter. Figure 2.8 from van der Aalst
and ter Hofstede (2005) represents different scenarios of using multiple instances. The first case, “a”, requires between 1 and 10 instances to process witness statements and does not allow the creation of new instances while being processed, since it is described as “static”. Enabling the “archive” task in the first case requires the execution of all instances. In the second case, “b”, new instances can be created while processing witness statements due to the dynamic parameter. Lastly, Case “c” enables the “archive” task if all instances or at least three instances, which is the threshold, have executed. All these cases can execute the multiple instances in parallel.

![Figure 2.8: Examples of Multiple Instances Tasks Using YAWL (van der Aalst and ter Hofstede, 2005)](image)

2.7 Formal Modelling of Workflows

The need to have precise semantics for workflows have resulted in several works formalising different workflow approaches. In this section we present some of the works that gave workflows formal semantics:

2.7.1 Formalising Control Flow Patterns Using Abstract State Machine (ASM)

ASM or Abstract State Machine (Börger, 2005), is a rule based modelling approach. In Börger (2007a), Börger uses ASM to represent the forty three workflow patterns defined in Russell et al. (2006a) by the business process modelling centre. However, Börger finds out using ASM, the forty three workflow patterns can be reduced into eight
generic patterns out of which more complex patterns can be derived using parameter instantiation, composition and refinement.

The generic eight workflow patterns are described using ASM in Börger (2007b). The eight basic patterns are divided into two categories, sequential and parallel, each containing four patterns. The sequential category includes the following four patterns:

1. **Sequence Patterns**: Deals with sequencing of multiple flows, where an activity is enabled after the completion of another activity.

2. **Iteration Patterns**: Where an activity is repeatedly executed.

3. **Begin/Termination Patterns**: Where a flow is terminated depending on a specified stopping criteria. Börger also relates the termination pattern with the cancellation patterns.

4. **Selection Patterns**: Includes a choice among multiple flows.

The parallel category includes:

1. **Parallel Split Patterns**: Where one flow splits into multiple independent flows.

2. **Merge Patterns**: Where multiple flows converges into a single thread of control.

3. **Interleaving Patterns**: Where activities are executed one after another in an arbitrary order.

4. **Trigger Patterns**: The ability of an activity to be triggered by another part of the process or by an external environment.

In the eight different workflow patterns, Börger does not differentiate between single and multiple instances behaviour or between static and dynamic behaviour, i.e. whether the number of branches or instances is known or unknown until runtime, are left as a matter of how the parameter is declared.

### 2.7.2 Other Work on Formalising Workflows

Wang et al. (2010) describe a guarded workflow language based on the CSP (Communicating Sequential Processes) (Hoare et al., 1985) Notation. The actions in this formal language are guarded transactions which in turn will help in determining the pre-conditions for a successful completion of a workflow or a combination of workflows.

Butler et al. (2005a) use the StAC (Structured Activity Compensation) language (Butler and Ferreira, 2000), which is also based on process algebras such as CSP (Hoare
et al., 1985). Similar to CSP, StAC supports operators for sequencing, concurrency, choice and hiding, but also provides support for exception handling using compensation. In Butler et al. (2005a), they use StAC for activity orchestration and the B formal method (Abrial, 1996) for data handling. They also map BPEL (OASIS, 2007), which is the Business Process Execution Language for Web Services to StAC, thus giving BPEL formal semantics.

Other works have also given BPMN formal semantics such as Bryans and Wei (2010) and Börger and Sörensen (2011). The former translated BPMN to the Event-B formal language, and the latter translated BPMN to ASM. Others have also given activity diagrams formal semantics. As a natural sequence of the activity diagrams semantics which is based on Petri Nets, Syriani and Ergin (2012) translated activity diagrams to Petri Nets, and in Börger et al. (2000), activity diagrams are given formal semantics using ASM.

In addition to formalising workflow modelling approaches, others have used a combination of state-based and event-based formal methods, such as CSP and B in Butler (2000) and Circus (Woodcock and Cavalcanti, 2002) which is a combination of Z (Spivey, 1992) and CSP. In these approaches, the event-based CSP is used to formally represent control flow.

2.8 Conclusion

In this chapter, we have presented some of the main specification and workflow modelling approaches. All approaches cover the basic control flow constructs, but their strengths vary when it comes to complex cases such as multiple instances and cancellations.

Jackson Structure Diagrams (JSD) can simplify complex problems by applying a hierarchical structure. UML diagrams and BPMN are popular modelling techniques, but are still criticised for their imprecise semantics. Advocates of Petri Nets praise its graphical nature as well as its formal semantics. Then, YAWL comes to address the limitations in Petri Nets when it comes to modelling multiple instances, complex synchronisation and cancellation. However, YAWL is still criticised for its complex formal semantics. That is why, in this thesis we focus on a structured workflow modelling of multiple instances using formal semantics.
Chapter 3

Formal Modelling Approaches

3.1 Introduction

In this chapter we introduce some key concepts in formal methods needed to understand our work. First we start with introducing formal methods and an overview of some formal modelling languages. Then we introduce Event-B, a formal method for system modelling. We describe the structure, refinement, proof obligation rules and finally the tool support of the Event-B formal method and one of its important plug-ins UML-B. We also introduce the Event Refinement Structures approach (ERS), a diagrammatic representation that we use to model workflows in Event-B. We present the different patterns of ERS, from their diagrammatic notation to their Event-B semantics.

3.2 Formal Methods

Formal methods are mathematical techniques used for the systematic specification, verification and development of both hardware and software systems (Hall, 1990; Wing, 1990; Bowen and Hinchey, 1995). Similar to blueprints used in other engineering disciplines, formal models can be used to formally reason about the system during its construction (Abrial, 2007). This will mainly help in finding some errors during the construction of the system, instead of delaying it to the testing phase.

Nevertheless, many people are still questioning the practicality of formal methods. Opponents of formal methods usually claim that they are difficult and only mathematicians can use them. However, according to Abrial (2007), some of the real difficulties in formal methods lie in the lack of good requirement documents, and the common practice of rushing towards coding without spending enough time to reason about the system. On the other hand, semi-formal notations which are graphical representations of the system are criticised for their lack of formal semantics. Therefore, to tackle the issues of
mathematical difficulty and lack of formal semantics, in this thesis we use a combination of a graphical representation and a formal method.

### 3.2.1 Overview of Some Formal Methods

In this section, we will discuss briefly some formal methods that have been used in industry, in particular VDM, Z, the B-Method and ASM. Later in Section 3.3, we will describe in more detail Event-B, which is the formal method, we use in modelling our case studies.

- **VDM** or Vienna Development Method (Jones, 1990), was developed in an industrial environment at the IBM Laboratory, Vienna. A VDM specification describes the behaviour of the system using operations. These operations are defined using pre- and post-conditions, which describe the state variables of the system before and after the operation (Lindsay et al., 1991).

  One of the key features of **VDM** is **data reification**, this involves the transition from abstract data types to concrete data types, or in other words data types that are more complex and includes more detail. This transformation requires several steps until the data types can be expressed in the implementation language. In order to relate the abstract data types with the corresponding concrete data types, a function called a “retrieve function” is required, resulting in some proofs to justify the correctness of data reification (Jones, 1990).

  Another key feature is **operation decomposition**, which is required after the data types are refined into the level of an implementation language. This results in the decomposition of operations into combinators that combine the different types of operations, e.g., while loop. Operation decomposition also entails more proofs to justify its correctness (Jones, 1990).

- **Z** pronounced as “Zed”, similar to **VDM**, it is based on set theory and first order predicate logic. However, **Z** includes **schema notation** (Figure 3.1), which improves the structuring and the readability of the specifications, but unlike **VDM**, **Z** does not explicitly separate pre- and post-conditions of the operations (Bowen, 2003).

  $$
  \begin{array}{l}
  BirthdayBook \\
  known : \mathcal{P} \text{NAME} \\
  birthday : \text{NAME} \rightarrow \text{DATE} \\
  known = \text{dom} \cdot birthday
  \end{array}
  $$

  Figure 3.1: Example of Z Schema Notation (Bowen, 2003)

Figure 3.1 shows an example of a schema notation, where the top part of the box describes the variables and their associated constraints defining their types, while the bottom half describes additional constraints between these variables.
Z is more considered as a formal specification notation rather than a formal method, and in industry Z is mainly used for specification rather than proof (Bowen, 2003). When we say formal specification, we mean a mathematical representation of the customer’s requirements in order to remove ambiguities, whereas a formal method includes both a formal specification and a framework to develop a specification towards an implementation (Wing, 1990; Jones, 1990).

- **B-Method**, or **Classical B** was first introduced by Abrial (Abrial, 1996) for software development. Similar to Z and VDM, the B-Method is based on set theory and first order predicate logic. From a practical point of view, the main difference is that B has abstract assignments rather than postconditions. But unlike VDM, which uses three-valued logic to describe undefinedness, the B-Method uses the classical two-valued logic (True and False) (Cansell and Mery, 2003).

The structure of the B-Method is based on **machines** (Figure 3.2) and refinement (Cansell and Mery, 2003; Butler et al., 2005c), refinement will be explained in more detail in Section 3.3.2.

![Figure 3.2: B-Method Abstract Machine (Cansell and Mery, 2003)](image)

Figure 3.2, describes an abstract machine “m”. The “sets” and “constants” clauses define the sets and the constants of the system respectively. The “properties” clause contains definitions of the constants. The “variables” describe the state of the system and the “invariants” define the properties of these variables, where “assertions” describe the safety properties of the variables. The “initialisation” gives the variables an initial value to describe the initial state of the system. Finally, “operations” can change the state of the variables but need to always preserve the invariants.
• ASM or Abstract State Machine (Börger, 2005), is a formal method that extends the Finite State Machine (FSM) with an enriched notion of state, where a state is an arbitrary data structure. The ASM method starts with an abstract model called ground model, and then more detail is added through stepwise refinement.

ASM has a simple syntax in the form of pseudo code, and is defined as a set of rules called ASM rules with the general form (Börger, 2003, 2005):

\[
\text{if Condition then Updates}
\]

The condition, also called guard, evaluates to true or false, and the Updates is a finite set of assignment statements. Simultaneous and non-deterministic execution of ASM rules can also be represented using the following notation, where rule(x) should satisfy the condition \( \varphi \):

Simultaneous Execution: \( \forall x \text{ with } \varphi \) \( \text{rule}(x) \)

Non-Deterministic Execution: \( \text{choose } x \text{ with } \varphi \) \( \text{rule}(x) \)

### 3.3 Event-B

Event-B is a formal method for modelling systems that may include hardware and mechanical components as well as software (Abrial, 2010). Event-B is mainly influenced by the B-Method (Abrial, 1996) and Action Systems (Back, 1990). The Event-B language is based on set theory and first order logic. An Event-B model starts with an abstract machine which is linked to a series of concrete machines through refinement. Correctness and consistency of models, are proved by means of mathematical proofs (Abrial et al., 2010).

#### 3.3.1 Structure

The Event-B model, unlike the B-Method, is divided into a static part, the “Context” and a dynamic part, the “Machine”. The context consists of the set types, constants and their properties as axioms. A machine consists of variables, invariants describing the constraints applied on the variables, and events that show how the variables’ states change provided some guard conditions hold. In addition, theorems can be defined in both context and machine (Metayer et al., 2005; Abrial, 2010).

Figure 3.3 presents a simple Event-B model showing both the static and the dynamic parts of the model. Machine \( M \), on the left, consists of two events: the initialisation event and “Event1”. The variable \( p\_set \) is defined by the invariant \( \text{inv}_1 \) as a subset
of the set $P_SET$. The set $P_SET$ is defined in the context $C$, where the “sees” clause gives machine $M$, access to the context $C$. The event $Event1$ updates the variable $p_set$, by adding new values in the action ($act1$), but first checks the type of the newly added values by the guard ($grd1$).

![Figure 3.3: Simple Event-B Model](image)

### 3.3.2 Refinement

Refinement is a key concept in the Event-B modelling, where instead of building a single model, we start with an abstract one, focusing on the main functionalities of the system. Then during refinement levels, details of the model and/or new functionalities are added gradually. The refinement process aids understanding the model and proving its correctness and consistency (Abrial and Hallerstede, 2007).

Refinement can be categorised into two types, horizontal refinement and vertical refinement (Damchoom and Butler, 2009). Horizontal refinement includes the introduction of new features or properties to the model without changing the previously existing ones, resulting in the addition of new events and variables to the model (Damchoom et al., 2008; Damchoom and Butler, 2009). On the other hand, vertical refinement aims at changing the structure of the abstract model using data refinement (Damchoom and Butler, 2009). Data refinement is used to change the state of the variable in a way that makes its implementation using a programming language possible (Abrial and Hallerstede, 2007).

In Event-B, refinement is described using the “refines” clause in a machine. In case of refinement, a special type of invariant, called “gluing invariant”, can be used. Gluing invariants describe the relation between the abstract and the concrete, or refining, variables. An abstract event can be refined by one or more concrete events and new events
can be added. In Event-B the variables of the abstract machine can be inherited by its refining machine which can also add new variables or even drop some of its abstract machine variables. Dropping variables in refinement requires gluing invariants. Similarly, where parameters are dropped in the concrete machine, “Witnesses” for them should be added to the refining events, relating the refining parameters to their corresponding abstract parameters in the same way gluing invariants relate abstract and concrete variables, thus making the refinement proof obligations easier to discharge (Abrial, 2010; Abrial and Hallerstede, 2007).

The new events added during refinement, refine skip which is an implicit event that does nothing at the abstract level. This allows a refining event to modify variables introduced in the refinement as if they make no change at the abstract level. An event’s status can also be described using the “status” clause as ordinary, convergent or anticipated. A convergent event is an event which cannot take control forever, while an anticipated event is typically refined into a convergent event. In case of convergent or anticipated events, a variant is needed. A variant must be a natural number or a finite set, which a convergent event must decrease and an anticipated event must not increase. An ordinary event is the most commonly used, it is an event that is neither convergent nor anticipated (Abrial, 2010).

When applying machine refinement, a machine can only refine one other machine and so on, where one can construct a chain of refinements. Machine refinement results in additional proof obligations, including guard strengthening, invariant preservation, action simulation and witness feasibility (Hallerstede et al., 2010).

Unlike a machine, a context can extend more than one other context. This extension can include adding new sets, constants defined by axioms, but does not result in associated proof obligations (Abrial and Hallerstede, 2007). Figure 3.4 adopted from Butler (2013) describes the different relationships between machines and contexts. The dashed line indicates an implicit “sees” between Machine M2 and Context C1, since Machine M2 sees Context C2 which extends Context C1.

Figure 3.4: Machine and Context Relationships
3.3.3 Proof Obligation Rules

Proof Obligations are used in proving the consistency properties of the Event-B model and the consistency between refinement levels. These proof obligations are automatically generated by the Rodin tool, and can be proved both automatically and interactively (Abrial et al., 2006, 2010). Proof Obligations are concerned with model consistency, refinement, theorems, convergence, and well-definedness of the model. In this thesis we make a lot of use of Event-B and proof, so a brief description of the main proof obligation rules is described in Table 3.1. Regarding the variant proof obligations, numeric variant (NAT) and finite set variant (FIN), only one of these proof obligations can be generated for a given variant, since a variant can be either numeric or a finite set.

<table>
<thead>
<tr>
<th>Proof Obligation Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Invariant Preservation (INV)</td>
<td>Ensures that invariants are maintained by all events</td>
</tr>
<tr>
<td>Feasibility (FIS)</td>
<td>Ensures that a non-deterministic action is feasible when its guards are True</td>
</tr>
<tr>
<td>Guard Strengthening (GRD)</td>
<td>Ensures that the guard of the concrete event is stronger than the guard of its corresponding abstract event</td>
</tr>
<tr>
<td>Simulation (SIM)</td>
<td>Ensures that all actions in abstract events are properly simulated in the corresponding concrete events</td>
</tr>
<tr>
<td>Numeric Variant (NAT)</td>
<td>Ensures that the numeric Variant of a convergent or anticipated event is a natural number under their guards</td>
</tr>
<tr>
<td>Finite Set Variant (FIN)</td>
<td>Ensures that the Variant of a convergent or anticipated event is a finite set under their guards</td>
</tr>
<tr>
<td>Variant (Var)</td>
<td>Ensures that the convergent event decreased the numeric or finite set variant</td>
</tr>
<tr>
<td>Non-Deterministic Witness (WFIS)</td>
<td>Ensures the existence of all witnesses proposed in a concrete event</td>
</tr>
<tr>
<td>Theorem (THM)</td>
<td>Ensures the provability of theorems described in contexts and machines</td>
</tr>
<tr>
<td>Well-Definedness (WD)</td>
<td>Ensures that all elements of the model are well defined e.g. we cannot divide by zero</td>
</tr>
<tr>
<td>Preserved Variable (EQL)</td>
<td>Ensures that the value of a variable at the abstract event does not change, if assigned a value at a concrete new event</td>
</tr>
</tbody>
</table>

Table 3.1: Event-B Proof Obligation Rules (Abrial, 2010)

3.3.4 Rodin: An Event-B Modelling Tool

Rodin is a tool for Event-B that generates proof obligations and automatically discharge the trivial ones (Butler and Hallerstede, 2007; Abrial et al., 2010).
Rodin is an open tool, implemented on top of the Eclipse Framework (Butler and Hallerstedt, 2007). The openness of this tool allows other parties to contribute to this tool and extend it, through the use of plug-ins (Abrial et al., 2006; Butler and Hallerstedt, 2007). UML-B is one of these important plug-ins that integrates semi-formal modelling with Event-B. This plug-in is explained in more detail in Section 3.3.5.

The graphical user interface of Rodin, allows the user to easily switch the perspective between modelling and proving (Abrial et al., 2010; Butler and Hallerstedt, 2007).

The Rodin tool consists of three major components (Abrial et al., 2006, 2010):

- **Static Checker**: Performs syntax and type checking of the contexts and machines.
- **Proof Obligation Generator**: Generates proof obligations automatically. These proof obligations are described in more details in Section 3.3.3.
- **Proof Obligation Manager**: Keeps track of proof obligations and associated proofs. It also discharges proof obligations automatically, and allows the user to interactively prove un-discharged proofs.

### 3.3.5 UML-B

UML-B combines both graphical representation and formal modelling using Event-B. UML-B is a UML-like formal modelling language based on Event-B (Snook and Butler, 2008; Said et al., 2009). UML-B is first introduced by Snook, as a UML profile that is translated to classical B, or B-Method explained in Section 3.2.1, with the help of U2B tool (Snook and Butler, 2006; Hallerstedt and Snook, 2011). UML-B is supported by the UML-B tool, which is an extension of the Rodin tool, generating Event-B models from the corresponding UML-B diagrams. Event-B translation of the state machines in UML-B can be done as disjoint sets or state functions representations (Said et al., 2009).

UML-B includes the following four diagram types (Snook and Butler, 2008):

- **Package Diagrams**: Based on UML package diagrams, these describe the relationships between contexts and machines.
- **Context Diagrams**: Based on UML class diagrams, these describe the static part or the context of the model.
- **Class Diagrams**: Classes are described as variables subset of the types or class types introduced in the context.
• **State Machine Diagrams:** Are attached to the classes of the class diagram and introduce a variable that partitions the behaviour of the class. The transitions in the state machine are translated into events in the Event-B machine that describe the change of states.

**Refinement** in UML-B is based on Event-B refinement, it can be either a class refinement or a state machine refinement. Both of these two types of refinement can be described as node refinement (Hallerstede and Snook, 2011). In a class refinement, classes and attributes can be inherited, added, or dropped variables as in Event-B refinement. Events are refined in the same way as Event-B refinement and new events can be introduced. State machine refinement can refine states or state machines. This can be done through state elaboration or transition elaboration. In a state elaboration a super state can be refined into sub-states resulting in the addition of new invariants. In transition elaboration, a transition is replaced by several transitions similar to event refinement in Event-B, where an event can be refined by one or more events and new events can be added refining skip (Said et al., 2009).

A new refinement method for UML-B is introduced in Said et al. (2013), called “Context Diagram Extension”. In this method, the class types are extended with new features such as associations, axioms etc. without changing and repeating the old features.

### 3.4 Event Refinement Structures Approach (ERS)

The Event Refinement Structures (ERS) approach was first introduced by Butler (Butler, 2009a). ERS provides a graphical representation based on Jackson Structure Diagrams (JSD) (Jackson, 1983). JSD provides a tree-like graphical representation of structured programs with sequential behaviour indicated by left to right placement of nodes. ERS diagrams help in understanding the relations between the abstract and concrete levels of the Event-B models, and also help to explicitly describe the control flow of events, which is implicitly described through guarded events in an Event-B model (Butler, 2009a).

Figures 3.5 and 3.6 show examples of ERS diagrams, where each node represents an event. The ERS diagram indicates that the atomicity of the abstract event, appearing in the root node, is decomposed into some concrete sub-events, appearing in the leaf nodes. Similar to Jackson’s structure diagrams, the leaf events are read from left to right, so $A$ is executed first, followed by $B$ and finally $C$.

The dashed line represents an event that refines skip, or in other words a newly added event. The solid line represents a refining event, where exactly one event can refine an abstract event. A combinator can also be applied to the events and it can be one of the following: and, or, xor, loop, all, some, one. A combinator can be connected to one or more events depending on the combinator type. We can also add parameters to the
events to indicate multiple instances modelling (Figure 3.6). Multiple instances modelling can be used to model concurrency, where leaf events of different instances of a flow may interleave. These parameters need to be inherited by subsequent children (Butler, 2009a; Salehi Fathabadi et al., 2012).

Semantics is given to the ERS diagram by transforming it into an Event-B model. Salehi Fathabadi et al. (2012) describe some of the rules of the ERS diagram transformation into Event-B modelling language. They apply ERS, or what is previously referred
to as the atomicity decomposition approach, in two case studies (Salehi Fathabadi and Butler, 2010; Salehi Fathabadi et al., 2011) to evaluate its role in clarifying the refinement and the explicit ordering of events in the Event-B modelling language.

Figures 3.5 and 3.6 also show the corresponding Event-B translation of leaf $B$, in the single instance and the multiple instances case respectively. In both cases, leaf $B$ is transformed to an event in Event-B with a corresponding control variable ($B$), that is used to explicitly describe the ordering of the leaf event ($B$). In the single instance case (Figure 3.5) the control variables are represented as boolean flags in Event-B, with the interpretation that the flag is $false$ before the occurrence of the event and is set to $true$ after the event occurs, as shown by the disabling guard ($grd_{self}$) and the action ($act$) of the leaf event $B$. The sequencing guard ($A = TRUE$) checks that event $A$ has executed before $B$ is allowed to execute, which conforms with the left-to-right sequencing of the ERS diagram. The ordering of $B$ is also specified by the sequencing invariant ($inv_{B_{seq}}$).

In the multiple instances case (Figure 3.6), the control variables are defined as subsets of the instance parameter set ($TYPE(p)$). In this case, the first leaf of the ERS diagram ($A$) is defined as a subset set of $TYPE(p)$ ($inv_{A_{type}}$), and $B$ as a subset set of $A$ ($inv_{B_{seq}}$), since the execution of $B$ must follow the execution of $A$. In the corresponding event of leaf $B$, the sequencing guard checks that $A$ has occurred for the parameter value $p$, while the second guard ($grd_{self}$) prevents $B$ from executing again for the same instance value.

### 3.4.1 Event Refinement Structures (ERS) Combinators

In this section, we describe the existing combinators of the ERS approach. For more details, refer to Salehi Fathabadi (2012). The ERS approach defines four groups of combinators:

- **Sequencing Pattern**: Defined by the left to right arrangement of events.
- **Logical Combinators Pattern**: and, or, xor
- **Loop Pattern**
- **Replicator Patterns**: all, some, one

In ERS the combinators are represented within an oval applied to one or more events, where the events are represented within a rectangle. It is also worth noting that, when the lines, whether solid or dashed, originate from a rectangle, this indicates a sequential behaviour; and when they originate from an oval shape, this indicates a non-sequential behaviour, as shown by Figure 3.7.
Chapter 3 Formal Modelling Approaches

The sequencing pattern is described earlier in Section 3.4 (Figures 3.5 and 3.6). ERS also allows the choice between strong sequencing and weak sequencing within different refinement levels. Strong sequencing means sequencing rules are applied to each sub-event, whereas weak sequencing only applies the sequencing rules to the solid line sub-event with sub-events from the previous refinement level. Weak sequencing cannot be applied within the same refinement level.

To make things clearer, consider the example in Figure 3.8, where Event_2 is refined into the sequence of events Event_a, Event_b and Event_c. If weak sequencing interpretation is applied to the refinement of Event_2, then there is no ordering constraints between Event_1 and Event_a and also no ordering constraints between Event_c and Event_3. However, within the same refinement level ordering must be applied, so that Event_a must execute first followed by Event_b then Event_c. Moreover, the solid line of Event_b enforces ordering with sub-events from previous refinements, so Event_b must follow Event_1 and precede Event_3.

Under the weak sequencing interpretation, both of these events traces are possible behaviours of Figure 3.8:

\[ \langle \text{Event}_1, \text{Event}_a, \text{Event}_b, \text{Event}_c, \text{Event}_3 \rangle \]
\[ \langle \text{Event}_a, \text{Event}_1, \text{Event}_b, \text{Event}_3, \text{Event}_c \rangle \]

These are not possible under the strong sequencing interpretation, where the only possible execution trace is:

\[ \langle \text{Event}_1, \text{Event}_a, \text{Event}_b, \text{Event}_c, \text{Event}_3 \rangle \]

The rest of the ERS combinators are described in the following sub-sections:
3.4.1.1 And Combinator

The *and-combinator* is used to describe that the execution of all its sub-events should finish, before the next event can execute. The order in which its sub-events execute is non-deterministic. It should also include at least two sub-events. In Figures 3.9 and 3.10, $C$ can only execute after both $B1$ and $B2$ finish executing.

![Diagram of And Combinator](image)

**Figure 3.9: ERS and-combinator (Single Instance Case)**

Applying the *and-combinator* will affect the sequencing constraint of its following event ($C$). In the single instance case (Figure 3.9), the sequencing invariant ensures that the control variable $C$ cannot be *true*, unless both $B1$ and $B2$ are already set to *true*. The effect of the *and-combinator* also appears in the sequencing guard, which requires the occurrence of both $B1$ and $B2$ ($B1 = TRUE \land B2 = TRUE$) to enable the execution of $C$.

Similarly in the multiple instances case (Figure 3.10), enabling the execution of $C$ requires the execution of both $B1$ and $B2$, which in this case is done by ensuring that the set $C$ is subset of the intersection of the sets $B1$ and $B2$. The gluing invariant in both the single and multiple instances cases ($C = AbstractEvent$) is the result of applying the solid line to $C$, which is also reflected in the event $C$ by using the keyword *refines*.

3.4.1.2 Or Combinator

The *or-combinator* is used to describe multiple choice, in other words at least one of its sub-events is required to execute, but possibly more than one can also execute. In the
Chapter 3 Formal Modelling Approaches

3.4.1.3 Xor Combinator

The xor-combinator is used to describe mutual exclusiveness, in other words exactly one of its sub-events can be executed at any time. Similar to the or-combinator and the and-combinator, it should be applied to two or more sub-events.

Figures 3.13 and 3.14 show the effect of applying the xor-combinator in the single instance case and the multiple instances case respectively. In both the single instance and multiple instances case the sequencing constraint of $C$ is similar to applying the or-combinator. However, the xor-combinator adds additional constraints to its sub-events to ensure that exactly one sub-event can execute at any time. Applying the partition
operator in Figure 3.13 ensures that at any time only one control variable of the xor-combinator sub-events can be true. An additional guard (grd_xor) will be added to both B1 and B2 to check the execution of the other. In Figure 3.13, we also show B1 which checks that B2 has not occurred (B2 = FALSE), the same also applies to B1.
In the multiple instances case (Figure 3.14), the partition operator used in inv_xor ensures that the sets \( B_1 \) and \( B_2 \) are disjoint. The grd_xor is added to \( B_1 \) to ensure the mutual exclusiveness property by checking that \( B_2 \) has not occurred for the same instance value, the same also applies to \( B_2 \).

### 3.4.1.4 Loop Combinator

The loop-combinator when applied to an event, means zero or more executions of that event. A loop-combinator should only be applied to one event. In case we need to apply it to more than one event, an extra level of refinement is added and a resetting event will be generated. The resetting event is required because a control variable will be generated for the loop sub-events to control their ordering, hence the need to reset the loop sub-events control variables to enable more than one execution. A loop is represented using the star symbol as shown in figures 3.15 and 3.16.

Unlike other events in ERS, the loop event does not have a corresponding control variable in its Event-B representation. In Figure 3.15, we show the effect of applying this combinator on the loop event, \( B \), and on the event following this combinator, \( C \). \( B \) can execute zero or more times, that is why, there is no need for a control variable to record its execution. The execution of \( B \) is terminated by the execution of its follow-on event \((C)\), this is ensured by the additional guard \((C = \text{FALSE})\), which is added to \( B \).
Because the execution of \( B \) is optional, the sequencing of \( C \) depends on \( A \) and not \( B \) (inv\_C\_seq).

Similarly in the multiple instances case (Figure 3.16), the execution of \( B \) for an instance value is disabled if \( C \) executes for the same instance value (\( p \notin C \)). We can also see that
the set $C$ is subset of $A$ because of the possible zero execution case of $B$.

### 3.4.1.5 All Combinator

The *all-combinator* is used to represent execution of an event for all values of a parameter from some set. This combinator is a generalisation of the *and-combinator*, so the event following this combinator cannot be executed, until the *all-combinator* event completes execution for all the instance values of the parameter set. Figures 3.17 and 3.18 show an example of applying this combinator with $p_2$ as the *all-combinator* parameter, in the single and multiple instances case respectively. If we need to apply this combinator to a sequence of events, an extra level of refinement is required.

In the single instance case (Figure 3.17), the type of the *all-combinator* event ($B$) is the same as that of the *all-combinator* parameter ($B \subseteq \text{Type}(p_2)$). The sequencing invariant of $B$ means that if $B$ has executed for at least one instance value ($B \neq \phi$), then $A$ must have occurred. The sequencing invariant of $C$ indicates that $C$ can only occur after $B$ completes execution for all instance values of the set of $p_2$. These sequencing invariants are maintained by the sequencing guards added to the events ($B$ and $C$).

In the multiple instances case (Figure 3.18), the type of $B$ is extended with the type of $p_2$, which is the *all-combinator* parameter ($B \subseteq \text{Type}(p) \times \text{Type}(p_2)$). The sequencing invariant of $C$ ensures that for any instance value of parameter $p$ in the set $\text{Type}(p)$, $B$ has executed for all instances of $p_2$ in the set $\text{Type}(p_2)$.
Figure 3.17: ERS *all-combinator* (Single Instance Case)

Figure 3.18: ERS *all-combinator* (Multiple Instances Case)
### 3.4.1.6 Some Combinator

The *some-combinator* is a generalisation of the *or-combinator*. Similar to the *all-combinator*, it includes the addition of a parameter (Figures 3.19 and 3.20). As an *or-combinator* generalisation, it means that the *some-combinator* sub-event must be executed for at least one instance of the *some-combinator* parameter before the next events are enabled. However, the execution of the events following the *some-combinator* does not mean the *some-combinator* sub-events have to stop execution.

![Diagram](image)

**Figure 3.19: ERS *some-combinator* (Single Instance Case)**

Figure 3.19 shows the effect of applying the *some-combinator* to the Event-B translation of both $B$ and $C$ in the single instance case. Compared to the *all-combinator*, the difference is in the sequencing guard and invariant of $C$, which requires only the execution of $B$ for one instance of the *some-combinator* parameter $p2$ to be enabled.

Similarly in the multiple instances case (Figure 3.20), the sequencing invariant of $C$ ($C \subseteq \text{dom}(B)$) ensures that $B$ has executed for at least one instance value of $p2$.

### 3.4.1.7 One Combinator

The *one-combinator* is a generalisation of the *xor-combinator*, so it requires the execution of its sub-event for exactly one instance of its parameter. Similar to the other replicators *all* and *some*, it requires the addition of a parameter (Figures 3.21 and 3.22).
Figure 3.20: ERS *some-combinator* (Multiple Instances Case)

Figure 3.21: ERS *one-combinator* (Single Instance Case)
The sequencing of \( B \) and \( C \) is similar to the \textit{some-combinator}, but the main effect of the \textit{one-combinator} appears in its sub-event \((B)\). Figure 3.21 shows the result of applying the \textit{one-combinator} in a single instance case, which is illustrated by the one invariant and guard. The \textit{one} invariant \((\text{card}(B) \leq 1)\) ensures that \( B \) can at most occur once. The guard \( B = \phi \) checks that the \( B \) has not executed before.

![Diagram of Abstract Event and Invariants](image)

**Figure 3.21**: ERS \textit{one-combinator} (Single Instance Case)

Similarly in the multiple instance case (Figure 3.22), the \textit{one} invariant ensures that \( B \) can occur at most once for any instance value \((p_2)\). This is maintained by the guard \((p \notin \text{dom}(B))\) which is added to \( B \).

### 3.4.2 Event Refinement Structures (ERS) Constraints

ERS has some restrictions that are required for refinement to work smoothly and to simplify the modelling process. One of the significant restrictions is the \textit{single solid line rule}, where exactly one leaf can be connected to its parent event by a solid line. The \textit{single solid line rule} can be justified by the event execution trace of the model, where hiding newly added events from the event execution trace of the concrete model, should result in the same event execution trace of the abstract model (Butler, 2009b,a).

Consider the ERS diagram in Figure 3.23, which represents a generic pattern for introducing a replicator in a refinement step. The event execution trace at the abstract
level is \(< A >\), while the event execution trace at the concrete level allows multiple executions of \(B\) for different instances of the parameter \(p\). One possible execution is \(< B(p_1), B(p_2), B(p_3), C >\), where \(p_1, p_2\) and \(p_3\) represent different instance values of the replicator parameter \(p\). Hiding the newly added events will result in the event execution trace \(< C >\), which is the same as the abstract level, since \(C\) refines \(A\). However, if we violate the single solid line rule and allow \(B(p)\) to refine \(A\), we will have multiple executions of events that refine \(A\), represented by \(< B(p_1), B(p_2), B(p_3) >\), which does not correspond to the abstract trace which only allows a single execution of event \(A\).

![Figure 3.23: General Pattern for Introducing Replicator](image)

The rest of the restrictions can be summarised as follows (Salehi Fathabadi et al., 2012):

- As a consequence of the single solid line rule, where exactly a single event can refine an abstract event, the combinators and, or, loop, all, some and their directly connected children are always connected to the previous level using dashed lines.

- In ERS the only case where more than one event can be connected to its parent using solid line is when applying the xor-combinator, since the xor-combinator ensures the execution of exactly one of its sub-flows.

- The only replicator which can be refining (i.e., connected to its parent by a solid line) is the one-combinator, because it allows the execution of exactly one instance value.

- The events which are directly connected to a combinator, will inherit their refining property from their combinator.

- A combinator cannot be applied to another combinator at the same refinement level, i.e., a combinator can only be connected to an event. That event can be elaborated in later refinements.

- Replicators (Figure 3.23) can have only one child event. If more than one event needs to be replicated, this has to be done through refinement.

- Similar to replicators, a loop can only be connected to one child event.

- A loop cannot be the last child of a flow.

- Multiple instance modelling can be introduced by either adding a parameter to the root of an ERS diagram and/or by using a replicator.
• The set of values of a parameter added to a replicator can only be a static set, i.e., a constant or a carrier set.

• When a parameter is applied for multiple instance modelling, it is inherited by all the children of a tree or sub-tree.

• The abstract level is distinguished from the other refinements by having events with only dashed lines, and its root is not an event, it is the name of the ERS flow.

• The same event name cannot be used in different branches of the tree. The only case the same event name can be used is the case of a parent and its solid child leaf.

3.5 Conclusion

In this chapter, we have introduced the main concepts required to understand our work. Our work focuses on the formal method Event-B and the diagrammatic notation ERS which extends Event-B with a visual representation. We choose Event-B for its mathematical simplicity and important concepts in facilitating the formal modelling using refinement. ERS, on the other hand provides a visual side of Event-B supporting control-flow as well as refinement. The combination of Event-B with the ERS approach can result in a formal workflow modelling approach facilitating formal modelling on one hand, and formalising workflow modelling on the other hand.
Chapter 4

Extending ERS by Dynamic Workflow Modelling

4.1 Introduction

The ERS approach provides Event-B with explicit control flow using different control flow combinators. The ERS combinators not only have simple logical combinators (and, or, xor), but also supports behaviour replication through quantified combinators (all, some, one).

Despite the variety of ERS combinators, ERS still lacks the flexibility to model dynamic control flow that supports dynamic changes in the degree of concurrency. Specifically in the cases where the degree of parallelism is data dependent and data values can change during execution. In this chapter we start by identifying some of the limitations of the ERS approach, and we also extend the original ERS approach to address the identified limitations.

The work presented in this chapter and Chapter 5 is submitted to “Software and Systems Modeling (SoSyM)” journal.

4.2 Limitations of the Original ERS Approach in Control Flow Modelling

4.2.1 Data Dependent Workflow Limitations

One of the main objectives of ERS is the explicit representation of control flow in Event-B. While ERS has a variety of combinators to support control flow, ERS still lacks flexibility to support dynamic changes in the degree of concurrency.
To better illustrate this limitation, consider the example ERS diagram in Figure 4.1. The ERS diagram describes part of the flow of an emergency dispatch system, which is responsible for sending appropriate resources to emergency incidents. The root of the tree represents the name of the workflow model. The parameter $i$, introduced at the root of the tree and inherited by all the nodes and leaves of the tree, indicates that there are multiple instances of the Emergency_Dispatch workflow with each instance being distinguished by parameter $i$. The leaves of the tree represents events in Event-B, where a leaf is an atomic or non-decomposed task. The leaf events of different instances of a workflow are interleaved.

![ERS Diagram of a Fire Dispatch Workflow](image)

As explained earlier in Chapter 3 the order of execution of the ERS leaf events is from left to right. In Figure 4.1, the event Set_Action_Plan sets an action plan $ap$ for the incident, identifying the resources needed to handle an incident in the Incident_Management activity. The Incident_Management activity is in turn decomposed into Handle_Incident where we apply the all combinator, the generalisation of and. Handle_Incident can interleave for different instances $r$ in the set $ap(i)$, i.e., the action plan of incident $i$. The expression $ap(i)$ represents the set of resources required for incident $i$ and parameter $r$ of ranges over this set.

The all$(r : ap(i))$ structure is an example of a data dependent workflow. The set $ap(i)$ represents the action plan of the incident $i$, and the value of $ap(i)$ is determined by the event Set_Action_Plan. Now the challenge is that in a real system the action plan of an incident can change during the execution of this data dependent workflow. This is not allowed in the original ERS. For this reason we extend the ERS approach with a new flexible combinator that allows dynamic modelling of data dependent workflows. We also improve the parameter definition to allow the parameter set to be any expression and not only constants as the original ERS did.

### 4.2.2 Loop Limitations

The compositional structure of ERS means that the loop is a structured loop, where there is only one entry point and one exit point. Originally ERS required the loop to
have only one child node, to be executed zero or more times. To give the loop more
flexibility without breaking its structured nature, we generalise the loop to have one
or more child nodes. Such generalisation supports dynamic behaviour through non-
deterministic execution of event at each iteration. For example, the itinerary selection
of a travel agency booking system allows the user to book different flights, hotels and/or
cars at the same time, this can be represented using a loop, where at each iteration the
user can perform a different selection (flight, hotel or car), but all selections still have
the same exit point, which is when the user decides to check out.

Using the original form of the loop will require refinement and additional combinators
like the and-combinator. For example to represent the non-deterministic iterations of
the events $A$ and $B$, we cannot simply do as shown in Figure 4.2. The first diagram is
invalid because it implies some sequencing between the loops $A$ and $B$. The second case
is also invalid because it applies two consecutive combinators at the same level.

![Figure 4.2: Invalid Representation of Non-Deterministic Loop Events](image)

The correct representation is in Figure 4.3, which can be only done by introducing
refinement. There are two possible ways either using the or-combinator or using the
and-combinator. The first disadvantage is the inconsistent representation of such a
simple behaviour. The second disadvantage is that, the refinements will result in the
addition of unnecessary events, event $C$ in the or-refinement case, and events $A2$ and
$B2$ in the and-refinement case. The simple solution is to generalise the loop definition
to allow its application to more than one event, as shown in Figure 4.4.

4.3 Advantages of ERS in Control Flow Modelling

The restrictions described in Section 4.2 should not undermine the importance of ERS
in modelling workflows. In ERS the main characteristics of control flow modelling are
supported through different combinators described in Chapter 3. These combinators
support the typical aspects of control flow modelling which are: sequencing (left to right
placement of nodes), choice (xor, or), concurrency (and, all, some), repetition (loop,
all, some) and synchronisation (and, all). We have seen in Figure 4.1 an example of
using ERS in modelling sequencing, concurrency, repetition and synchronisation. Not
only does ERS support control flow, but its translation to Event-B allows the handling of data, e.g., updating variables through event actions. Supporting control flow and handling data using Event-B, in addition to the graphical nature of ERS and its Event-B formal semantics, which allows validation and verification, makes ERS well suited for workflow modelling.

In order to better understand the different ERS combinators and how they can be used to control the flow of events, we will summarise the disabling conditions of the events under a combinator, i.e., the conditions that stop the flow of events from executing again, and the enabling conditions of their successor events according to the combinator applied.

Consider Figure 4.5, where $P$ and $Q$ are themselves ERS diagrams. $P$ and $Q$ are represented using the ERS diagram in a triangle to indicate the possibility of being a single event or a sub-flow. $P$ occurs to the left of $Q$, and the combinator $C$ applied to $P$ defines the disabling conditions for $P$ and the enabling conditions for $Q$.

Table 4.1 describes the disabling conditions of $P$ and the enabling conditions for $Q$ according to the different forms of combinator $C$. The enabling condition for $P$ is determined by treating $P$ as $Q$ and whatever precedes $P$ as $P$. If nothing precedes $P$ then no additional enabling condition is required. Note that the conditions for disabling $P$ and
enabling $Q$ are not necessarily the same. Moreover, the events within $P$ may have other explicit data guards which might cause $P$ to be disabled. The conditions in the middle and last columns define conditions (represented as guards) that are conjoined with any explicit event guards.

Table 4.1: Disabling and Enabling Conditions for ERS Patterns

<table>
<thead>
<tr>
<th>Form of C</th>
<th>Disable $P$</th>
<th>Enable $Q$</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>once</strong> $P$</td>
<td>$P$ has executed</td>
<td>$P$ has executed</td>
</tr>
<tr>
<td><strong>loop</strong> $P$</td>
<td>$Q$ has made a step</td>
<td>True</td>
</tr>
<tr>
<td><strong>P1 and P2</strong></td>
<td>Both $P1$ and $P2$ have executed</td>
<td>Both $P1$ and $P2$ have executed</td>
</tr>
<tr>
<td><strong>all</strong> $p2$: $S$. $P(p2)$</td>
<td>$P(p2)$ has executed for each $p2$ in $S$</td>
<td>$P(p2)$ has executed for each $p2$ in $S$</td>
</tr>
<tr>
<td><strong>P1 or P2</strong></td>
<td>Both $P1$ and $P2$ have executed</td>
<td>$P1$ or $P2$ has executed</td>
</tr>
<tr>
<td><strong>some</strong> $p2$: $S$. $P(p2)$</td>
<td>$P(p2)$ has executed for each $p2$ in $S$</td>
<td>$P(p2)$ has executed for some $p2$ in $S$</td>
</tr>
<tr>
<td><strong>P1 xor P2</strong></td>
<td>$P1$ or $P2$ has executed</td>
<td>$P1$ or $P2$ has executed</td>
</tr>
<tr>
<td><strong>one</strong> $p2$: $S$. $P(p2)$</td>
<td>$P(p2)$ has executed for one $p2$ in $S$</td>
<td>$P(p2)$ has executed for some $p2$ in $S$</td>
</tr>
</tbody>
</table>

The first form describes the case where $P$ is a single execution flow (indicated diagrammatically by a line with no oval). Then one execution of $P$ will disable further executions of $P$ and enable $Q$. The loop pattern will disable $P$ once $Q$ starts execution, but there are no conditions on enabling $Q$, which means it is possible there will be zero executions of $P$. 
The *and-combinator* requires both sub-flows $P_1$ and $P_2$ to execute before $P$ is disabled and $Q$ is enabled. In the generalised form of the *and-combinator*, the *all-replicator* requires the sub-flow to be executed once for each $p_2 \in S$ before $P$ is disabled and $Q$ is enabled.

The *or-combinator* requires at least one sub-flow of $P$ to execute before enabling $Q$, and even if $Q$ has executed, $P$ can continue its execution until all its sub-flows have executed, in this case both $P_1$ and $P_2$ should execute to disable $P$. The *some-replicator* requires the sub-flow $P$ to be executed for at least one $p_2 \in S$ to enable $Q$. The sub-flow $P$ will be disabled when all $p_2 \in S$ have executed.

Similar to the *or-combinator*, the *xor-combinator* requires at least one sub-flow, $P_1$ or $P_2$, to execute before $Q$ is enabled. However unlike the *or-combinator*, the *xor-combinator* sub-flow will be disabled once one of its sub-flows ($P_1$ or $P_2$) executes. The *one-replicator*, which is the generalisation of the *xor-combinator* will enable $Q$, when the sub-flow $P$ executes for at least one $p_2 \in S$, while $P$ will be disabled when exactly one $p_2 \in S$ executes.

As we can see in Table 4.1, ERS not only supports simple combinators like (*or*, *and*, *xor*), but also explicitly supports behaviour replication through different combinators (*all*, *some*, *one*). The multiple instances support is one of the ERS key strengths, that is why in the following sections, we extend ERS to make better use of it in workflow modelling.

### 4.4 Parameter Extension

In the original translation of ERS diagrams to Event-B, the parameter set of a replicator is always described as a constant defined in the context of the model (Wiki.event-b.org, 2012a; Fathabadi et al., 2014). However, it is not always possible to define the parameter set in the context, as we have seen in Figure 4.1, the parameter set of $r$ is determined by an event (*Set_Action_Plan*) in the machine.

This requires extending the definition of the parameter and adding some more guards and invariants when applying a replicator to enable the use of variables and not only constants. We start by extending the parameter definition by adding a `range_expression` which can be a variable or a constant. The definition of a parameter will be extended as follows:

```
par = “par”(name, Type, range_expression)
```

We will explain this through a simple example represented by the ERS diagram of Figure 4.6, where each doctor is allocated a set of patients to visit at home. The patients
that the doctor needs to visit will differ from one day to the other. The patients will be allocated to each doctor \((d)\) in the event \(Allocate\_Patients\).

In this case the parameter \(\text{par}\) of the \(all\)-combinator is: \(\text{par} = \text{"par"}(p, PATIENTS, patients[{d}])\), where the parameter name is \(p\), the type is \(PATIENTS\) and the range expression is the variable \(patients[{d}]\), which is the set of patients allocated to doctor \(d\).

![Home Care System Example](image)

Using a variable instead of a constant requires the event \(Visit\_Patient\) to explicitly check that it is executing for an instance in the variable set. Therefore, we need to explicitly add the guard \(p \in patients[{d}]\) to the event \(Visit\_Patient\). This guard can be enforced by an invariant to ensure that \(Visit\_Patient\) can only execute for values in the range expression set \((patients[{d}])\) as follows:

\[
\forall d. d \in DOCTOR \Rightarrow Visit\_Patient[{d}] \subseteq patients[{d}]
\]

If the range expression guard \((p \in patients[{d}])\) is removed from the event \((Visit\_Patient)\), then in this case the doctor can visit any patient, whether allocated to them or not. Moreover, the invariant enforcing the guard, will also ensure that patients cannot be removed from the allocated patients of the doctor after visiting them.

### 4.5 A Dynamic Replicator: Par-Replicator

**Par-Replicator Behaviour:** The purpose of the \(par\)-replicator is to execute a sub-flow for several instances of a certain parameter, but the difference to other ERS replicators \((all, some, one)\) is that the number of instances of that parameter is not predetermined and can change during the execution of its flow, depending on the environment.

The behaviour of the \(par\)-replicator can be considered as a combination of the \(loop\) and the \(some\)-replicator. It is similar to a \(loop\) in the sense that it can execute zero or more times and it stops execution when the follow-on event executes. It is similar to the \(some\)-replicator, since it does not have to be applied to all the instances of the parameter set, and its sub-flow may be interleaving for different instances. However, the main difference with the \(some\)-replicator is that the \(some\)-replicator sub-flow can
continue execution even after the execution of its follow-on event and the parameter instances cannot change during the execution of the some-replicator sub-flow.

**Diagrammatic Representation:** Similar to the ERS replicators, the par-replicator is represented within an oval with the par keyword followed by the name of the parameter, as shown in Figure 4.7. The parameter p2 of the par-replicator has p_Set as the range_expression. The non-connected lines in the ERS diagram, means we are only showing part of the flow. The non-refining restriction which is the result of the single solid line rule, explained in Chapter 3, applies to the par-replicator, i.e. the par-replicator sub-event cannot be a direct refinement to an event at the previous abstract level and will always be connected to the previous level using a dashed line. The non-refining restriction is due to the fact that the par-replicator sub-event can execute zero or more times, which violates the single solid line rule of ERS, where exactly one event can be a direct refinement of the parent event.

![Figure 4.7: Diagrammatic Representation of the par-replicator](image)

**Parallel Producer-Consumer Pattern:** The par-replicator supports dynamic behaviour, where the number of replications cannot be predetermined. Take the example of a travel agency, where the travel agency attempts to book flights, hotels etc., while the user is still selecting the itinerary. In this case the number of instances to be booked changes dynamically as the user extends their itinerary. This example can be generalised to any parallel Producer-Consumer pattern, as shown in Figure 4.8. The parallel Producer-Consumer pattern shows how elements are consumed, while other elements are still being produced. In this case the and-combinator indicates the concurrent production and consumption of elements, where the left hand side sub-flow of the and-combinator, shows how new elements are produced and added to the set a in the Produce event, while the right hand sub-flow of the and-combinator represents the elements consumption in the Consume event.

In both producing and consuming elements we use the par-replicator. When producing elements, we do not know how many elements are needed, and disabling element production is determined by the End Produce event, that is why the par-replicator was a good candidate to represent this behaviour. The Consume event can start consuming elements from the set a, while other elements are still being produced, hence the need for the par-replicator. This pattern cannot be supported by the previous ERS replicator.
combinators (Salehi Fathabadi et al., 2012), because all existing replictors do not allow adding new elements while executing.

![Figure 4.8: Parallel Producer-Consumer Pattern](image)

### 4.5.1 Par-Replicator Semantics

The *par-replicator* has the same disabling and enabling conditions as the *loop* in Table 4.1. We can add the *par-replicator* to Table 4.1, as shown in Table 4.2. The *par-replicator* sub-flow (*P*) will be disabled when *Q* starts execution. The enabling of *Q* is independent of the *par-replicator*, that is why it is True.

<table>
<thead>
<tr>
<th>Form of C</th>
<th>Disable P</th>
<th>Enable Q</th>
</tr>
</thead>
<tbody>
<tr>
<td>par ( p2:SP(p2) )</td>
<td>( \text{par}(p2):S ) ( P(p2) )</td>
<td>Q has made a step</td>
</tr>
</tbody>
</table>

Figures 4.9 and 4.10, show the Event-B encoding of the *par-replicator* in the single instance case and the multiple instances case respectively. In both cases the atomicity of the *Abstract Event* is decomposed into the events *A*, followed by *B* where we apply the *par-replicator*, and finally event *C* which is the refining event. In addition to the normal sequencing, typing and gluing invariants and guards, the *par-replicator* results in an additional guard \( \text{grd}_{\text{par}} \) added to the *par-replicator* event (*B*). This guard disables the execution of the *par-replicator* event (*B*), if the follow-on *C* event has executed. In the single instance case the *par-replicator* disabling guard is \( C = \text{FALSE} \), while in the multiple instances case, it is \( p \notin C \) checking that *C* has not executed for a particular instance of the parameter set.

The *par-replicator* is similar to any ERS *replicator*, where its child type is extended by the parameter type (*Type(p2)*) as shown by \( \text{inv}_B.\text{type} \) in both the single instance and multiple instances cases. The invariant \( \text{inv}_B.\text{rep} \) and the guard \( \text{grd}_{\text{exp}} \) are the result of
extending the parameter with range_expression, as explained in Section 4.4, which apply to any replicator (all, some, one, par). The range_expression invariant and guard will ensure that the values of p2 which are added to B are only from the range_expression.

There is also a possibility that the par-replicator event does not execute, which is why we can see the sequencing guard (grd_seq) and invariant (inv_C_seq) of the follow-on event C is similar to B.

We explained earlier that the set of the par-replicator parameter can change during execution. However, this is restricted to “non-decreasing” change. Otherwise this will violate the range_expression invariant (inv_B_rep) in figures 4.9 and 4.10, which requires that the par-replicator event (B) to execute only for values in the range_expression (p_set).
Chapter 4 Extending ERS by Dynamic Workflow Modelling

53

Figure 4.10: Multiple Instances Semantics of the par-replicator

Refinement: When a par-replicator child is refined into some sequence of events, we need to ensure that the follow-on event, C, which can disable their execution, does not interrupt the execution of their sub-events in the middle of their execution. This is because when decomposing the par-replicator event into some sequence of sub-events, we consider the activity of the par-replicator event is represented by the execution of all the sub-events and not only the refining event. This can be done by adding a guard to the follow-on event (C), that prevents it from executing before the sub-flows of the par-replicator complete their execution. This constraint on the follow-on event C is also represented by an invariant.

Consider the scheme in Figure 4.11 where the par-replicator child, B, is refined into the sequence of events starting with B1 and ending with Bn. The disabling guard of
the *par-replicator* event is added to the first child of the refined *par-replicator* \((B1)\), disabling the execution of the *par-replicator* sub-flow when the follow-on event, \(C\), executes. The invariant \((inv_{C,par})\) and guard \((grd_{par})\) of the *par-replicator* follow-on event \((C)\) ensure that \(C\) does not interrupt the *par-replicator* sub-flow in the middle of its execution. This is done by checking that if the first child of the *par-replicator* flow has executed, then the last child of the *par-replicator* flow must have executed, before the follow-on event can disable the *par-replicator* sub-flow. In Figure 4.11, \(inv_{C,par}\) and \(grd_{par}\) of \(C\) check the equality of \(B1\) and \(Bn\) ensuring that \(B1\) and \(Bn\) executed for the same instance values, before \(C\) can be executed.

![Diagram](image)

**Figure 4.11: Refinement of the *par-replicator* Event**

In the case of multiple instances, the same guard \(B1 = Bn\) will be added to \(C\) if both events have the same parameter sets. In both single instance and multiple instances cases, if the parameter sets are not the same for the first and last events \((B1\) and \(Bn)\) as a result of applying a replicator, we need to project out the non-common parameters by applying the *dom* operator. For example if \(B1\) has an additional parameter \(p3\) as a result of applying a replicator, the *par-replicator* guard added to \(C\), will be \(dom(B1) = Bn\). In the case a logical combinator is applied to the first and/or the last child of the refined *par-replicator*, we apply union and intersection of sets as needed.
Note that if the *par-replicator* is refined into a single event (i.e., \( n \leq 1 \)) or a refining *xor* or *one* flows only, the guard will not be added to \( C \), as the execution cannot be interrupted.

There is also another possibility that we only block the execution of the follow-on event \( C \), until the refining event has occurred. In this case the guard that will be added to \( C \) in Figure 4.11 will be \( B_1 = Br \), where \( Br \) as shown in Figure 4.12 is the refining event of \( B \) and \( 1 \leq r \leq n \), i.e., if \( B_1 \) has occurred \( C \) will be blocked until \( Br \) has occurred. However, we have decided to adopt the approach that blocks the *par-replicator* follow-on event until the completion of the *par-replicator* flow, this is because we do not have enough cases to demonstrate the necessity of the partial block of the follow-on event.

![Diagram](image_url)

**Figure 4.12: The *par-replicator* Refining Event**

### 4.6 Non-Deterministic Loop

The compositional structure of ERS means that the *loop* is a structured loop, where there is only one entry point and one exit point. Originally ERS requires the *loop* to have only one child flow, to be executed zero or more times (Salehi Fathabadi et al., 2012). To give the loop more flexibility, we generalise the *loop* to have one or more child flows, with one sub-flow being chosen non-deterministically for execution in each iteration. The disabling and enabling conditions of \( P \) and \( Q \) of Figure 4.5 of the generalised form of the *loop* are still the same as the original *loop* shown in Table 4.1, as all the *loop* events will be disabled by the execution of the follow-on flow.

**Diagrammatic Representation:** A *loop* is represented by an asterisk within an oval, as shown in Figure 4.13. Similar to Figure 4.7, the non-connected lines indicate that we are only showing part of the flow. A *loop* can have one or more child events. Similar to the *par-replicator*, the *loop* events cannot be the last events of a flow, and they are always connected to their parent by a dashed line. The previous form of the *loop* is a special case of the general *loop* where the *loop* has only one child.
4.6.1 Non-Deterministic Loop Semantics

Figures 4.14 and 4.15 represent the Event-B encoding of the generalised form of the loop in the single instance case and the multiple instances cases respectively. The loop sub-events ($B_i$) do not require control variables and no action is required to record their execution. This is because of the zero execution case of the loop, which is also reflected by the sequencing invariant ($inv_{C Seq}$) and guard of the follow-on event $C$, which is similar to the loop sub-events depends on the execution of event $A$.

The effect of applying a loop is the addition of the $grd$ loop to all the loop sub-events ($B_i$), which checks the execution of the follow-on event. In the single instance case the
loop disabling guard is \((C = \text{FALSE})\), and in the multiple instances case it is \((p \notin C)\).

![Diagram of Abstract Event](image)

**Invariants**
- \(@\text{inv\_A\_typeA} \subseteq \text{Type}(p)\)
- \(@\text{inv\_C\_seqC} \subseteq A\)
- \(@\text{inv\_C\_gluing C} = \text{Abstract\_Event}\)

**Event A**
```plaintext
any p
where
  @grd_self p \notin A
then
  @act A = A \cup \{p\}
end
```

**Event C refines AbstractEvent**
```plaintext
any p
where
  @grd_self p \notin C
  @grd_seq p \in A
then
  @act C = C \cup \{p\}
end
```

**Event Bi**
```plaintext
// 1 \leq i \leq n
where
  @grd_seq p \in A
  @grd_loop p \notin C
end
```

**Figure 4.15: Multiple Instances Semantics of the Loop**

**Refinement:** If any of the **loop** sub-events is refined, the children of the refined sub-event will require control variables to ensure their proper execution order (Figure 4.16). These control variables need to be reset after their execution in order to enable more than one execution of the **loop**. Resetting the control variables is done in a separate event \((\text{reset\_loop\_B}_i)\) for each loop refined sub-event, after the execution of its last leaf event \((B^n_i)\), as shown by the sequencing guard \((B^n_i = \text{TRUE})\). The control variables of the refined **loop** will be reset by setting them to \text{FALSE} in the single instance case (Figure 4.16), while in the multiple instances case, the control variables are reset by removing the parameter instance from the control variable set \((B^n_i = B^i_i \setminus \{p\})\). As explained in Wiki.event-b.org (2012b), the resetting of control variables is not done in the last event of the loop child \((B^n_i)\), in order to avoid complicated cases like the case when the last child is an **and-combinator** child.

Similar to the **par-replicator**, in order to avoid interrupting the **loop** refined child events in the middle of their execution, for each refined **loop** sub-event, we add a guard \((\text{grd\_loop})\) and an invariant \((\text{inv\_C\_loop})\) to the **loop** follow-on event \((C)\) checking that the first leaf
event of the refined loop child has not executed. The loop guard in the follow-on event is ensured by the resetting of the control variables.

4.7 Conclusion

In this chapter we have extended the ERS approach to address the limitations presented in Section 4.2. Our extended ERS approach has given ERS the flexibility to model the dynamic behaviour of the Producer-Consumer pattern. The newly introduced replicator, the par-replicator can be used to model data dependent workflows where data is not predetermined and can change during the workflow execution. Such replication behaviour without a priori knowledge is not supported by many important workflow modelling approaches like BPMN and activity diagrams (Russell et al., 2006a).
Our parameter extension with the range expression is a natural extension of the parameter sets, as data cannot always be known statically in contexts, but in lots of cases other dynamic factors can determine the parameter sets, as we have seen in different examples where the parameter set are data dependent and can be determined by other executing events.

The extended ERS approach still does not support arbitrary loops, where the loops can have different entry and/or exit points like BPMN and activity diagrams can do with their flexible notations. Such arbitrary loops cannot be supported by ERS due to its compositional structure, however this does not mean we cannot make the loops more flexible than their original definition. Our generalised form of the loop supports non-deterministic execution of the loop events that have the same entry and exit points.

In this chapter we have also summarised the enabling and disabling conditions of the different control flow combinators of ERS, making it easier for the modeller to understand and select the required combinator. The ERS approach with its new dynamic extensions are applied to a case study, which will be presented in Chapter 5.
Chapter 5

Fire Dispatch Case Study

5.1 Introduction

In this chapter, we present the fire dispatch case study, which is an improved version of the case study presented in Dghaym et al. (2013) in order to support dynamic changes in the fire dispatch workflow. We model the fire dispatch workflow using the extended ERS approach, presented in Chapter 4, and Event-B. The complete version of the model is presented in Appendix A.

The aim of this case study is first to validate our ERS extensions by applying them to a complex workflow like the fire dispatch workflow, and second to present our workflow modelling approach, which uses ERS to model control flow and Event-B to model data flow.

The combination of ERS and Event-B has reduced the complexity of this case study, with ERS supporting hierarchical decomposition and multiplicity in flows and sub-flows using the replication combinators, in addition to the simple mathematical language of Event-B, which made it easier to represent the different data structures, their relations and formulate and prove the consistency properties of the fire dispatch workflow.

5.2 Fire Dispatch System Overview

The main goal of the fire and rescue service is the public safety as well as the safety of the fire fighters. To achieve this goal, the fire service works on responding to fire or emergency incidents as fast as possible and in an expedient manner. This requires a 24/7 reliable dispatch system, as well as a robust communication system.
As described by Hantsfire.gov.uk (2000), response to emergency calls starts from the control room. The control operator workstation consists of 3 systems as shown in Figure 5.1.

- Integrated Communications Control System (ICCS): Which allows all telephone and radio communications.

- The Command and Control System (C and C): Which is the main focus of our case study, this system is responsible for handling all the incident information, finding the nearest available resources for the incident, alerting the stations and fire fighters and monitoring the status of the appliances, so basically this system is the centre of the whole dispatch operation.

- The Geographical Information System (GIS): This has 6 different mapping views of the covered areas showing the stations, incidents and resources locations.

These 3 systems can communicate and link together to facilitate the role of a control operator, but the way these systems communicate together is not studied in this case study. We focus on the dispatch system and how resources are allocated to incidents and the change of status of both resources and incidents from the start of an incident call until its closure.

5.3 Fire Dispatch System Requirements

In this section, we present the main functional requirements of the dispatch system that we are going to model. We group these requirements according to the problem they are going to solve into the following categories:

- **Incident Information:** The control operator tries to get information about the incident, which will allow the system to determine if it is a duplicate incident or not. These requirements are summarised in Table 5.1.

- **Incident Handling:** From the incident’s information, the system must identify a possible action plan to handle the incident. The action plan identifies the resources required to handle the incident’s type. Table 5.2 presents the incident handling requirements.
### Chapter 5 Fire Dispatch Case Study

#### Table 5.1: Incident Information Requirements

<table>
<thead>
<tr>
<th>REQ_ID</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>INF_1</td>
<td>The control operator must get information about the location and type of the incident</td>
</tr>
<tr>
<td>INF_2</td>
<td>Subsequent calls to an existing incident are referred to as duplicate calls</td>
</tr>
<tr>
<td>INF_3</td>
<td>A control operator can decide that an incident is duplicate at any point and there might be cases where an incident can be described as duplicate upon a resource arrival</td>
</tr>
<tr>
<td>INF_4</td>
<td>Each incident type is associated to a default priority level</td>
</tr>
<tr>
<td>INF_5</td>
<td>An operator can change the priority level of an incident</td>
</tr>
</tbody>
</table>

#### Table 5.2: Incident Handling Requirements

<table>
<thead>
<tr>
<th>REQ_ID</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HND_1</td>
<td>The system should identify the predetermined action plan (PDA) according to the location type and type of the incident</td>
</tr>
<tr>
<td>HND_2</td>
<td>The action plan must identify the resource types that need to attend the incident and their number</td>
</tr>
<tr>
<td>HND_3</td>
<td>The action plan can be changed by requesting more resources to attend the incident</td>
</tr>
<tr>
<td>HND_4</td>
<td>A stop message can be sent to the operator to stop sending resources to the incident when there is sufficient resources to deal with it, and this message should be sent to all resources that are not in attendance</td>
</tr>
<tr>
<td>HND_5</td>
<td>When proposing available resources to be allocated to the incident, this can include resources mobilised to lower priority incidents</td>
</tr>
</tbody>
</table>

#### Resource Manipulation:

The resource types to be allocated to an incident are identified by the action plan associated to that incident. The allocation of the resources to an incident is done by the operator, who chooses a resource from the system’s suggestions. The system suggests resources according to their type, availability and how quick they can attend an incident. Table 5.3 summarises the resource manipulation requirements.

#### Table 5.3: Resource Manipulation Requirements

<table>
<thead>
<tr>
<th>REQ_ID</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RES_1</td>
<td>If a resource is reallocated to a higher priority incident or becomes unavailable due to a break down for example, the system must alert the operator for a replacement resource</td>
</tr>
<tr>
<td>RES_2</td>
<td>Before a resource is in attendance, the system can keep looking for quicker resources that become available and suggest reallocating them to the operator</td>
</tr>
<tr>
<td>RES_3</td>
<td>A resource status can be available at station, mobilised to an incident, in attendance when it is at an incident location, or unavailable</td>
</tr>
</tbody>
</table>

#### Incident Closure:

In this workflow we restrict closing an incident to two cases, the case where an incident is handled by the fire service and the case where it is considered...
as a duplicate incident, which means it is handled by another incident call. Table 5.4 describes when an incident can be closed.

<table>
<thead>
<tr>
<th>REQ_ID</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLS_1</td>
<td>An incident must not be closed if any of the items in the action plan are not completed</td>
</tr>
<tr>
<td>CLS_2</td>
<td>An incident must not be closed if it still have allocated resources</td>
</tr>
<tr>
<td>CLS_3</td>
<td>An incident must not be closed if a stop message has not been sent</td>
</tr>
<tr>
<td>CLS_4</td>
<td>An incident should be closed if identified as a duplicate incident</td>
</tr>
</tbody>
</table>

Table 5.4: Incident Closure Requirements

5.4 Incremental Development of the Fire Dispatch Model

Using Refinement (Section 3.3.2) in modelling helps in reducing the complexity of the model and enhances our understanding of the problem by gradually adding more details to the model. Refinement is also supported by the hierarchical elaboration of the ERS diagrams. That is why when modelling the fire dispatch system we use refinement supported by the ERS diagrams.

The model of the fire dispatch system consists of five contexts to define the static part of the model, while the dynamic part of the model consists of an abstract and seven refinement levels as follows:

- **Abstract Level**: Models creation, information gathering and identifying the incident as duplicate or not.
- **First Refinement**: Introduces the closure of duplicate incidents and management of non-duplicate ones.
- **Second Refinement**: Introduces the ability to update the action plan during incident management.
- **Third Refinement**: Models the ability to update required resources during incident handling.
- **Fourth Refinement**: Introduces further incident details such as incident location and type.
- **Fifth Refinement**: Refines incident handling, showing how a resource can be allocated and deallocated from an incident.
- **Sixth Refinement**: Introduces physical resources, which are assigned to handle incidents and their relation with the required resources of an incident action plan.
- **Seventh Refinement**: Adds further details about the location and the duration of a physical resource to the incident.
Chapter 5 Fire Dispatch Case Study

5.4.1 Abstract and First Refinement: Duplicate and Non-Duplicate

When modelling the fire dispatch system using ERS diagrams, we apply multiple instances modelling to enable possibly simultaneous dispatch to different incidents. Multiple instances modelling is indicated by the addition of the parameter $i$, representing incidents, to the root of the diagram (Figure 6.11). The set INCIDENT is defined in the context, representing all potential incidents.

![ERS Diagram for the Fire Dispatch Abstract Level](image)

At the abstract level (Figure 6.11), we distinguish between two cases of incident handling which are duplicate and non-duplicate incidents as indicated by the application of the xor-combinator. The decision about the possibility of the incident being duplicate is determined after getting information (GatherInfo) about the incident during the incident call, which is initiated at the beginning by the CreateIncident event.

Having two different types of incidents (non-duplicate and duplicate) will result in two different paths when dealing with the incident, the normal path when the incident is not duplicate and the exceptional path when the incident is duplicate. This is shown in the refinement of the events Non-Duplicate and Duplicate in Figure 5.3.

![Refinement of Non-Duplicate and Duplicate Events](image)

In the case of non-duplicate incident (Figure 5.3(a)), an initial plan must be determined (SetInitialAP) to manage the incident (IncidentManagement), while in the case of a duplicate incident (Figure 5.3(b)), the incident will be simply closed (CloseDuplicate). Up to this stage we have not introduced any data variables or any additional constraints manually, and both machines of the abstract and first refinement are generated automatically from the ERS diagrams.
5.4.2 Second and Third Refinement: Incident Management

Once it is established that an incident is not duplicate, an initial action plan is established. The action plan is followed through during the incident management. However, in some cases the initial action plan can be updated during the incident management to request additional resources. Such behaviour is a parallel producer-consumer pattern, which was introduced in Chapter 4. In Figure 5.4 we show the refinement of the IncidentManagement event of the Non-Duplicate case (Figure 5.3(a)) using the parallel producer-consumer pattern. In this case the event RequestAdditionalRes will produce more resources to be added to the action plan, while HandleIncident will consume the resources in the action plan.

![Figure 5.4: Application of the Parallel Producer-Consumer Pattern for Incident Management](image)

In the third refinement, we extend the context with the carrier set $LR$, which represents the logical resources. In the fire dispatch case study, we make a distinction between logical resources ($LR$) and physical resources ($PR$). Logical resources represent the resources required by the action plan to handle the incident, e.g., an incident may have several different logical resources with the property class 3 fire engine. Physical resources will be introduced in Section 5.4.4, they are the actual resources that are going to be assigned to the incident, e.g., a specific fire engine. A physical resource, unlike a logical resource, can be only allocated to one incident. In our model we use logical resources for the action plan and physical resources for the resource allocation to the incident.

The action plan of the incident is represented by the data variable $ap$, which is added manually in the third refinement of the model and is defined using the following invariants:

\[
\text{inv}\_\text{ap\_type}: ap \in INCIDENT \leftrightarrow LR \\
\text{inv}\_\text{ap\_seq}: \text{dom}(ap) = \text{SetInitialAP}
\]
The first invariant \((inv\_ap\_type)\) defines the type of the data variable \(ap\), which is a relation between incidents and logical resources (LR). The variable \(ap\) is a relation because an incident can be associated to different logical resources and a logical resource can be associated to different incidents. In the second invariant, we are relating the data variable \(ap\) to the control variable \(SetInitialAP\), this is to ensure that setting the action plan \((ap)\) cannot start before the \(SetInitialAP\) event, and when this event occurs then the incident must have an action plan \((ap)\). This invariant \((inv\_ap\_seq)\) shows how we can combine data variables \((ap)\) with control variables \((SetInitialAP)\) to represent and enforce the requirements using invariants.

In Figure 5.5, we present the Event-B specification of \(SetInitialAP\) event after introducing the data variable \(ap\) in the third refinement.

In Figure 5.5, the numbered guards \((grd1, grd2)\) and action \((act1)\) are added manually, while the rest are generated from the ERS diagram. The data variable \(ap\) is initialised to the empty set, and then the \(SetInitialAP\) sets the action plan of the incident \(i\) for the first time in \(act1\) to the set of logical resources \(lrs\). At this point \(lrs\) is defined as any set of logical resources which is not empty \((grd1\) and \(grd2)\). The set \(lrs\) will be defined more precisely in Section 5.4.3.

Regarding the producer part of Figure 5.4, the Event-B specifications of its events are shown in Figure 5.6. After setting the initial action plan, it is possible to update the action plan \((ap)\), by requesting additional resources. This is done manually by updating \(ap\) in \(act1\) of \(RequestAdditionalRes\). The last guard \((grd1)\) of \(RequestAdditionalRes\) is added manually. The guard \(lr \notin ap\{i\}\) is to ensure that the newly added resource is a fresh resource, where \(ap\{i\}\) is the relational image of \(ap\) with respect to \(\{i\}\) i.e., \(ap\{i\}\) is the set of logical resources required for incident \(i\).
The reason for not combining the guard \( lr \notin \mathit{ap}\{i\} \) as part of the range expression of the \( \text{par-replicator} \), i.e., having the \( \text{grd}_{\text{Exp}} \) of the parameter set as \( lr \in (LR \setminus \mathit{ap}\{i\}) \), is the “non-decreasing” restriction of the \( \text{par-replicator} \) range expression.

Requesting additional resources is disabled by sending a stop message, which is sent by a system operator to indicate that there are sufficient resources to deal with the incident, in this case it is the follow-on event of the \( \text{par-replicator} \) (\text{SendStopMsg}). When a stop message is sent, then it means the incident is satisfied by whatever resources it already had and no more resources are required to attend, even if there are still more logical resources in the action plan that are not satisfied yet. That is why we also introduce the data variable \( \text{notRequired}_{\mathit{ap}} \) which is defined by the invariant: \( \text{notRequired}_{\mathit{ap}} \subseteq \mathit{ap} \).

This data variable will be updated with the logical resources of the action plan (\( \mathit{ap} \)) that are not needed any more to manage the incident, as shown by the manually added action (\( \text{act1} \)) of \text{SendStopMsg}, which could be also empty. The logical resources that can be added to \( \text{notRequired}_{\mathit{ap}} \), which are defined by \( \text{grd1} \) as any set of logical resources in the action plan of the incident, will be defined more precisely in Section 5.4.4.

Handling the incident will consume logical resources in the action plan of the incident (\( \mathit{ap}\{i\} \)). In Figure 5.7, we show the Event-B encodings of the consumer events (\text{HandleIncident} and \text{IncidentManaged}). All the guards and actions associated with the consumer events are generated automatically from the ERS diagram.

### 5.4.3 Fourth Refinement: Incident Details

In the fourth refinement level, we introduce more details about the incident by refining \text{GatherInfo} to introduce the type and the location of the incident, Figure 5.8. We
extend the context to include the carrier sets INCIDENT\_TYPE, LOCATION and LOCATION\_TYPE representing the possible types, locations and location types of the incident. We also introduce three data variables \( iType \), \( iLocation \) and \( iLocType \) to associate an incident with a specific type, location and location type. These data variables are defined as functions in the invariants \( \text{inv\_iType} \), \( \text{inv\_iLocation} \) and \( \text{inv\_ilocType} \) to ensure that an incident can have one type, one location and one location type. Their domains are equal to the control variable \( \text{SelectType} \) and \( \text{SelectLocation} \) to ensure that these events will set the type and the location of the incident respectively. This is done in the manually added actions \( \text{act1} \) and \( \text{act2} \) of both events \( \text{SelectType} \) and \( \text{SelectLocation} \), Figure 5.8.

\[
\begin{align*}
\text{inv\_iType}: & \quad iType \in \text{SelectType} \Rightarrow \text{INCIDENT\_TYPE} \\
\text{inv\_iLocation}: & \quad iLocation \in \text{SelectLocation} \Rightarrow \text{LOCATION} \\
\text{inv\_ilocType}: & \quad iLocType \in \text{SelectLocation} \Rightarrow \text{LOCATION\_TYPE}
\end{align*}
\]

The requirements state that the type and the location type of the incident will help determine the action plan of the incident. This is done by introducing the predetermined action plan \( PDA \) constant. \( PDA \) is defined in the context by axioms \( \text{axm\_PDA\_1} \) and \( \text{axm\_PDA\_2} \). The first axiom defines \( PDA \) as a total function between the incident type and location type on one hand and the power set of logical resources i.e., \( PDA(t, l) \) is the set of logical resources required for an incident of type \( t \) at location type \( l \). The second axiom is to ensure the different types and location types are not assigned to the
empty set.

\textbf{axm\_PDA\_1:} \( PDA \in (\text{INCIDENT\_TYPE} \times \text{LOCATION\_TYPE}) \rightarrow \mathbb{P}(LR) \)

\textbf{axm\_PDA\_2:} \( \forall it, loc \cdot \text{it} \in \text{INCIDENT\_TYPE} \land \text{loc} \in \text{LOCATION\_TYPE} \implies PDA(\text{it} \mapsto \text{loc}) \neq \phi \)

After introducing \( PDA \), the \textit{SetInitialAP} event will be refined to set the action plan \( (ap) \), according to the type and location type of the incident as can be seen from \( grd1 \), \( grd2 \), \( grd3 \) and \( act1 \) of Figure 5.9.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure5.9.pdf}
\caption{Refinement of the Initial Action Plan According to Type and Location}
\end{figure}

Another extension refinement related to the location of the incident is done to the \textit{IsDuplicate} event, where we manually add some guards to satisfy the newly added
invariant:
\[ \forall i \cdot i \in IsDuplicate \rightarrow (\exists i_0 \cdot i_0 \in NotDuplicate \land iLocation(i) = iLocation(i_0)) \]

This invariant means for an incident to be considered as duplicate, there must be at least one incident with the same location.

5.4.4 Fifth, Sixth and Seventh Refinement: Physical Resources

In the previous sections, we only dealt with logical resources. In the following refinements we start by refining the HandleIncident event showing the different steps in handling an incident, then introducing physical resources which are related to the different stages in handling an incident, finally we add more details related to physical resources.

The HandleIncident flow is shown in Figure 5.10, illustrating how a logical resource \( b_r \) is managed for incident \( i \). Once a physical resource is allocated to a logical resource by the AllocateRes event, it is possible for two changes to occur to the resource any number of times prior to the resource attending the incident or receiving a stop a message: either the physical resource is replaced by a quicker one that becomes available or the physical resource location is updated. Finally after a resource attends the incident and completes its task or if the resource receives a stop a message, the resource will be deallocated from the incident in DeallocateRes event.

![Figure 5.10: Different Steps of Incident Handling](image)

After introducing ResAttend, it is now possible to specify the action plan resources that are not required to be completed (notRequired_ap) in SendStopMsg more precisely, by updating its manually added guard (grd1) of Figure 5.6 to become:

\[ lrs = ap\{\{i\}\} \setminus ResAttend\{\{i\}\} \]

This update is to ensure a stop message is sent to all resources that are part of the action plan, but not in attendance yet, letting them know that the incident is satisfied with the resources currently attending the incident location and no more resources are required to handle the incident.

In the sixth refinement, physical resources are introduced by the data variables alloc and alloc_LR. The following invariants describe alloc and alloc_LR and the relationship
between them.

\textbf{inv1:} \( alloc \in PR \rightarrow INCIDENT \)

\textbf{inv2:} \( \text{ran}(alloc) \subseteq \text{SetInitialAP} \)

\textbf{inv3:} \( alloc_{LR} \in PR \rightarrow LR \)

\textbf{inv4:} \( \forall pr, lr \in \text{dom}(alloc_{LR}) \land alloc_{LR}(pr) = lr \Rightarrow ptype(pr) = ltype(lr) \)

\textbf{inv5:} \( \text{dom}(alloc) = \text{dom}(alloc_{LR}) \)

The data variable \( alloc \) maps physical resources to the incident they are allocated to, \( \text{inv}1 \). The variable \( alloc \) is defined as a partial function so that each physical resource can be assigned to at most one incident. The incident must have an action plan which is why we introduce \( \text{inv}2 \) to ensure that incidents are in \( \text{SetInitialAP} \). Invariant \( \text{inv}3 \) defines \( alloc_{LR} \), which assigns a logical resource to a physical resource. This variable will help us in keeping track of which logical resource in the action plan, the physical resource is assigned to. Finally, \( ptype \) and \( ltype \) are constants defined in the context to describe the type of the physical and logical resources respectively. The last two invariants ensure that the type of a physical resource must be the same as the type of the logical resource assigned to it, and the equality of the domains ensures that all physical resources allocated to an incident are associated with a logical resource.

In Figure 5.11, the event \( AllocateRes \) will add new values to \( alloc \) and \( alloc_{LR} \), whereas \( DeallocateRes \), Figure 5.11, will remove physical resources from \( alloc \) and \( alloc_{LR} \). Therefore, \( DeallocateRes \) will ensure that an incident cannot be closed if it still has allocated physical resources. This is enforced by invariant \( \text{inv}6 \) which states that an incident whether duplicate or not can be only closed if it has no physical resources allocated to it. The following invariant, \( \text{inv}7 \), ensures that all required resources have completed their tasks and have been deallocated before the incident can be closed. This is done by checking that all logical resources in the action plan of the incident \( i \), that are not marked as not required as a result of a send stop message, are part of the relational
image of DeallocateRes with respect to incident $i$.

\[ \text{inv6}: \forall i. i \in (\text{CloseIncident} \cup \text{CloseDuplicate}) \Rightarrow i \notin \text{ran}(\text{alloc}) \]

\[ \text{inv7}: \forall i. i \in \text{CloseIncident} \Rightarrow \text{ap}[\{i\}] \setminus \text{notRequired}_\text{ap}[\{i\}] \subseteq \text{Deallocation}[\{i\}] \]

These invariants show another example of combining control variables with data variables.

In the last refinement level, we extend the model with the location of the physical resource and the duration it needs to get to the incident by introducing the data variables $rLocation$ and $duration$ as follows:

\[ \text{inv}\_rLocation: rLocation \in \text{PR} \rightarrow \text{LOCATION} \]

\[ \text{inv}\_duration: duration \in (\text{LOCATION} \times \text{LOCATION}) \rightarrow \mathbb{N} \]

Initially the physical resources will be associated with some stations, and their location will be set to the station location. The event $\text{Update}\_\text{ResLocation}$ will be extended to update the location of the resource, i.e., update the data variable $rLocation$ of $pr$. We also extend the event $\text{ResAttend}$ with an extra guard to ensure the location of the resource is the same as that of the incident, i.e., $rLocation(pr) = iLoction(i)$ where $pr$ is a resource allocated to the incident $i$. The $duration$, which represents the time needed to get from one location to another, is mainly needed when reallocating a new quicker resource to the incident as shown in Figure 5.12, where $grd5$ checks that the duration of the new physical resource $pr$ to be allocated to incident $i$, is less than the initially allocated resource $pr0$.

![Figure 5.12: Reallocation of a Quicker Resource to the Incident](image)

In Figure 5.12 the only Event-B parts that are generated from the ERS diagram in Figure 5.10 are the first two guards $grd\_seq$ and $grd\_loop$, while the rest of the guards and actions are manually added. $grd1$ checks that $pr0$ is a physical resource allocated to incident $i$ and corresponds to the logical resource $lr$. Guards, $grd2$ and $grd3$, ensure that
the new physical resource \( pr \) to be allocated to incident \( i \) is not allocated to another incident and has the same type as the logical resource \( lr \). \( grd4 \) checks that the logical resource \( lr \) is still required for incident \( i \), otherwise there is no need for reallocation. The actions will swap the allocation of the physical resource \( pr0 \) to \( i \) with the new physical resource \( pr \). This is another example of how we used ERS to manage the control flow of events, while the data flow is done manually using Event-B.

## 5.5 Evaluation of the Fire Dispatch Case Study

We evaluate the model presented in Section 5.4, by tracing the requirements in Table 5.5 to validate that the model (Section 5.4) incorporates all the requirements presented in Section 5.3, and to assess the suitability of the ERS features for the case study. In this requirements trace we show at which refinement levels the requirements were mainly addressed and the directly related data variables and events.

<table>
<thead>
<tr>
<th>Req_ID</th>
<th>Machine</th>
<th>Data Variables</th>
<th>Events and Control Variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>INF_1</td>
<td>M0, M4</td>
<td>iT, iLoc, iLocT</td>
<td>GatherInfo, SelectLocation, SelectType, NonDuplicate</td>
</tr>
<tr>
<td>INF_2</td>
<td>M0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>HND_1</td>
<td>M2, M4</td>
<td>PDA</td>
<td>SetInitialAP</td>
</tr>
<tr>
<td>HND_2</td>
<td>M3</td>
<td>ap</td>
<td>SetInitialAP</td>
</tr>
<tr>
<td>HND_3</td>
<td>M2, M3</td>
<td>ap</td>
<td>UpdateActionPlan, RequestAdditionalRes, SendStopMsg</td>
</tr>
<tr>
<td>HND_4</td>
<td>M3, M4</td>
<td>notRequired_ap</td>
<td></td>
</tr>
<tr>
<td>RES_2</td>
<td>M5, M7</td>
<td>duration, rLoc</td>
<td>ReallocateQuickerRes, UpdateResLocation, ResAttend</td>
</tr>
<tr>
<td>CLS_1</td>
<td>M3, M5, M6</td>
<td>alloc, alloc_LR, alloc_LR</td>
<td>HandleIncident, CloseIncident, AllocateRes, DeallocateRes</td>
</tr>
<tr>
<td>CLS_2</td>
<td>M3, M5, M6</td>
<td>alloc, alloc_LR</td>
<td>CloseIncident, DeallocateRes, CloseDuplicate</td>
</tr>
<tr>
<td>CLS_3</td>
<td>M2</td>
<td></td>
<td>SendStopMsg</td>
</tr>
<tr>
<td>CLS_4</td>
<td>M1</td>
<td></td>
<td>IsDuplicate, CloseDuplicate</td>
</tr>
</tbody>
</table>

Comparing Table 5.5 with the requirements tables in Section 5.3, we have the following requirements missing from the model: \( INF_3, INF_4, INF_5, HND_5, RES_1 \) and \( RES_3 \).

Regarding requirement \( INF_3 \), we introduced the possibility of having duplicate incidents, but we restricted the decision making of a duplicate incident to be only done after gathering information about the incident, and this decision cannot be made if an incident is marked as \( NonDuplicate \), as indicated by the application of the \( xor\)-combinator.
in Figure 5.2. Using the xor-combinator we have not modelled the possibility of interrupting an incident handling if the duplication is discovered late as requirement \textit{INF.3} gives the flexibility of making such decision at any stage, even after the arrival of a resource to the incident. The reason for this restriction is that in ERS we do not have combinators that can interrupt a flow especially in this case the interruption can possibly happen at different places during the incident handling which consist of many events as shown in Figure 5.10. Using the existing ERS combinators this can be done by using multiple xor-combinators to show the different possibilities, which can lead to a complex model, that is why we did not fully model requirement \textit{INF.3}.

Requirements \textit{INF.4} and \textit{INF.5} can be easily modelled with the existing ERS combinators, but at this stage we decided to postpone the introduction of the incident priority due to its effect on resource allocation as required by \textit{HND.5} and \textit{RES.1}. Requirement \textit{HND.5} states that we can allocate a resource to an incident even if it is allocated to another incident on the condition that reallocation takes place from a lower priority incident to a higher priority incident. Furthermore, \textit{RES.1} requires replacing the resource in the lower priority incident or in any incident if its resource becomes unavailable due to a breakdown for example.

Requirement \textit{HND.5} can be easily modelled by changing the guard on resource allocation using Event-B to allow allocating from a lower priority incident to a higher priority incident, but the problem is in requirement \textit{RES.1}. In order to replace a lost resource due to reallocation or breakdown, we need to repeat some events again, such as \textit{AllocateRes}. The problem in ERS is once an event executes for an instance \(i\), it cannot execute again for the same instance due to the disabling guard \(\text{grd}_{\text{self}}\), unless we use a \textit{loop}. The problem in using a \textit{loop} in this case, is the control variables will reset every time, whereas here resetting is only needed in the case a replacement is needed. Another option is to do the resetting of control variables manually, but the manual modification of the control variables is not recommended in ERS and could lead to lots of problems due to the sequencing dependencies between events, hence, we decided to postpone modelling this behaviour at this stage. We also decided to postpone modelling requirement \textit{RES.3} because we have not introduced the unavailability of a resource which is also related to requirement \textit{RES.1}.

This shows the limitation in the ERS approach when dealing with interruptions and cancellation of behaviour. These limitations will be addressed in Chapter 6.

By overcoming the previously mentioned limitations in ERS, we managed to model most of the requirements of the fire dispatch workflow as shown in Table 5.5. We have also shown how in many cases we combined control variables with data variables to model the requirements, for instance invariants \textit{inv6} and \textit{inv7} use both control variables (\textit{CloseIncident}, \textit{DeallocateRes} ...), and data variables (\textit{alloc}, \textit{ap} ...) to enforce requirements \textit{CLS.1} and \textit{CLS.2}. 
The approach we followed in modelling workflows, where we do the control flow using ERS diagrams and leave the data handling to Event-B, seems successful as this leads to simpler diagrams and the generated model can be manually extended with data variables without causing problems. The `ReallocateQuickerRes` event in Figure 5.12 is an example of how we combined both ERS generated Event-B elements with manually added elements.

In addition to that, we managed to relate the data variables to the control variables using invariants, which helps in knowing when or at which stage the data variable will be available. For example the invariant \( \text{dom}(ap) = \text{SetInitialAP} \), indicates that when the `SetInitialAP` event occurs then the data variable \( ap \) will be set, or in other words the incident will have an action plan. This would not have been possible, if we did not have explicit control variables associated to the events.

In this case study we have also applied the ERS extensions introduced in Chapter 4. We used the parallel `producer-consumer` pattern in Figure 5.4 to show how additional resources can be requested while managing an incident, this pattern provides flexibility when dealing with dynamic variables such as the action plan of an incident \( \{ap[i]\} \).

Requirement `CLS_1` which requires that all the tasks in the action plan be satisfied before closing an incident suggests that the `all-replicator` is a possible candidate to represent this behaviour. However, we applied the `par-replicator` due to the dynamic nature of the variable set. We managed to model `CLS_1`, using a combination of control and data variables as shown by `inv7`. This invariant ensures that for an incident to be closed, all the resources that are still required by the action plan are part of the deallocated resources.

If we replaced the `par-replicator` with an `all-replicator`, we will end up with an invariant stronger than `inv7` which cannot be satisfied. This is because the `all-replicator` will require the equality of all the items in the action plan with the deallocated resources before considering an incident as managed, which is not always possible. The problem is in the case when some unallocated resources in the action plan are no longer required due to sending a stop message, hence they cannot be allocated and consequently cannot be deallocated. However, we managed to express the requirement by the manually added invariant `(inv7)`, but in a way that can be satisfied by the model.

We have also used the parameter extension, as we have seen in the case study, where the action plan cannot be determined statically, it depends on the data collected (location and type) by the events, hence the extension of the parameter was needed in this case. The `loop` generalisation was also used in the case of updating the resource location and allocating a quicker resource of the incident (Figure 5.10).

In Event-B, a variable introduced at a refinement level \( i \) can only be modified in subsequent refinement levels by events that refine events of level \( i \). For example, the data
variable \( ap \) in the fire dispatch workflow has been updated and changed by several events, so we postponed the introduction of this application-specific variable until all the events that change its value were introduced. The overall ERS diagram of the dispatch system has helped us to make such decision.

In addition to that, the explicit invariants generated by ERS help in discharging proof obligations. Take for example \( M5 \):

\[
\text{inv\_par\_ref}: \forall i \in \text{IncidentManaged} \implies \text{AllocateRes}\{i\} = \text{DeallocateRes}\{i\}
\]

If we do not have an explicit \textit{par-rePLICator} invariant related to refinement, it will not be possible (without adding an explicit guard) to discharge the proof obligation related to the guard strengthening (GRD) of the \textit{par-rePLICator} guard \((i \notin \text{IncidentManaged})\) in \textit{DeallocateRes} event, as shown in figures 5.13 and 5.14.

![GRD Discharged with ERS Invariant](image)

(a) GRD Discharged with ERS Invariant

![GRD Undischarged without the ERS Invariant](image)

(b) GRD Undischarged without the ERS Invariant

Figure 5.13: Effect of ERS Invariant in Discharging Proof Obligations

### 5.6 Conclusion

In this chapter we have modelled the fire dispatch system workflow, by applying the ERS approach to model the control flow and representing the data handling using Event-B. In modelling the case study we tried to make small changes in each refinement, making the verification and the validation of the model easier than making big changes in each refinement as explained by some studies (Butler and Yadav, 2008; Butler, 2009b). The ERS approach helped in achieving this goal by its refinement restrictions. Most of the proof obligations in this case study where either proved automatically using Rodin provers or automatically with the help of additional external prover plug-ins such as SMT solvers (Figure 5.15). In Figure 5.15, most of the proofs marked as manual are
Figure 5.14: Proof Details of the DeallocateRes/GRD

actually done automatically using external provers or interactively with the help of different external provers.

Figure 5.15: Fire Dispatch Model Statistics

In this case study we have also validated the extensions presented in Chapter 4 showing that they are needed to give ERS the flexibility to model dynamic behaviour. While modelling the case study we identified some limitations in the case study related to interruptions and cancellation behaviour, these limitations will be addressed in Chapter 6.
Chapter 6

Extending ERS by Exception Handling

6.1 Introduction

When developing workflows, we usually concentrate on how we are expecting the system to behave in normal circumstances. However, errors and unexpected behaviours are unavoidable, therefore a good workflow modelling approach should be capable of handling exceptions. According to Casati et al. (1999), exceptions are those occasional asynchronous deviations from the normal flow, which may cause backtracking of previous steps or even sudden termination. Modelling exceptions can be done in Event-B, by defining events that can deal with exceptions, but how can we include exceptions and error recovery in ERS?

First when dealing with exceptions, it is better to distinguish between the normal flow and the exception handling flow. This is to avoid complicating the model with various alternatives related to exceptions (Russell et al., 2006b,c). Therefore, in order to avoid complicating the ERS diagrams with various xor-combinator branches capturing unexpected behaviour and to focus on the expected behaviour of the system, we introduce two explicit combinators that deal with exceptions.

Second, what are the recovery actions or how exceptions can be handled in ERS? In Eder and Liebhart (1996), which is one of the early works about workflow recovery, when recovering from failures or exceptions in workflows, we can do forward execution, backward recovery or forward recovery. Forward execution entails progressing forward by executing remaining activities, backward recovery requires controlled rollback where successfully completed activities are undone, while forward recovery is in most cases a combination of backward recovery and forward execution. Also in Russell et al. (2006b,c), they identify three possible actions for exception handling: no action, rollback, or compensate. In all
of the above recovery actions, we need to provide means to allow the undoing of actions in ERS. This could be also part of the cancellation control patterns which ERS does not support explicitly, but still can be achieved using Event-B.

In this chapter we start with introducing two exception handling combinators for ERS, *interrupt* and *retry*. We also show the application of these exception handling combinators to the case study introduced in Chapter 5. Finally we compare our approach in exception handling with other workflow modelling approaches.

### 6.2 Exception Handling Combinators

In this section we extend the ERS approach with two combinators to support exception handling. The two exception handling combinators are *interrupt* and *retry*. The *interrupt* will stop the execution of a flow, once the interrupt event occurs. The *retry* will also stop the flow execution once the interrupt event takes place, but *retry* will reset the control variables of the flow, in order to enable execution of the normal flow again.

The first child, the left hand side, of the exception handling combinator represents the normal control flow, while the second child, the right hand side, represents the exception or the interrupting flow which can stop the normal flow and transfer control to the exception handling flow. Unlike the other ERS combinators (*xor*, *and*...), the children (*normal-child* and the *interrupting-child*) of the exception handling combinators are not commutative, since their effect is not symmetric.

In Figure 6.1, we present a generalised representation of the *interrupt* and the *retry* combinators respectively. $P$ represents the normal sub-flow, while $I$ represents the interrupting sub-flow. $R$ with combinator $C$ and $Q$, represent respectively the preceding and follow-on flows of the exception handling combinator.

![Generalised Form of the interrupt and the retry Combinators](image)

**Figure 6.1:** Generalised Form of the *interrupt* and the *retry* Combinators

Table 6.1 defines the enabling and disabling conditions of the normal sub-flow $P$ and the interrupting sub-flow $I$, and the enabling conditions for the follow-on flow $Q$, for both the *interrupt* (Figure 6.1(a)) and the *retry* (Figure 6.1(b)) combinators. In both cases,
the enabling condition for \( P \) is determined by the combinator \( C \) of the preceding flow \( R \). However in the case of \textit{retry}, \( P \) can also be enabled by the completion of \( I \).

<table>
<thead>
<tr>
<th>Enable P</th>
<th>Interrupt Combinator</th>
<th>Retry Combinator</th>
</tr>
</thead>
<tbody>
<tr>
<td>Disable P</td>
<td>( P ) has executed or ( I ) has made a step</td>
<td>( P ) has executed or ( I ) has made a step</td>
</tr>
<tr>
<td>Enable I</td>
<td>Same enabling condition as ( P )</td>
<td>Same enabling condition as ( P )</td>
</tr>
<tr>
<td>Disable I</td>
<td>( P ) has executed</td>
<td>( P ) has executed</td>
</tr>
<tr>
<td>Enable Q</td>
<td>( P ) or ( I ) has executed</td>
<td>( P ) has executed</td>
</tr>
</tbody>
</table>

The other difference between the two combinators is the enabling condition of the follow-on flow \( Q \). Enabling \( Q \) following the \textit{interrupt-combinator} requires the execution of either \( P \) or \( I \), while for the \textit{retry-combinator}, \( Q \) can be only enabled after the execution of \( P \). The rest of the conditions are the same for both combinators, so \( P \) will be disabled if it completes execution or if \( I \) starts execution. The interrupting sub-flow \( I \) has the same enabling conditions as the normal sub-flow \( P \) of its exception handling combinator, which depends on the previous flow in both cases while \textit{retry} has an additional possible enabling condition which is the completion of \( I \). In both cases \( I \) will be disabled if \( P \) completes execution.

### 6.2.1 The Interrupt Combinator Semantics

The aim of the \textit{interrupt} combinator is to stop the execution of a sub-flow once a certain event, which we refer to as the interrupting event, takes place.

The similarity between the \textit{interrupt-combinator} and the \textit{xor-combinator} is that the sequencing of the follow-on sub-flow, if it exists, requires the execution of either child of the \textit{interrupt-combinator}. However the main difference between the \textit{interrupt} and the \textit{xor} is that in the \textit{xor-combinator}, once one child starts execution, it disables the execution of all other children of the \textit{xor-combinator}, whereas the \textit{interrupt-combinator} allows the interrupting sub-flow of the \textit{interrupt-combinator} to stop the execution of the normal sub-flow at any point before it completes execution.

For example, a user shopping on-line can decide at any time before completion to abort or cancel the order. In this case the cancel order event is the interrupting event of the \textit{interrupt-combinator}, and the on-line shopping process is the normal sub-flow of the \textit{interrupt-combinator}. 
Figure 6.2: Single Instance Semantics of the *Interrupt* Combinator

**Diagrammatic Representation:** Similar to the other ERS combinators, the *interrupt-combinator* is represented within an oval shape with the “X” symbol, as shown in Figure 6.1(a). The *interrupt-combinator* is always connected to its parent using dashed lines to satisfy the *single solid line* rule of ERS, since the normal sub-flow can execute partially if the interrupt sub-flow executes. The *interrupt-combinator* is always connected to exactly two dashed nodes, the first node represents the normal flow and the second node represents the interrupting flow.

**Event-B Semantics:** The Event-B semantics of the *interrupt-combinator* is shown in Figures 6.2 and 6.3 for the single instance case and the multiple instances case respectively. In both figures the first child, in this case $B$, represents the normal child of the *interrupt-combinator* and the second child, in this case $I$, represents the interrupting child. At the abstract level (Figures 6.2 and 6.3), the *interrupt-combinator* is similar to the *xor-combinator*, the difference between the two combinators is only relevant once refinement is introduced (Figure 6.5).

As shown in the Event-B models of both the single instance case and the multiple instances case, the sequencing constraint of $I$ is the same as $B$, which represents the normal sub-flow. In this case, $I$ can follow the execution of $A$ as shown by the sequencing...
guards (\textit{grd\_seq}) and invariants (\textit{inv\_I\_seq}) of the interrupting event \( I \). In the single instance case \( I \) requires the execution of \( A \) (\( A = \text{TRUE} \)) to be enabled, while in the multiple instances case, \( I \) can be enabled for a certain instance of parameter \( p \), only if \( A \) has executed for that instance (\( p \in A \)).

The interrupting event \( I \) will be disabled once the normal sub-flow (\( B \)) completes execution, as shown by the guard \textit{grd\_interrupt} of \( I \). In the single Instance case, the disabling guard is (\( B = \text{FALSE} \)) and in the multiple instances case (\( p \notin B \)).

For the normal sub-flow, we add a disabling guard, \textit{grd\_interrupt}, to stop the execution of the normal sub-flow if \( I \) executes. In the single instance case the disabling guard of \( B \) is (\( I = \text{FALSE} \)) and in the multiple instances cases (\( p \notin I \)).

Both \( B \) and \( I \) will be disabled if the other executes, so we enforce this behaviour by the \textit{inv\_I\_interrupt}, where the \textit{partition}(\( S, s_1, ..., s_n \)) means that, the union of all the elements (\( s_1..sn \)) constitute \( S \), and all the elements (\( s_1..sn \)) are disjoint.
Refinement: Both the normal leaf and the interrupting leaf of the interrupt-combinator can be refined. However, when refining the normal child of the interrupting-combinator, B in Figures 6.2 and 6.3, the last child in the refinement must be the refining event or the solid child, which can be explained by Figure 6.4. Allowing I to interrupt between B1 and B2 is inconsistent with refinement: at the abstract level, I is disabled once B occurs. Therefore in the refinement, I must be disabled once B1 occurs. Hence, the need for the restriction that the solid child of the normal flow of the interrupt-combinator must be always the last child of the flow.

![Figure 6.4: Invalid Refinement of the Interrupt Combinator](image)

The Event-B semantics of introducing refinement to the normal sub-flow (B) of Figure 6.2 is shown in Figure 6.5. As we can see the interrupt disabling guard (grd\textsubscript{interrupt}) will be added to each leaf (Bi) of the normal sub-flow, stopping the execution of the normal sub-flow at any point I executes. The interrupting event I can be only disabled if the last child of the normal sub-flow (Bn) executes, that is why the mutual exclusiveness property is only between I and Bn as shown by the invariant inv\textsubscript{I, interrupt}. The same thing applies to the multiple instances case of Figure 6.3.

Refining the interrupting event I does not have any restrictions, in this case the execution of the first child of I will always stop the normal flow, whether the first child is solid or dashed and in both strong and weak sequencing.

6.2.2 The Retry Combinator Semantics

The retry-combinator is similar to the interrupt-combinator, in the sense that an interrupting event can stop the execution of the normal flow at any time before completion. However, the main difference between the two combinators is that, once the interrupting event stops the normal flow and completes execution, the normal flow is enabled again. This combinator is useful when some exception happens and the user needs to roll back to some point in the workflow.

Diagrammatic Representation: Similar to the other ERS combinators, the retry-combinator, Figure 6.1(b), is represented within an oval shape with the “X*” symbol. The “X", the symbol of the interrupt-combinator, to indicate the interruption action and the “*”, the symbol of the loop, to indicate the possible repetition of the flow.
Similar to the interrupt-combinator, the retry-combinator is always connected to its parent node using dashed line and is always connected to exactly two dashed nodes, the first representing the normal sub-flow and the second representing the interrupting sub-flow.

**Event-B Semantics:** The Event-B semantics of the retry-combinator is shown in Figures 6.6 and 6.7, for the single instance and multiple instances cases respectively. The encoded Event-B related to sequencing and disabling conditions of an event in the normal sub-flow $B$ and the interrupting event $I$, are similar to those of the interrupt-combinator, Figures 6.2 and 6.3. However, the interrupting event $I$ of the retry-combinator has additional actions to reset the control variables of the normal sub-flow. In the single instance case the normal sub-flow $B$ is reset to FALSE, and in the multiple instances case the parameter $p$ is removed from the set $B$. 

---

**Figure 6.5: Refinement Semantics of the normal sub-flow $B$ of Figure 6.2**
At the abstract level (Figures 6.6 and 6.7), the behaviour of the retry-combinator can be represented by a loop as shown in Figure 6.8. However, when the normal-child ($B$) is refined, this equivalence with the loop is not applicable.

Resetting the control variable, $B$, of the normal sub-flow is not enough to enable the normal sub-flow due to the presence of the interrupting guard $grd\_interrupt$. Therefore, we add a resetting event, event $reset\_I$, to reset the value of the interrupting event after its execution. The resetting event can execute after the completion of the interrupting flow, which is represented by the sequencing guard of the reset event ($reset\_I$). The sequencing guard of the resetting event is $I = TRUE$ in the single instance case and $p \in I$ in the multiple instances case. The action(s) of the reset event will reset the control variables of the interrupting sub-flow, in this case there is only one event $I$, so
Figure 6.7: Multiple Instances Semantics of the Retry Combinator

Figure 6.8: Behaviour of retry (Figure 6.6) at Abstract Level

the resetting action will be $I = FALSE$ in the single instance case, and $I = I \setminus \{p\}$ in the multiple instances case.

Another difference to the interrupt-combinator is the sequencing of the follow-on event, $C$, which depends only on the normal sub-flow, so the normal sub-flow must complete
execution without interruption to enable \( C \). This is represented by the sequencing guard \( B = TRUE \) in the single instance case and \( p \in B \) in the multiple instances case and their associated invariants \( inv.C.seq \). If it is possible for the retry exception to persist, we can add an interrupt-combinator at a higher level to avoid retrying forever. For example in Figure 6.9, \( I_0 \), the interrupting event of the interrupt-combinator, can interrupt both \( B \) and \( I \) of the retry-combinator.

![Figure 6.9: Introducing Exception Handling at Two Levels](image)

**Refinement:** The same refinement restrictions of the interrupt-combinator applies to the retry-combinator, i.e., when refining the normal child of the retry-combinator the last child must be the solid child.

One can argue that a solution to avoid the restriction of the solid last child of the normal flow, is to have the follow-on event of the retry-combinator \( (C) \) to disable the interrupting sub-flow. This does not work, first because we do not always have an event after the retry-combinator. Second and most importantly in the retry-combinator case, this solution would make it possible for the normal sub-flow to execute successfully more than once. This contradicts the aim of the retry combinator which is only to retry the normal flow if an exception occurs, represented by the occurrence of the interrupting event, i.e., the normal sub-flow can complete execution only once, while the interrupting sub-flow can complete execution more than once.

Figure 6.10 shows the encoded Event-B of the refinement of \( B \) in the multiple instances case. Similar to the interrupt-combinator the interrupt disabling guard \( (grd.interrupt) \) will be added to each child leaf \( Bi \), and the interrupting event \( I \) will be disabled by the execution of the last child of the normal sub-flow \( (Bn) \) and enforced by the invariant \( inv.I.interrupt \). The event \( I \) will also reset each child leaf \( Bi \) of the normal sub-flow. The same applies to the single instance case.

### 6.3 Application to the Fire Dispatch Workflow

In this section we model the fire dispatch case study introduced in Chapter 5 using the exception handling combinators. The aim of this section is to model the requirements not
covered in Chapter 5 due to limitations in the ERS approach as explained in Section 5.5 of Chapter 5. The complete version of this model is presented in Appendix B.

In this version of the fire dispatch model, the dynamic part consists of an abstract and ten refinement levels, while the static part consists of seven contexts. The control flow of the model is done using the ERS approach, but in the $9^{th}$ refinement ($M9$), we use a state machine to model the different states of a physical resource. Data handling of the

---

**Figure 6.10: Refinement Semantics of the normal sub-flow $B$ of Figure 6.7**

---
case study is also done manually using Event-B, where additional data variables were added alongside the ERS control variables. Most of the requirements which were already done in Section 5.4 of Chapter 5 are the same and we will only show the differences in this section.

The machines of the case study are modelled as follows:

- **M0**: Models the creation of an incident and identifying it as duplicate or not.
- **M1**: Refines the non-duplicate incident case, introducing the information gathering and setting the action plan of the incident and managing the incident.
- **M2**: Models the ability to update the action plan of the incident by requesting additional resources.
- **M3**: Introduces further incident details such as incident location and type.
- **M4**: Introduces the priority of the incident.
- **M5**: Refines the incident management according to the action plan of the incident.
- **M6**: Introduces the possibility of losing an allocated resource and how to handle this exception.
- **M7**: Shows in further details, how an incident is handled and how allocated resources can be lost.
- **M8**: Introduces physical resources which are actually assigned to an incident and how a duplicate incident is handled in case duplication is discovered after resource allocation.
- **M9**: Introduces the different states of a physical resource using a state machine.
- **M10**: Adds further details about the location and the duration of a physical resource to the incident.

### 6.3.1 Distinction Between Duplicate and Non-Duplicate Incidents Using the Interrupt Combinator

The main difference starts right at the abstract level (Figure 6.11), where in order to model requirement $INF_3$ in Table 5.1, we represent the $IsDuplicate$ event as an interrupting event of the *interrupt-combinator* introduced in Chapter 6. Replacing the *xor-combinator* in Figure 5.2 with the *interrupt-combinator* in Figure 6.11 gives the controller the flexibility to determine that an incident is duplicate at any point, hence there is no need to continue with the normal activities associated with the non-duplicate incidents, which are shown by refining the *Not-Duplicate* event.
In the first refinement \((M1)\), the atomicity of \(NotDuplicate\) is broken into the sequence of events starting with gathering information about the incident \((GatherInfo)\), followed by setting an initial action plan to manage the incident \((SetInitialAP)\), then the concurrent execution of managing the incident \((ManageIncident)\) and updating the action plan \((UpdateAP)\), finally after the incident is managed the incident call can be closed \((CloseIncident)\), after which the incident cannot be interrupted by the \(IsDuplicate\) event.

In the case study, we are using multiple instances modelling, therefore sequencing between the events is determined by the subset sets approach, for instance the invariant to describe the order of execution of \(GatherInfo\) event will be \(GatherInfo \subseteq CreateIncident\), similarly the ordering of \(IsDuplicate\) depends only on \(CreateIncident\), as a result of using the interrupt-combinator. Figure 6.12 shows the invariants of the first refinement level of the model \((M1)\), where all these invariants are the result of transforming the ERS diagram of Figure 6.11 to Event-B.

```
\begin{verbatim}
| invariants |
| @inv_CreateIncident_type CreateIncident \subseteq INCIDENTS |
| @inv_GatherInfo_seq GatherInfo \subseteq CreateIncident |
| @inv_SetInitialAP_seq SetInitialAP \subseteq GatherInfo |
| @inv_ManageIncident_seq ManageIncident \subseteq SetInitialAP |
| @inv_UpdateAP_seq UpdateAP \subseteq SetInitialAP |
| @inv_CloseIncident_seq CloseIncident \subseteq (UpdateAP \land ManageIncident) |
| @inv_IsDuplicate_seq IsDuplicate \subseteq CreateIncident |
| @inv_CloseIncident_glu CloseIncident = NotDuplicate |
| @inv_IsDuplicate_interrupt partition((CloseIncident \cup IsDuplicate), CloseIncident, IsDuplicate) |
\end{verbatim}
```

Figure 6.12: Invariants of the First Refinement Level (M1)

The gluing invariant \((inv\_CloseIncident\_glu)\) showing the equality of \(CloseIncident\) with \(NotDuplicate\) is the result of the solid line connecting \(CloseIncident\) to \(NotDuplicate\), which conforms to the restriction of the interrupt-combinator, where the last event...
must be the refining event. The last invariant is the result of applying the interrupt-combinator, where if the normal sub-flow (Not-Duplicate) completed execution successfully, it cannot be interrupted by IsDuplicate, and if IsDuplicate executes before completing execution, the normal sub-flow cannot complete execution. This invariant is different from the xor-combinator invariant where in case of the xor-combinator the partition will be between the first event (GatherInfo) and IsDuplicate. At the abstract level (M0) there is no difference between applying the xor-combinator or the interrupt-combinator to the events NotDuplicate and IsDuplicate, the difference appears in the refinement of the normal sub-flow (M1), where each event will have the guard \( i \notin \text{IsDuplicate} \) and not only the first event like applying the xor-combinator.

![Event-B Specification of some Events of the Refinement Level (M1)](image)

Figure 6.13: Event-B Specification of some Events of the Refinement Level (M1)

In Figure 6.13, we show the events Closeincident and IsDuplicate of the first refinement level of the model. All the guards and the actions of these events are the result of the ERS diagram in Figure 6.11. The guard \( i \notin \text{IsDuplicate} \) (grd Interrupt) is the result of the interrupt-combinator, and will be also inherited by the events (GatherInfo, SetInitialAP, ManageIncident, UpdateAP) to show the interruption effect of executing the IsDuplicate event.

6.3.2 Introducing the Priority of the Incident

Introducing the priority of the incident, requirements INF_4 and INF_5 in Table 5.1, was possible in the model presented in Chapter 5, but we decided to leave it due to its relation with requirements HND_5 and RES_1 in tables 5.2 and 5.3 respectively.

In order to relate incident types with a priority level, we add the axiom \( \text{default_priority} \in INCIDENT\_TYPE \rightarrow 1 \cdot 3 \) to the context associating each incident type with a default priority level that can range between one and three, where priority level one is the highest priority.

In the dynamic part of the model, we introduce the data variable priority which is defined by \( \text{inv_priority} \), the invariant uses a total function to ensure that each incident
with a selected type will have a priority level.

**inv_priority:** $\text{priority} \in \text{SelectType} \rightarrow 1 \cdot 3$

We refine $\text{SelectType}$ as shown in Figure 6.14 to introduce the priority of the incident. Requirement $\text{INF}_4$ does not specify how often the priority of the incident is allowed to change and when we can change it, therefore when introducing the priority we left it open, by allowing changing the priority zero or more times using the loop, moreover since it is not specified when the priority can be updated, we introduced the priority change in the refinement using weak sequencing. Applying the weak sequencing interpretation means that sequencing constraints only applies between the solid event ($\text{SelectType}$) and the left and right hand side events of the previous refinement level. As a result, $\text{CompInfo}$ ordering will remain the same as before where its sequencing invariant is $\text{CompInfo} \subseteq (\text{SelectLocation} \cap \text{SelectType})$. However, sequencing still holds at the same level, i.e., between $\text{SelectType}$, $\text{ChangePriority}$ and $\text{DisablePriorityChange}$ events. Therefore, it is possible to change the priority during the incident management, which makes sense because the incident officer can assess upon arrival the right priority of the incident, and having the possibility to be changed more than once is also acceptable as things might happen during the incident requiring the change of the priority.

![Figure 6.14: Introducing Priority of the Incident Using Weak Sequencing](image)

In Figure 6.14, we manually set the priority of the incident $i$ to the default priority when selecting the type of the incident ($\text{SelectType}$). We can also see how all the events related to the non-duplicate case, inherited the interrupting guard ($i \not\in \text{IsDuplicate}$). The $\text{ChangePriority}$ event will update the current priority to a new one as shown by the
action of the event \( (\text{priority}(i): = p) \). In both events, \text{ChangePriority} and \text{DisablePriority}, we add the guard \( i \notin \text{CloseIncident} \), this means that we will disable the priority change when the incident is closed.

### 6.3.3 Handling the Lost Allocated Resources with the Retry Combinator

The next main update in the model is handling resources which are lost due to reallocation to a higher priority incidents or due to breakdown, in order to cover requirements \textit{HND.5} and \textit{RES.2} in Table 5.2 and Table 5.3 respectively. For this, we use the \textit{retry-combinator} introduced in Chapter 6 to interrupt the flow of the resource and allow a replacement resource to be allocated to the incident.

The incident management of a non-duplicate incident, which also constitutes the consumer part of the action plan in Figures 5.4 and 5.10 of Chapter 5 will be updated as shown in Figure 6.15 to introduce the\textit{ retry-combinator} which handles the resource loss.

![Figure 6.15: Incident Management Using the Retry Combinator](image)

In Figure 6.15 we ended up with an additional refinement level due to the ERS restriction that do not allow more than one combinator to be applied to the same event at the same level, which actually makes the modelling process simpler by making smaller changes at each level. For example, applying more than one combinator to several events at the same time can result in more complex sequencing invariants that are harder to read and prove, whereas dividing these changes in separate stages and making use of the ERS refinement gluing invariants will result in simpler sequencing invariants. In this case we had to introduce the resource handling of a resource in two steps before introducing a
detailed handling process, which are first introducing the *par-replicator* then the *retry-combinator*, because the loss and replacement of a resource applies to a single resource at a time.

As explained earlier in Chapter 6, the last event in the refinement of the normal part of an exception handler must be the solid line, as the interrupting event, in this case *RemoveAlloc* and *RemoveBrokenRes*, cannot interrupt the handling of the incident and the incident does **not** require a replacement after being deallocated.

After introducing the physical resources, allocation of resources is updated to allow reallocation from lower priority incidents to higher priority incidents so the condition that \((pr \notin \text{dom}(alloc))\) in Figure 5.11(a) is not applicable. In the updated version of the model, *AllocateRes* will allow allocation of physical resources that are already in the domain of *alloc* as shown in Figure 6.16(a), where *grd3* ensures that if *pr* which is the physical resource to be allocated, was in the domain of *alloc*, then the incident it was allocated to must be of lower priority than the current incident *i*. *RemoveAlloc* (Figure 6.16) is one of the interrupting events that is related to *AllocateRes* event, where *RemoveAlloc* is enabled when a resource gets reallocated from a lower priority incident to a higher priority incident. This relation is specified by the manually added guards *grd1* and *grd2*, where *grd1* specifies that allocation of a resource occurred for the logical resource *lr*, but *grd2* indicates that there is no actual physical resource allocated to the logical resource *lr* by checking that the mapping from *i* to *lr* is not in the forward composition of the inverse of *alloc* and *alloc_LR*, in other words the allocated resource for the logical resource is lost.

In this case *RemoveAlloc* will remove the maplet \((i \mapsto lr)\) from *AllocateRes, ResAttend, StopMsgReceived, DeallocateRes* to enable allocating a new physical resource for the logical resource *lr*. The replacement will be done after resetting *RemoveAlloc* (Figure 6.17(b)) because *RemoveAlloc* is an interrupting event. Another possible interruption is a broken down resource as indicated by the interrupting event *RemoveBrokenRes* in Figure 6.15, the only difference with *RemoveAlloc* is that removing the physical resource allocation from the incident \((alloc, alloc_LR)\) will be done in this event and the broken resource will not be allocated to another incident (Figure 6.17(a)).

In both interrupting events *RemoveAlloc* and *RemoveBrokenRes*, we can see the interrupting guard \(i \notin \text{IsDuplicate}\), since the duplicate interrupt is at higher level, therefore it can still interrupt the lower level interrupts (*RemoveAlloc, RemoveBrokenRes*).

### 6.3.4 Handling Allocated Resources to Duplicate Incidents

In the previous version of the case study in Chapter 5, deciding that the incidents are duplicate was only done before handling the incidents and allocating physical resources
Chapter 6 Extending ERS by Exception Handling

(a) AllocateRes Event (M8)

```
Event AllocateRes refines AllocateRes
any i lr pr
where
  @grd_seq i ∈ SetInitialAP
  @grd_self i → lr ∈ AllocateRes
  @grd_Exp lr ∈ ap\(\{i\}\)
  @grd_par i ∈ IncidentManaged
  @grd_Cancel i ∈ IsDuplicate
  @grd_Retry i → lr ∈ (RemoveAlloc ∪ RemoveBrokenRes)
  @grd1 i → lr ∈ notRequired_ap
  @grd2 ptype(pr) = ltype(lr)
  @grd3 pr ∈ dom(alloc) ⇒ priority(i) < priority(alloc(pr))
then
  @act AllocateRes := AllocateRes ∪ \{i → lr\}
  @act1 alloc(pr) := i
  @act2 alloc LR(pr) := lr
end
```

(b) RemoveAlloc Event (M8)

```
Event RemoveAlloc refines RemoveAlloc
any i lr
where
  @grd_self i → lr ∈ RemoveAlloc
  @grd_seq i ∈ SetInitialAP
  @grd_par i ∈ IncidentManaged
  @grd_Retry i → lr ∈ RemoveBrokenRes
  @grd_Cancel i ∈ IsDuplicate
  @grd_Retry i → lr ∈ DeallocateRes
  @grd1 i → lr ∈ AllocateRes
  @grd2 i → lr ∈ alloc~; alloc LR
then
  @act RemoveAlloc := RemoveAlloc ∪ \{i → lr\}
  @act1 AllocateRes := AllocateRes \{i → lr\}
  @act2 ResAttend := ResAttend \{i → lr\}
  @act3 StopMsgReceived := StopMsgReceived \{i → lr\}
  @act4 DeallocateRes := DeallocateRes \{i → lr\}
end
```

Figure 6.16: Event-B Specification of the Allocation of Resources and its Cancellation at the Refinement Level (M8)

to it, thus requirement \(CLS_2\) of Table 5.4 is always satisfied for closing duplicate incidents since allocation of resources cannot be done for duplicate incidents. However, in Section 6.3.1, we have updated the model to allow the \(IsDuplicate\) event to interrupt the handling of resources, which means that in some cases it is possible to have allocated physical resources to a duplicate incident. That is why we refine the \(IsDuplicate\) event as shown in Figure 6.18 to handle the allocated resources of a duplicate incident that is discovered late.

Using the all-replicator in \(HandleDupAlloc\) ensures that requirement \(CLS_2\) is satisfied by removing all the physical allocations before closing the duplicate incident in the manually added actions \(act1\) and \(act2\). This is an example of how we compensated the
Chapter 6 Extending ERS by Exception Handling

(a) RemoveBrokenRes Event (M8)

(b) Reset_ResLost Event (M8)

Figure 6.17: Event-B Specification of RemoveBrokenRes and the Resetting Event at the Refinement Level (M8)

allocated resources in the case of duplicate exception. In Figure 6.18, it was possible to use the all-replicator and benefit from its sequencing constraint that affect CloseDuplicate because the control variable AllocateRes, which in this case represents the parameter set of the all-replicator, cannot be changed once IsDuplicate event executes.
6.3.5 Extending the Model with State Machines

As explained in Section 5.5 of Chapter 5, we did not model requirement RES_3 of Table 5.3, because we have not modelled the unavailability of a resource due to breakdown for example. Now we have introduced the possible unavailability of a resource as shown in Section 6.3.3, hence it is possible to model the different states of physical resources.

In modelling the different states of a resource we found using a state machine is more suitable than an ERS diagram, because state machines are best known for describing the state changes of objects. Hence in this case the ERS diagram will describe the overall behaviour of the fire dispatch system, while the state machine will be used to focus on the state changes of the physical resources.

We extended the model with the state machine in Figure 6.19, which shows the different states of a physical resource, which are in this case Available and Unavailable. The available state of a physical resource is in turn divided into two cases when the resource is free, i.e. not allocated, the physical resource can be either StationAvailable or Returning, while in the case of an allocated physical resource, it can be either Mobilised or InAttendance.

![Figure 6.19: State Machine of the Status Change of a Physical Resource](image)

In modelling the state machine in Figure 6.19, we used the iUML-B statemachine (Snook, 2014; Wiki.event-b.org, 2014), which is another form of UML-B (presented in Section 3.3.5 of Chapter 3), but with a stronger integration with Event-B. The dark circle in the middle of the state pr_allocated is a feature of iUML-B called junction which generates a disjunctive guard.

In the iUML-B state machine, we used the Enumeration translation to Event-B, where the other translation option is Variable translation, we also applied lifting as shown
in Figure 6.20, where we specified the instance set and the self-name to \( PR \) and \( pr \) respectively. Using lifting in iUML-B is another feature which allows the state machine to be lifted to a set of instances so that the state machine represents the behaviour of all the instances in the set (Wiki.event-b.org, 2014), which is \( PR \) in our case.

![Figure 6.20: Properties Overview of the Physical Resource State Machine](image)

The iUML-B translation to Event-B will generate an implicit context to define the different states of the state machine according to the required translation type. The generated sets are: \( pr\_status\_STATES \), \( pr\_available\_sm\_STATES \), \( pr\_allocated\_sm\_STATES \), \( pr\_free\_sm\_STATES \). These sets represent the different state machines in Figure 6.19, where we have used four state machines, the overall state machine \( pr\_status\_STATES \), which contains another state machine \( pr\_available\_sm\_STATES \), which in turn consists of two state machines \( pr\_allocated\_sm\_STATES \) and \( pr\_free\_sm\_STATES \). We have used multiple state machines to make the diagram simpler compared to having lots of transitions from each state.

The different states of the state machines are defined using the following axioms which are generated from the iUML-B state machines in Figure 6.19. Each state is defined as a constant in the related state machine set, and the states ending with the keyword \( \text{NULL} \) corresponds to states outside the current state machine:

\[
\begin{align*}
\text{axm\_sm1} & \quad \text{partition}(pr\_status\_STATES, \{\text{Available}\}, \{\text{Unavailable}\}) \\
\text{axm\_sm2} & \quad \text{partition}(pr\_available\_sm\_STATES, \{pr\_allocated\}, \{pr\_free\}, \{pr\_available\_sm\_NULL\}) \\
\text{axm\_sm3} & \quad \text{partition}(pr\_allocated\_sm\_STATES, \{\text{Mobilised}\}, \{\text{InAttendance}\}, \{pr\_allocated\_sm\_NULL\}) \\
\text{axm\_sm4} & \quad \text{partition}(pr\_free\_sm\_STATES, \{\text{Returning}\}, \{\text{StationAvailable}\}, \{pr\_free\_sm\_NULL\})
\end{align*}
\]
In Figure 6.19, the transitions from one state to another are marked with the event names that update the state of the physical resource. Some of the events are common to the fire dispatch system presented in the ERS diagrams. The events common with the ERS diagrams will be extended with status update (Figure 6.21(a)), whereas the uncommon events are newly added events to the model that are only related to the physical resources, but not related to the incidents (Figure 6.21(b)). In both cases the state of the resource before the event execution will be added as a guard, while the status update will be done using action(s) as shown in Figure 6.21.

![Diagram](image)

(a) An ERS Event Extended With Resource Status (M9)

![Diagram](image)

(b) New Event Related to the Resource Status Only (M9)

Figure 6.21: Event-B Specification of Some of the Events Updating the Resource Status

In Figure 6.21, the variables $pr_{\text{available_{sm}}}$, $pr_{\text{allocated_{sm}}}$ and $pr_{\text{free_{sm}}}$ are automatically defined as total functions relating the physical resources (PR) with the corresponding statemachine set. In addition to the automatically generated invariants, we have manually added invariants to relate the status of a physical resource with $\text{alloc}$. These invariants are:

$$\text{inv}_{\text{pr}_{\text{alloc}}} \forall pr \cdot pr_{\text{available_{sm}}}(pr) = pr_{\text{allocated}} \iff pr \in \text{dom}(\text{alloc})$$

$$\text{inv}_{\text{pr}_{\text{alloc}} \_2} \forall pr \cdot pr_{\text{available_{sm}}}(pr) \in \{pr_{\text{free}}, pr_{\text{available_{sm\_NULL}}}\} \iff pr \notin \text{dom}(\text{alloc})$$

The first invariant ($\text{inv}_{\text{pr}_{\text{alloc}}}$) will ensure that allocated resources are available, while the second invariant ($\text{inv}_{\text{pr}_{\text{alloc}} \_2}$) will ensure that all free resources and unavailable ones are not allocated to an incident.

Only one of the events in the ERS diagram that involve physical resource change was not shown in the statemachine (Figure 6.19), this event is $\text{ReallocateQuickerRes}$. The reason for this is that $\text{ReallocateQuickerRes}$ involves the state change of two different physical
resources, the initially allocated physical resource and the quicker physical resource to be allocated. We have done the status change manually and we followed the same approach followed by the statemachine translation, so we extended \textit{ReallocateQuickerRes} as shown in Figure 6.22.

\begin{figure}[h]
\centering
\begin{verbatim}
event ReallocateQuickerRes extends ReallocateQuickerRes
  where
    @grd5 pr_available_sm(pr_quicker) = pr_free
    @grd6 pr_allocated_sm(pr) = Mobilised
  then
    @act3 pr_available_sm := {(pr, pr_quicker) \in pr_available_sm} \cup {pr \mapsto pr_free}
    \cup {pr_quicker \mapsto pr_allocated}
    @act4 pr_free_sm := {(pr, pr_quicker) \in pr_free_sm} \cup {pr \mapsto Returning} \cup 
    {pr_quicker \mapsto pr_free_sm_NULL}
    @act5 pr_allocated_sm := {(pr, pr_quicker) \in pr_allocated_sm} \cup {pr \mapsto pr_allocated_sm_NULL} \cup {pr_quicker \mapsto Mobilised}
end
\end{verbatim}
\caption{ReallocateQuickerRes Extended Manually with the Resource Status Update}
\end{figure}

The \textit{ReallocateQuickerRes} will only allocate a quicker resource, if that resource is free, i.e., not allocated to another incident, and the status of the current allocated resource is \textit{Mobilised}. When reallocating the quicker resource, the status of the previously allocated resource (\textit{pr}) will be changed to free and that of the quicker resource (\textit{pr_quicker}) will be changed to allocated as shown by \textit{act3}. Moreover in \textit{act4} the status of \textit{pr} will be \textit{Returning} in the sub-state machine \textit{pr_free_sm}, while that of \textit{pr_quicker} will be removed and assigned to the null state (\textit{pr_free_sm_NULL}). In \textit{act5}, \textit{pr_quicker} will be updated to \textit{Mobilised} while that of \textit{pr} will be removed from the \textit{pr_allocated_sm} state machine.

\subsection{Evaluation of the Updated Fire Dispatch Case Study}

Modelling the fire dispatch system with the new exception handling combinators (Chapter 6), made it possible to model the requirements related to interruptions and cancellation, such requirements were difficult to model previously due to the impact they have on the workflow course. That is why having additional explicit combinators, like the exception handling combinators, is important to show the effect of one sub-flow on the other, whereas in previous combinators all the sub-flows of the combinator were treated the same way.

We showed how applying the \textit{interrupt-combinator} managed to represent how a late duplicate incident discovery can stop future allocations to the incident, and how we handled any allocated resources to the duplicate incident. Additionally, we successfully applied the \textit{retry-combinator} to attempt reallocating a new resource replacing a lost physical resource due to reallocation to higher priority incident or breakdown. Using
these explicit combinators showed the possibility of such exceptions and how they are handled, without complicating the diagrams.

In this model we have also combined a statemachine with the ERS diagrams, this shows how ERS can be easily extended with other control flow approaches, as well as any additional Event-B data.

### 6.4 Comparison with Other Work on Exception Handling

In this section we compare our ERS exception handling approach to other workflow modelling approaches, such as activity diagrams and BPMN.

#### 6.4.1 UML Activity Diagrams

UML activity diagrams support exception handling mechanisms using Interruptible Activity Regions and Exception Handlers. Interruptible Activity Regions, Figure 6.23(a), are a group of activities where execution can terminate once a certain behaviour connected to an interrupting edge is accepted. Once the region is interrupted the execution will transfer to a target activity outside the interruptible region but connected to the interrupting edge. An interrupting edge will have its source within the interruptible activity region and its target outside the region. Activity diagrams also have explicit exception handlers similar to the “try catch” statement used in programming, Figure 6.23(b). When the execution of the protected node completes whether the exception handler was triggered or not, execution will continue to the actions following the protected node as normal.

![Interruptible Activity Region](image1) ![Exception Handler](image2)

Figure 6.23: Activity Diagrams Exception Handling Mechanisms (Object Management Group (OMG), 2015)

The ERS exception handling combinators, _interrupt_ and _retry_, can be compared to the interruptible activity region due to the way an exception is triggered through external events. Similar to the interruptible activity region, the _interrupt_ and the _retry_ combinators abort the execution of the normal flow, which in an activity diagram is within the interruptible activity region, and execution is transferred to the interrupting event. Unlike the _retry-combinator_, the interruptible activity region does not retry the execution of the interrupted region until it succeeds. However, it is possible to represent this behaviour if we connect the interrupting event and its handler back to the main flow.
Consider the example of the order workflow using the interruptible activity region, Figure 6.24 which is taken from Object Management Group (OMG) (2015). In Figure 6.24, if Cancel Order is triggered, according to activity diagrams the payment flow starting with Send Invoice will not be interrupted. However, the join-node needs both Ship Order and Accept Payment to execute, hence the join-node will never be executed and will be implicitly interrupted. This kind of behaviour cannot be modelled directly with our ERS combinators as all of the normal flow (including payment) is interruptible. Given that the join-node of Figure 6.24 is not executed if the order is cancelled, it is not clear that the behaviour allowed by interruptible activity regions is desirable.

![Figure 6.24: Interruptible Activity Region (Object Management Group (OMG), 2015)](image)

### 6.4.2 BPMN

BPMN supports exception handling and compensation through different types of intermediate events. Intermediate events (Object Management Group, 2011) are represented by a circle which can be attached to the boundary of an activity or task to represent exception handling. There are different types of intermediate events, some of them are interrupting and others are not. We are only interested in interrupting intermediate events like the error intermediate event, represented by a flash symbol, and the cancel event, represented by an X marker. The error intermediate event is used to catch an error and always interrupts the activity it is attached to and diverts the flow according to what the error event is connected to. Therefore, it is similar to the interrupt-combinator, and if it is connected back to the same activity, its behaviour will be similar to the retry-combinator. Similarly, the cancel event can also interrupt processes but it can be only attached to transaction sub-processes.

BPMN also has a compensation intermediate event, represented by a rewind symbol, which when triggered will roll back completed activities. In compensation events, the
interrupting property does not hold as only completed activities can be compensated. However, the compensation will be typically raised by an error handler as part of a cancellation process (Object Management Group, 2011). In compensation, activity roll-back will be done in reverse order of execution and can be performed concurrently if no sequencing is applied. Unlike the ERS retry which will only reset the control variables of all the activities in the interrupting region whether completed or not and this will be done concurrently. Compensating other manually added data is the responsibility of the modeller, as we have seen in Figure 6.18, where we refined the interrupting event IsDuplicate and added additional events to manually compensate for allocated resources.

Figure 6.25 (Object Management Group, 2011) shows a transaction for flight and hotel bookings, where each transaction has an associated compensation which will be triggered in case of failed bookings, represented by the cancel intermediate event, undoing all successful bookings. Exceptions and hazards are represented by the error intermediate event, which when triggered will interrupt the bookings and divert the flow to be handled by customer service.

In Figure 6.26, we show a possible ERS representation of Figure 6.25. In Figure 6.26, we have represented the transaction for a single instance case. We have also introduced two data variables ReservedF and ReservedH to represent the successful flights and hotels bookings respectively. The booking cancellation is done manually using Event-B as we can see by the manually added actions in CancelFlight and CancelHotel events. We have
decided to model the cancellation events using a \textit{loop} to cover the case where there are no successful bookings.

Regarding multiple instances, compensation in BPMN will be done for all instances, while in ERS as we have seen in Figure 6.15 we can apply it to a single instance. In ERS if we wanted to apply the \textit{retry-construct} of Figure 6.15 to all the instances then we could have introduced it at a level higher than the \textit{par-replicator}.

6.4.3 Little JIL

Little JIL (Wise et al., 2000) is a formal language, yet graphical, that supports process coordination in hierarchical steps. Figure 6.27 shows the graphical representation of a Little-JIL step, showing how Little-JIL represents control flow and exception handling. Regarding control flow which is specified by the parent step, at the leftmost of the step, Little-JIL supports sequential, parallel, choice and try flows of its sub-steps. In a choice flow, exactly one sub-step is executed and in a try flow, execution of sub-steps starts from left to right and stops when a sub-step completes successfully.

Exception handling which is the main focus of this chapter is represented by the “X” symbol at the rightmost of the step, which can be connected to the exception handler. If the exception did not find a matching handler, it goes up the decomposition tree until finding a match. After the exception handler completes, a continuation badge connected
Chapter 6 Extending ERS by Exception Handling

Figure 6.27: Little-JIL Step (Wise et al., 2000)

to the handler will determine whether the step will continue execution, successfully complete, restart execution or rethrow the exception (Group, 2006; Wise et al., 2000).

The ERS retry-combinator will be similar to an exception with a restart execution as the continuation badge of the handler, while the interrupt-combinator will be like an exception with a successfully complete badge. If there is an event following the interrupt-combinator, it will execute if either the normal flow completed execution, or the interrupt flow completed execution. Continuing the execution of the step, even if an exception occurs, is not represented by the ERS exception handling combinators. Rethrowing the exception means after handling the exception, execution will terminate, this can be only represented if there is no follow-on event after the interrupt-combinator, such as the IsDuplicate interrupt example of the fire dispatch case study in Section 6.3.1. Another difference between the exception handlers and the ERS interrupts is that ERS will inherit the interrupt and the handler which can be introduced as a refinement down the tree, so there is no need to propagate up the tree like little-JIL to look for a matching handler, which in ERS can be only associated to one interrupting flow.

In the Little-JIL step, there is also a lightning bolt symbol in the middle of the step, which represents received messages that can arrive during the execution of the step. The messages are connected to reactive steps which should be acted on immediately once received. The difference with the ERS interrupt is that the reactive steps can execute in parallel with the normal steps, while in ERS the messages which can be represented as interrupting events will stop the execution of the normal flow. Reactive messages unlike exceptions are local to the step and do not propagate up the tree.
6.4.4 Exception Handling and Fault Tolerance Patterns

The workflow initiative (Russell et al., 2006b,c) have also identified patterns for exception handling, which take into consideration four important aspects:

1. What is the exception type.
2. How the work item that captured the exception will be handled.
3. How other work items in the case will be handled.
4. What recovery action will be taken.

These four aspects combined together resulted in many patterns. They identified five exception types for expected exceptions, which are work item failure, deadline expiry, resource unavailability, external trigger and constraint violation. They also identify fifteen strategies for exception handling at work item level such as: restart (SRS), force fail (SFF). For handling at the case level they have three cases: continue (CWC), remove current case (RCC) and remove all cases (RAC). Recovery actions are three: no action (NIL), rollback (RBK) and compensate (COM).

The patterns are given abbreviated names related to the last three considerations and grouped according to the exception type. An example pattern is “SFF-CWC-COM” which is one of the patterns specified for the exception type work item failure. The first part of the name relates to work item level, the second part to case level and the last part to recovery action.

In ERS we do not have this large number of patterns and we do not differentiate between exception types. In ERS we give the mechanism to capture the exception and leave it to the user to define the handling process. Regarding retry we provide the mechanism to support approaches like rollback and restart because the user should not modify control variables, whereas handling data variables is left to the user, which can be introduced during the refinement of the interrupting events.

Another exception handling pattern classification is done in Lerner et al. (2010) which classifies the exception patterns into three generic approaches as follows:

1. Trying other alternatives: Ordered Alternatives and Unordered Alternatives patterns
2. Inserting behaviour: Immediate Fixing, Deferred Fixing, Retry, and Exception-Driven Rework patterns
3. Canceling behaviour: Reject and Compensate patterns
They also attempted to represent the exception handling patterns using Little-JIL, BPMN and activity diagrams. The three modelling languages provide support for exception handling, some have advantages in certain areas, e.g., Little-JIL provides better support to ordered and unordered alternatives due to the choice and try combinators, while BPMN has a compensate combinator providing better support to compensate patterns.

In ERS there is no direct support for the ordered alternatives pattern as found in Little-JIL with the \texttt{try} combinator, but similar to activity diagrams this can be represented by chaining the alternatives using different exception handlers. Unordered alternatives are represented using the \texttt{choice} combinator in little-JIL and in activity diagrams using a combination of conditional nodes, a loop and exception handler. In ERS and Event-B the events are atomic so that they cannot be interrupted unless we consider the alternative involves more than one event, in which case it could be interrupted by an exception. If the alternatives consist of one event, i.e., the alternative either execute or not, then this can be done using the \texttt{xor-combinator}. Otherwise, it is possible to represent this pattern indirectly using a combination of an \texttt{or-combinator} which allows the unordered execution of its branches, the \texttt{interrupt-combinator} and additional Event-B guards to check the execution of the \texttt{or-combinator} branches and the \texttt{interrupt-combinator} branches.

The patterns immediate fixing and retry patterns can be represented by the ERS \texttt{interrupt-combinator} and the \texttt{retry-combinator} respectively. Immediate fixing requires adding extra work before joining back the normal flow, this can be represented by the \texttt{interrupt-combinator}, since we can add additional actions after interrupting the normal flow, and as explained earlier if there is a flow following the \texttt{interrupt-combinator}, it can execute if either the normal flow completed execution or the interrupting flow completed execution. The \texttt{retry} pattern allows trying the activity again if it fails by possibly doing some context update before attempting the activity again, this behaviour can be achieved by the \texttt{retry-combinator}, which allows trying the activity again by resetting the control variables of the ERS diagram.

On the other hand deferred fixing and exception-driven rework are other versions of the immediate fixing and retry but allowing time to pass between the occurrence of the exception and its handling. There is no direct support for deferred fixing in ERS. Regarding rework and the ERS \texttt{retry-combinator} this is not possible because the \texttt{retry-interrupt} will reset the control variables of the normal flow once it occurs and the follow-on event depends on the normal flow execution, hence we cannot postpone the retrying of the normal flow.

The intent of the reject pattern is that some intended behaviour can become undesirable and the agent must be notified allowing the agent to take action or make changes and try again, so that the reject pattern can be used to abort a process or part of a process. In this case this pattern can be represented by either the \texttt{interrupt} or the \texttt{retry} combinator.
of ERS depending on the required handling as both combinators can interrupt the normal flow which is the intended behaviour.

Regarding the compensate pattern, which is part of the cancelling behaviour as described by Lerner et al. (2010), it provides a mechanism to determine what is the successfully completed work and undo it with the associated compensation actions, when cancellation of the activity occurs. If we compare the compensate pattern to the ERS retry-combinator, retry will only reset the control variables regardless if they have executed or not, and it is left to the modeller to undo other data related activities. Hence in order to undo only completed activities using either the retry or interrupt combinator of ERS, additional checks are needed, here we are talking about undoing data related actions. Hence the compensate pattern is not directly supported by ERS as there is no direct mechanism to only undo completed activities.

In Ball and Butler (2007), the authors define fault tolerance patterns for multi agent systems in Event-B, which are timeout, refuse, cancel, failure and not-understood pattern. These patterns are related to communication messages between different agents. Of those patterns, the cancel pattern is the only one that can be compared to the ERS exception handling combinators. The cancel pattern can be represented using the ERS interrupt-combinator, where an agent no longer requires to complete an action. The refuse pattern can be represented by the xor-combinator as the agent have the choice to accept or refuse to perform an action. It is also possible to represent the fail pattern using the interrupt-combinator or the xor-combinator, as the failure pattern is similar to the refuse pattern but it happens after commitment of an agent to an action while refuse happens before commitment. The timeout and the not-understood patterns are not represented explicitly by ERS.

6.4.5 Other Work on Compensation and Formal Languages

Compensation has also been studied by integrating formal methods with other approaches. In Butler and Ferreira (2003); Butler et al. (2005a), the authors use StAC (Structured Activity Compensation), which is a business process modelling language that was introduced by Butler and Ferreira (2000) to support parallel and sequential behaviour and most importantly to support compensation. Similar to our approach where we use ERS for the coordination of events and the Event-B formal method for data handling, they use StAC to represent the orchestration of business transactions while the B method is used for data manipulation.

Compensation in StAC is done by defining compensation pairs, one representing the normal process and the other representing the compensating process. The execution of the compensating process is done in reverse order of the normal process. The compensation process is remembered until it is no longer required, for this two operators
are introduced one called reverse which refers for the invocation of compensation or its execution and the other is called accept which indicates that the compensation process is no longer required and can be forgotten. They also provide a mechanism for compensation nesting using scoping brackets. They also extend StAC by introducing indexing for compensation which is used to support multiple compensations. An important operator in indexed StAC is the merge operator which allows merging all the compensating tasks belonging to the same task in parallel.

Indexing in StAC also allows the modelling of named compensation in BPEL, where in Butler et al. (2005a) they show how StAC can be used to give BPEL formal semantics. In Chessell et al. (2002), they also use indexed StAC to define two forms of compensation called selective and alternative. In selective compensation, the reversal chooses which activities should be compensated, while in alternative compensation, the reversal can choose which compensation task to execute among several alternatives.

Compared to our approach, we only provide the interrupting mechanism which the user can refine and define their own exception handling mechanism, which could also include different compensation alternatives or applying interleaving of different compensating tasks depending on what ERS combinators the modeller choose and the activities they define. In this sense, our ERS exception handling combinators are similar to the StAC reverse operator but without explicitly defining the compensation mechanism, with retry having an additional feature to support compensation by facilitating the activity redo. Regarding scoping, the ERS interruption will not be applicable once the normal flow completes execution, but we also allow the nesting of the different exception handling operators, as seen in Section 6.3.

In Butler et al. (2005b), they extend CSP (Hoare et al., 1985) with compensating operators, where they also use the concept of compensation pairs. Compensation pairs are similar to the StAC compensation pairs, where they define the compensation task for a successfully completed task. In Compensating CSP, they also define the operator THROW for throwing interrupts and a special operator for handling interrupts. Our ERS interrupt-combinator can be similar to the combination of interrupt Throw and interrupt handler.

### 6.5 Conclusion

The Interrupt combinator can be used whenever it is possible to abort a flow or a sub-flow at any time. The Retry combinator can be used whenever there is a need to stop the flow or sub-flow and redo it again.
Having a combinator like the *Interrupt* combinator has the advantage of avoiding multiple *xor* branches, since the interrupting event can start at any time before the completion of the normal flow. The *retry* combinator will reset the control variables of the interrupted flow, allowing the flow to be retried again thus allowing behaviour such as rollback. Any compensations can be added to the interrupting events or can be added through refinement following the interrupting event. By this ERS can support both forward and backward recovery actions.

Compared to other approaches, the *interrupt* and *retry* combinators capture similar exception handling mechanisms as other approaches, but in ERS we do not differentiate between the exception types. Moreover ERS does not support transaction compensation behaviour as found in languages such as BPMN, but supports a generic handling approach which is decided by the modeller according to the application.
Chapter 7

Travel Agency Case Study

7.1 Introduction

In this chapter, we model the on-line booking system of a travel agency. In modelling the case study, we apply some of the extended ERS approach, presented in Chapter 4 and Chapter 6, and Event-B. The model represents the on-line booking system of a trip, where a user can book different flights, hotels and/or cars.

In modelling this case study, we show the importance of having an explicit interrupt combinator in ERS, which allows the user to quit the entire booking process at any time. In addition to the importance of having a flexible replicator like the par-replicator, whenever the number of instances to be executed cannot be predetermined.

7.2 Requirements Overview of the Travel Agency Case Study

The on-line booking system of a travel agency allows a user to select and book multiple flights, cars and/or hotels. If a user selected a flight/car/hotel and had them in their itinerary for a long time without booking, the flight/hotel/car will be removed from their itinerary. In addition, it is possible for the user to cancel the whole trip at any time before checking out.

Many other approaches have modelled the travel agency example using compensations, such as Chessell et al. (2002), where they used StAC and Wong and Gibbons (2011) where they used BPMN.
7.3 Incremental Development of the Travel Agency Case Study

The Event-B model of the travel agency case study consists of two contexts representing the static part of the model, and one abstract and four refinement levels representing the dynamic part of the model as follows:

- **Abstract Level:** Models starting a trip, getting the itinerary and the possibility of quitting.
- **First Refinement:** Refines the trip itinerary to more detailed flight, car and hotel itinerary.
- **Second Refinement:** Introduces booking flights, cars and hotels.
- **Third Refinement:** Introduces the possibility of removing a flight, car and/or hotel if not booked.
- **Fourth Refinement:** Introduces deadline to remove flight, car and/or hotel from itinerary if not booked within a certain time.

In the following sections, we only show the refinement and the Event-B specifications of the flight itinerary, but the same applies to car and hotel itineraries. The complete version of the model is presented in Appendix C.

7.3.1 Abstract and First Refinement

We model the travel agency booking system for multiple instances, where at the context we define the carrier set \( \text{TRIP} \) representing all the possible trips. The user can quit the trip at any time before checkout, that is why we introduce \( \text{Quit} \) using the \texttt{interrupt-combinator}, as shown in Figure 7.1.

In Event-B the effect of \( \text{Quit} \) appears in the generated guard \( t \notin \text{Quit} \) in all the leaf events of \( \text{TripItinerary} \), which are in this case: \( \text{FlightItinerary}, \text{CarItinerary}, \text{HotelItinerary} \) and \( \text{Checkout} \). The user cannot quit the trip if \( \text{Checkout} \) is completed and the \texttt{interrupt-combinator} will ensure this by the guard \( t \notin \text{Checkout} \), which is added to \( \text{Quit} \), in addition to the invariant: \( \text{partition}((\text{Checkout} \cup \text{Quit}), \text{Checkout}, \text{Quit}) \).

We can see in Figure 7.1, that the last event of the normal sub-flow of the \texttt{interrupt-combinator} must be always the solid line, which is in this case \( \text{Checkout} \). As explained in Section 6.2.1 of Chapter 6, the importance of the \texttt{interrupt-combinator} appears after refining the normal child of the \texttt{interrupt-combinator}, to show that the interrupting event \( \text{Quit} \) can stop the execution of the normal sub-flow at any point before completion.
7.3.2 Second and Third Refinement

In the second refinement we extend the context to introduce the sets \textit{FLIGHT}, \textit{CAR} and \textit{HOTEL}, representing the possible flights, cars, hotels that can be added to the trip itinerary respectively. In Figure 7.2, we show the refinement of \textit{FlightItinerary}, where \textit{CarItinerary} and \textit{HotelItinerary} will be refined in a similar way but when applying the \textit{par-replicator}, we use the parameter sets \textit{CAR} and \textit{HOTEL} respectively.

The \textit{FlightItinerary} is refined to book or remove the selected flights from the flight itinerary of the trip for individual flights. When selecting flights, similarly for hotels and cars, we apply the \textit{par-replicator} because the number of flights to be selected is unknown. In the third refinement, we also introduce three data variables \textit{flights}, \textit{cars} and \textit{hotels} representing those flights, cars and hotels added to the trip itinerary. The data variables are defined manually using the following invariants, which relate trips to
flights, cars and hotels. We use a relation in defining these variables because the same flight can be booked by different trips and a trip can book different flights. Here, the carrier set $FLIGHT$ represents the flight number that is why it can be booked by several trips, the same applies to hotels and cars.

$$\text{inv}_{\text{flights}} \; \text{flights} \in \text{TRIP} \leftrightarrow \text{FLIGHT}$$

$$\text{inv}_{\text{cars}} \; \text{cars} \in \text{TRIP} \leftrightarrow \text{CAR}$$

$$\text{inv}_{\text{hotels}} \; \text{hotels} \in \text{TRIP} \leftrightarrow \text{HOTEL}$$

In the model once a flight is selected ($\text{SelectFlight}$), it is added to the flight itinerary which is represented by the variable flights (Figure 7.3(a)), and if $\text{RemoveFlight}$ is executed, it is removed from the set $\text{flights}[[t]]$ (Figure 7.3(b)), which represents all the flights of trip $t$. In Figure 7.3, only the data related guard $\text{grd1}$ and the action $\text{act1}$ of both events are added manually, while the rest are generated from the ERS diagram in Figure 7.2.

![Event-B Specification of the Events Updating the Flight Itinerary](a) SelectFlight Event (b) RemoveFlight Event

We have represented the relation between $\text{SelectFlight}$, $\text{RemoveFlight}$ and $\text{flights}$ using the following invariants:

$$\text{inv}_{\text{flights select}} \; \text{flights} \subseteq \text{SelectFlight}$$

$$\text{inv}_{\text{flights remove}} \; \text{RemoveFlight} \cap \text{flights} = \phi$$

Similar invariants are also added manually relating $\text{hotels}$ and $\text{cars}$ with their corresponding control variables. The first invariant ensures that $\text{flights}$ cannot be added to the itinerary unless selected, while the second invariant ensures that a flight cannot be in the itinerary, if it is removed.

The $\text{par-replicator}$ refinement rules, Section 4.5.1 of Chapter 4, will ensure that the flight itinerary cannot be completed ($\text{CompFlightItin}$), unless all the selected flights are
either booked or removed from the itinerary, as shown by \( \text{grd}_{\cdot \cdot} \) of \( \text{CompFlightItin} \) in Figure 7.4, which is also ensured by an invariant.

\[
\begin{align*}
\text{event } \text{CompFlightItin} & \text{ refines } \text{CompFlightItin} \\
& \text{any } t \\
& \text{where} \\
& \text{grd}_\text{seq}\_t \in \text{StartTrip} \\
& \text{grd}_\text{self}\_t \notin \text{CompFlightItin} \\
& \text{grd}_\text{par}\_t \Rightarrow \text{SelectFlight}([t]) = \text{BookFlight}([t]) \cup \text{RemoveFlight}([t]) \\
& \text{grd}_\text{interrupt}\_t \notin \text{Quit} \\
& \text{then} \\
& \text{act} \text{CompFlightItin} = \text{CompFlightItin} \cup \{t\} \\
\end{align*}
\]

Figure 7.4: \( \text{CompFlightItin} \) Event

If a user decides to \( \text{Quit} \), all the flights corresponding to the trip will be removed from the flight itinerary, similarly for cars and hotels. In this case we decided to remove all the flights, cars, and hotels related to the quitted trip in one step as shown by actions \( \text{act1} \), \( \text{act2} \) and \( \text{act3} \) of Figure 7.5. However it is possible to remove each booked flight, car and hotel separately using an approach similar to the refinement of \( \text{IsDuplicate} \) when handling allocated resources (Figure 6.18).

\[
\begin{align*}
\text{event } \text{Quit} & \text{ refines } \text{Quit} \\
& \text{any } t \\
& \text{where} \\
& \text{grd}_\text{seq}\_t \in \text{StartTrip} \\
& \text{grd}_\text{self}\_t \notin \text{Quit} \\
& \text{grd}_\text{interrupt}\_t \notin \text{Checkout} \\
& \text{then} \\
& \text{act} \text{Quit} = \text{Quit} \cup \{t\} \\
& \text{act1 flights} = \{t\} \in \text{flights} \\
& \text{act2 hotels} = \{t\} \in \text{hotels} \\
& \text{act3 cars} = \{t\} \in \text{cars} \\
\end{align*}
\]

Figure 7.5: Refinement of \( \text{Quit} \) to Remove Booked Flights, Hotels and Cars from Itinerary

In Figure 7.6, we show another possible approach to deal with booked flights, hotels and cars, by refining the \( \text{Quit} \) event and handling each booked item in the itinerary individually. However in the model, we have done it on one step as shown in Figure 7.5, to show different possible ways depending on the case study.

### 7.3.3 Fourth Refinement

In the last refinement level, we do not use ERS diagrams, we only extend the model with time constraints, to remove an item from the itinerary, if it is not booked after a certain time. We follow a similar approach as presented in Sarshogh and Butler (2011).
for modelling the expiry pattern, where we first record when the item was selected and then add a guard to the booking event to check if it passed a certain time.

We introduce the data variables $time$ and $deadline$ to respectively record the current time and define a time limit an item can be reserved for a trip before being removed. The data variables $tFlight$, $tCar$ and $tHotel$ will record the time an item is selected. These data variables are defined manually using the following invariants:

\[
\begin{align*}
\text{inv}_\text{time} & \quad time \in \mathbb{N} \\
\text{inv}_\text{deadline} & \quad deadline \in \mathbb{N} \\
\text{inv}_\text{tFlight} & \quad tFlight \in flights \rightarrow \mathbb{N} \\
\text{inv}_\text{tCar} & \quad tCar \in cars \rightarrow \mathbb{N} \\
\text{inv}_\text{tHotel} & \quad tHotel \in hotels \rightarrow \mathbb{N}
\end{align*}
\]

The data variables, $time$ and $deadline$, are defined as natural numbers, while $tFlight$, $tCar$ and $tHotel$ are total functions assigning each flight, car, and hotel in the itinerary to a natural number representing the selection time. In Figure 7.7, we show how the time is introduced to the model. First in Figure 7.7(a), we introduce a new event $Clock$ which only increments time, then once a flight is selected, similarly for cars and hotels, we set its time to the current time as shown in Figure 7.7(b).

In Figure 7.8, we extend the flight booking and removal with time where we do the same for cars and hotels. When booking a flight, we need to check that it did not pass the deadline as shown in $grd1$ of Figure 7.8(a). When removing a flight from an itinerary, we also remove its time tracking, as it is not required any more as shown by $act2$ of Figure 7.8(b). When removing a flight we did not add a guard to check that it passed the deadline, to give the user the flexibility to remove unwanted flights.
7.4 Evaluation of the Travel Agency Case Study

All the proof obligations of the travel agency case study were discharged automatically by the Rodin provers as shown in Figure 7.9, which shows the simplicity of the invariants generated by the ERS diagrams.

In the model, when a flight is removed (RemoveFlight) then it is not possible to book the same flight again, due to the disabling guard \( t \mapsto f \notin SelectFlight \) in the SelectFlight event. In this case it was fine because the requirements are not clear about that situation. However, a better solution to the model is applying the retry combinator, which allows the flight to be added again to the itinerary as shown in Figure 7.10. In this case, it would be possible to select the removed flight again because the interrupting event (RemoveFlight) will reset the control variables SelectFlight, CustomiseFlight and BookFlight. In Figure 7.10, we have added the event CustomiseFlight, which is not added in Figure 7.2 to show how in this case the retry combinator can appear clearly as a more
suitable combinator than the *xor-combinator* to model this interrupting behaviour. The same also applies to the car and hotel bookings.

Another possibility would be using the generalised *loop*. All the different options in modelling the same behaviour show how the details needed in the requirements can make one combinator more suitable than the other. In addition to the flexibility the new ERS extensions gave us the ability to cover more cases.

In this model, we have also shown how we related the data variables to the control variables like invariants *inv_flights_select* and *inv_flight_remove* in Section 7.3.2.

In Section 7.3.3, we have shown how we can easily extend the ERS models with timing patterns like the expiry pattern. Such timing patterns combined with exception handling combinators like in Figure 7.10 seems very compatible and can be studied further in the future.

In addition to that, we have seen in Section 7.3.2, how the guard generated by ERS as a result of the *par-replicator* refinement has ensured the proper behaviour of the model, where in this case an itinerary cannot be completed if a selection is not dealt with, by either booking or removing it form the itinerary (*grd_par* in Figure 7.4).

### 7.5 Conclusion

In the travel agency case study, we have shown how we applied some of the ERS extensions introduced in chapters 4 and 6. Using these extensions, we successfully managed
to model the requirements. The new ERS extensions have proved to be necessary where in some cases it was not possible to model the behaviour using the original ERS alone.

We have also shown how we can extend the models generated by ERS with additional patterns like the expiry pattern described in Section 7.3.3, where ERS controlled the flow of the events and using Event-B we handled the data changes.

Such distinction between the control flow and the data handling resulted in simpler diagrams that can be easily understood. The distinction between the control flow and the data handling did not prevent us from explicitly relating the control variables and the data variables, as we have seen in Section 7.3.2.
Chapter 8

ERS: Formal Specification and Tool Support

8.1 Introduction

In this chapter we will formally describe the ERS pattern extensions presented in chapters 4 and 6, using “Augmented Backus–Naur Form” (ABNF) (Overell and Crocker, 2008). The new ERS extensions involve some changes to the existing ERS translation rules which were defined by Salehi (Salehi Fathabadi, 2012; Salehi Fathabadi et al., 2012; Fathabadi et al., 2014) to include the new pattern changes, in addition to some new rules that are only related to the new patterns.

Regarding tool support, the ERS extensions are partly developed in the ERS plug-in (Dghaym et al., 2016), a tool supporting the ERS approach in Rodin (Section 3.3.4). However, as future work, the plug-in is extending to support all of the extensions.

8.2 ERS Language Formal: Specification

In formalising the ERS extensions, we follow the same approach used to formalise the existing ERS language which adopts “Augmented Backus–Naur Form” (ABNF) (Overell and Crocker, 2008). ABNF specifications are a set of derivation rules written as “rule = definition”. The following ABNF (Overell and Crocker, 2008) operators are used to describe the current and the extended ERS language:

- Elements enclosed in parenthesis are treated as one element and their order is important.
- Elements enclosed in square brackets are optional.
• Terminal values which represent string values are enclosed between double quotes.

• Alternatives are separated by forward slash (“/”), i.e., $A/B$ means $A$ or $B$.

• A star operator (“*”) preceding an element indicates repetition. We can also specify upper and/or lower bounds by $(m*n)$, where $m$ represents the lower bound and $n$ represents the upper bound, e.g., $^*A$ indicates the possible repetition of $A$ zero or more times, $2^*A$ indicates that $A$ can be repeated 2 or more times etc.

The syntax of the original ERS language is presented in Table 8.1, while the extended ERS syntax is in Table 8.2, where we highlight the extensions in red.

```
<table>
<thead>
<tr>
<th>Syntax</th>
<th>Original ERS Syntax</th>
<th>Extended ERS Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>par</td>
<td>(name)(type)</td>
<td>(name)(type)[range-expression]</td>
</tr>
<tr>
<td>flow</td>
<td>“flow”(name, <em>par, sw)(1</em>child(ref))</td>
<td>“flow”(name, <em>par-exp, sw)(1</em>child(ref))</td>
</tr>
<tr>
<td>child</td>
<td>“leaf”(name)/constructor/1*flow</td>
<td>“leaf”(name)/1*flow</td>
</tr>
<tr>
<td>cons-child</td>
<td>“leaf”(name)/1*flow</td>
<td>“leaf”(name)/1*flow</td>
</tr>
<tr>
<td>constructor</td>
<td>(“and”/“or”/“xor”)(2*cons-child)/(“all”/“some”/“one”)(par)</td>
<td>(“and”/“or”/“xor”)(2*cons-child)/(“all”/“some”/“one”)(par)</td>
</tr>
<tr>
<td>normal-child</td>
<td>cons-child</td>
<td>cons-child</td>
</tr>
<tr>
<td>interrupting-child</td>
<td>cons-child</td>
<td>cons-child</td>
</tr>
</tbody>
</table>
```

Table 8.1: Original ERS Syntax

Table 8.2: Extended ERS Syntax

The ERS syntax in Tables 8.1 and 8.2 can be described as follows:

• “par” represents a parameter, which can be added to an event to indicate multiple instances modelling. It is updated to “par-exp” in order to include the property “range_expression” to allow using variables and not only carrier sets and constants. The properties “name” and “type” remain the same describing the name and the type of the parameter.

• A “flow” represents a single root of an ERS diagram, the only difference in the updated version is the use of “par-exp” instead of “par”. A flow has a name, zero or more parameters and the boolean property “sw”. The boolean property “sw” is one in case of strong sequencing and zero in case of weak sequencing. A flow has
also one or more children with the boolean property “ref” referring to whether the child is refining or not. “ref” will be one in case of a solid line and zero in case of a dashed line.

- The “child” and the “cons-child” remain the same. A “child” can be a “leaf”, a “constructor”, or one or more “flows” when it is further decomposed during a refinement. A “cons-child” is similar to a “child”, but limited to be a “leaf” or “flow(s)" and not a “constructor”.

- A “constructor” can be a non-replicator (“and”, “or”, “xor”) with two or more “cons-child”. Alternatively, a “constructor” can be a replicator with one parameter and one “cons-child”. A replicator can be “all”, “some”, “one”, or a “par-rep” which is added in the updated version; or a “constructor” can be a “loop” which is updated to permit one or more “cons-child”. Otherwise, a constructor can be a an “exception-combinator” which is introduced in Chapter 6.

- An “exception-combinator” has two child nodes, both are “cons-child” nodes, but since its child nodes are not symmetrical, we give them different names (“normal-child”, “interrupting-child”) to distinguish between them.

---

![Figure 8.1: An Example of an ERS Diagram](image)

In order to aid understanding of the extended syntax (Table 8.2), we present an example of ERS diagram in Figure 8.1 and the associated syntax in Table 8.3. This example will be gradually explained to present the translation rules in the following sections.
Table 8.3: ABNF Representation of the ERS Example in Figure 8.1

<table>
<thead>
<tr>
<th>Abstract Level:</th>
<th>1st Refinement:</th>
<th>2nd Refinement:</th>
</tr>
</thead>
<tbody>
<tr>
<td>flow(process_Name, p, 1)</td>
<td>flow(process_name, p, 1)</td>
<td>flow(process_name, p, 1)</td>
</tr>
<tr>
<td>(leaf(A))(0),</td>
<td>(flow(A, p, 1)</td>
<td>(flow(A, p, 1)</td>
</tr>
<tr>
<td>par-rep(p1)(leaf(B))(0),</td>
<td>loop(leaf(H), leaf(I))(0),</td>
<td>loop(leaf(H), leaf(I))(0),</td>
</tr>
<tr>
<td>leaf(C)(0),</td>
<td>leaf(J)(1),</td>
<td>leaf(J)(1),</td>
</tr>
<tr>
<td>interrupt(leaf(D))(leaf(E))(0),</td>
<td>par-rep(p1)(flow(B, (p, p1), 1)</td>
<td>par-rep(p1)(flow(B, (p, p1), 1)</td>
</tr>
<tr>
<td></td>
<td>(all(p2)(leaf(K))(0),</td>
<td>(all(p2)(leaf(K))(0),</td>
</tr>
<tr>
<td></td>
<td>leaf(L)(1))(0),</td>
<td>leaf(L)(1))(0),</td>
</tr>
<tr>
<td></td>
<td>leaf(C)(0),</td>
<td>leaf(C)(0),</td>
</tr>
<tr>
<td></td>
<td>interrupt(flow(D, p, 1)</td>
<td>interrupt(flow(D, p, 1)</td>
</tr>
<tr>
<td></td>
<td>(retry(leaf(M))(leaf(N))(0),</td>
<td>(retry(leaf(M))(leaf(N))(0),</td>
</tr>
<tr>
<td></td>
<td>leaf(O)(1))</td>
<td>leaf(O)(1))</td>
</tr>
<tr>
<td></td>
<td>(leaf(E))(0),</td>
<td>(leaf(E))(0),</td>
</tr>
<tr>
<td></td>
<td>leaf(F)(0))</td>
<td>leaf(F)(0))</td>
</tr>
</tbody>
</table>

8.3 Auxiliary Functions

In defining the new translation rules, we are reusing some of the auxiliary functions defined in Salehi Fathabadi (2012), which are updated to include the new extensions in
chapters 4 and 6, in addition to some newly defined functions. The auxiliary functions are used to help in defining invariants and guards in the output Event-B element of the translation rules.

A more detailed definition of the auxiliary functions reused from Salehi Fathabadi (2012) and how they are updated to include the new ERS extensions can be found in Appendix D, but now they can be summarised as follows:

- **list_of_leaves**: Is a recursive function used to traverse down the sub-tree of a child to get the leaves at the final refinement level of the sub-tree. The result of this function is a list of leaf events with their names and parameters. We have not used this function in our translation rules but used two modified versions, `list_first_child_leaves` and `list_child_leaves`, which will be introduced in Section 8.3.1. For example, if we apply `list_of_leaves` to the sub-flow `A(p)` in Figure 8.1, the result is `G(p)` in a strong sequencing interpretation and `J(p)` in a weak sequencing interpretation. If we apply the function to `D(p)`, the result is `S(p), T(p)` in a strong sequencing interpretation and `O(p)` in a weak sequencing interpretation. However, the function that we use `list_first_child_leaves` (which will be introduced in Section 8.3.1) returns the same result in both strong and weak sequencing.

- **predecessor**: Is used to find the previous node of a leaf. This function is needed when defining the sequencing between two leaves. The predecessor of the \(i^{th}\) child of a sub-tree which is not the first child of the sub-flow is \((i^{th} - 1)\) child of the sub-tree, whereas, that of the first child of a sub-tree is its parent predecessor. In the case the predecessor is a loop or a par-replicator, we find their predecessor again due to the zero execution case in both combinators (Figure D.1 in Appendix D).

- **successor**: Is used to find the next node of a loop. The successor of the \(i^{th}\) child of a flow is its right sub-tree which is the \(i^{th} + 1\) child of a flow. A loop cannot be the first or the last child of a flow, so the \(i^{th}\) child of a flow is never the last child. This function is updated to include the par-replicator, which also cannot be the last child of a flow. In case the \(i^{th} + 1\) child of a flow is a loop or a par-replicator then we find its successor and so on. Regarding the loop and the par-rep, we also add another restriction that they cannot be followed by an exception handler combinator (interrupt, retry), therefore the successor of a par-rep and a loop cannot be an exception handler combinator (Figure D.2 in Appendix D).

- **conjunction_of_leaves**: The output of this recursive function is a predicate representing the conjunction of the guards of the leaf events. The output of `conjunction_of_leaves` applied to a leaf with no parameters is `leaf-name = FALSE`. If the leaf has parameters the output is `leaf-name = ∅`. This function is updated to include the par-replicator and the exception handling combinators (interrupt, retry). In the case of a par-replicator, the conjunction is applied to the par-replicator child
Chapter 8 ERS: Formal Specification and Tool Support

and its successor child due to the zero execution case. The conjunction is also applied to the normal-child and the interrupting-child of the exception handling combinators (Figure D.3 in Appendix D).

- **disjunction_of_leaves**: The output of this recursive function is a predicate representing the disjunction of the invariants of the leaf events. The output of disjunction_of_leaves applied to a leaf with no parameters is simply the leaf-name, while in the case the leaf has parameters, the output is leaf-name ≠ ∅. This function is updated similar to the conjunction_of_leaves: to include the par-replicator and the exception handling combinators (interrupt, retry), with the difference of applying disjunction instead of conjunction (Figure D.4 in Appendix D).

- **union_of_leaves**: The output of this recursive function is a predicate representing the union of the control variables of the leaf events. The domain function may be applied to each leaf event, according to the number of times a replicator exists in the traversing steps. This function is also updated to introduce the par-replicator and the exception handling combinators in a way similar to the conjunction_of_leaves, but with applying union instead of conjunction (Figure D.5 in Appendix D).

- **build_seq_grd**: The result of this recursive function is a guard predicate specifying the sequencing between two leaf events, where sequencing depends on the predecessor of a leaf. This function is updated to include the exception handling combinators, where in the case the predecessor is retry, we only take into account the normal-child, whereas in the case the predecessor is an interrupt, we require the execution of either the normal-child or the interrupting-child. We have not included the par-replicator because the par-replicator will be ignored due to the zero execution case as explained by the predecessor function (Figure D.6 in Appendix D).

### 8.3.1 New Auxiliary Functions

In addition to the previously updated auxiliary functions reused from Salehi Fathabadi (2012), we introduce some new auxiliary functions that help us in formalising the newly added translation rules.

These functions take into account the previous restriction that a loop cannot be a first or last child of a flow because its leaf events can have no associated variable names (Salehi Fathabadi, 2012). The par-replicator cannot be the last child of a flow, since it needs an event to stop its execution. In addition to these restrictions, we have also introduced additional restrictions to ERS related to the translation rules and the tool. These restrictions are required to ensure the proper behaviour of all the combinators.
These restrictions are mainly related to the combinators that depend on the follow-on or the successor events, which can be described as follows:

1. The *loop* and the *par-replicator* **cannot** be directly followed by an exception handling combinator (*interrupt*, *retry*).

2. If the *loop* and the *par-replicator* are followed by a flow (i.e., not a leaf), then the first child of the flow must be always a solid child.

3. If the *interrupting-child* of the *interrupt* and the *retry* combinators is a flow (i.e., not a leaf), then the first child of the interrupting flow must be a solid child.

4. If the *normal-child* of a an *interrupt* and *retry* is a flow, then the last child must be a solid child.

The last restriction is explained in Section 6.2.1 of Chapter 6. The simple justification for the rest of the restrictions is to simplify the translation rules. These restrictions can be also justified by the execution of the *loop* and the *par-replicator* which is disabled by its follow-on event ensuring that these combinators do not execute forever, hence we added the restriction that these combinators cannot be directly followed by an exception handling combinator (*interrupt*, *retry*), since the normal flows of these combinators can be interrupted. In addition if the follow-on child of the *loop* and the *par-replicator* is a flow then we will need the first child to disable their execution, in which case having the first child as the solid child makes sense, since the solid child should simulate the main behaviour of its parent. Similarly for the interrupting child of the *interrupt* and the *retry* combinators, their execution will be disabled by the completion of the normal sub-flow, hence having the first child as the solid child to simulate the behaviour of its parent is logical.

In addition, having the first child as the solid child will ensure that the first child cannot be a combinator like the *loop* for example with a zero execution case, and there will be no difference between strong and weak sequencing.

The newly added auxiliary functions takes into consideration the previously mentioned restrictions. Following the approach of Salehi Fathabadi (2012) in defining the auxiliary functions, we show the function definition, the output operation and the traversing steps which show how the function applies pattern-matching to extract the various sub-terms of its source, which could also rely on other functions. In all the auxiliary functions, the letter *f* is an abbreviation of the function name.

The auxiliary functions *list_child_leaves* (Figure 8.2) and *list_first_child_leaves* (Figure 8.3), are used similar to the function *list_of_leaves* to traverse down the last refinement level of the sub-tree. The difference is the output of these functions, where *list_child_leaves* returns every leaf of a flow, while *list_first_child_leaves* only returns the leaves of the
The auxiliary function `build_par_ref_grd` (Figure 8.4) is used in the `par-replicator` translation rules when the `par-replicator` child is a flow and not a simple leaf. This is to ensure that the follow-on child of the `par-replicator` cannot interrupt the `par-replicator` child in the middle of its execution. This function uses another function `par_ref_grd` which is defined in Figure 8.5.

The function `build_succ_SI_inv` (Figure 8.6) is used in building invariants for the `par-replicator` and the `loop` in the single instance case. These invariants are related to the successor of the `loop` and the `par-replicator`, which are needed in the case the child of these combinators are flows and not simple leaves.
8.4 Translation Rules

The original ERS patterns, presented in Chapter 3 are described formally using twenty-three translation rules in Salehi Fathabadi (2012). The rules are summarised in Tables 8.4 and 8.5. The tables present the rule name (first column) and the rule description (second column) presenting the ERS element and its corresponding element in Event-B. The ERS elements described include a solid leaf, which is a leaf connected to its parent.
by a solid line, and a dashed leaf, which is a leaf connected to its parent by a dashed line. A replicator leaf is a leaf connected to a replicator combinator, which can be one of the following (all, some, one). On the other hand, a non-replicator leaf is a leaf that is not connected to a replicator. A solid xor is a xor combinator connected to its parent using a solid line. Similarly, a solid one is a one combinator connected to its parent using a solid line.
Table 8.4 presents the rules related to an ERS leaf; these rules describe how a leaf is translated into variables, events, guards, actions, and invariants in Event-B, while Table 8.5 presents the rules that are specific to the applied combinator like the xor, loop and one, in addition to the weak sequencing flow rules.

<table>
<thead>
<tr>
<th>Rule Name</th>
<th>Rule Signature: ERS $\Rightarrow$ Event-B</th>
</tr>
</thead>
<tbody>
<tr>
<td>TR_leaf1</td>
<td>leaf $\Rightarrow$ variable</td>
</tr>
<tr>
<td>TR_leaf2</td>
<td>non-replicator leaf $\Rightarrow$ typing invariant</td>
</tr>
<tr>
<td>TR_leaf3</td>
<td>replicator leaf $\Rightarrow$ typing invariant</td>
</tr>
<tr>
<td>TR_leaf4</td>
<td>leaf $\Rightarrow$ sequencing invariant</td>
</tr>
<tr>
<td>TR_leaf5</td>
<td>solid leaf $\Rightarrow$ gluing invariant</td>
</tr>
<tr>
<td>TR_leaf6</td>
<td>solid leaf $\Rightarrow$ refining event</td>
</tr>
<tr>
<td>TR_leaf7</td>
<td>dashed leaf $\Rightarrow$ new event</td>
</tr>
<tr>
<td>TR_leaf8</td>
<td>leaf $\Rightarrow$ sequencing guard</td>
</tr>
<tr>
<td>TR_leaf9</td>
<td>non-replicator leaf $\Rightarrow$ guard</td>
</tr>
<tr>
<td>TR_leaf10</td>
<td>replicator leaf $\Rightarrow$ guard</td>
</tr>
<tr>
<td>TR_leaf11</td>
<td>non-replicator leaf $\Rightarrow$ action</td>
</tr>
<tr>
<td>TR_leaf12</td>
<td>replicator leaf $\Rightarrow$ action</td>
</tr>
</tbody>
</table>

Table 8.4: ERS Leaf-Related Translation Rules

These rules have different roles, for example the typing invariant rules will define the type of the control variable, while sequencing invariant rules will define the execution order of the control variable with respect to previously defined control variables, which in most cases are the ones directly to their left in the ERS diagram. The gluing invariant will relate the refining control variable to its parent in the more abstract level. The sequencing guard rules will add a guard to the event to check the ordering of its corresponding control variable. Other guards such as the replicator and non-replicator leaf guards in Table 8.4 will add a guard to the event to check for previous execution of the event, while the action rules will add an action to the event to record its execution. Other guards and invariants in Table 8.5 are specific for the combinator.

<table>
<thead>
<tr>
<th>Rule Name</th>
<th>Rule Signature: ERS $\Rightarrow$ Event-B</th>
</tr>
</thead>
<tbody>
<tr>
<td>TR_xor1</td>
<td>solid xor $\Rightarrow$ gluing invariant</td>
</tr>
<tr>
<td>TR_xor2</td>
<td>xor $\Rightarrow$ invariant</td>
</tr>
<tr>
<td>TR_xor3</td>
<td>xor $\Rightarrow$ guards</td>
</tr>
<tr>
<td>TR_one1</td>
<td>solid one $\Rightarrow$ gluing invariant</td>
</tr>
<tr>
<td>TR_one2</td>
<td>one $\Rightarrow$ invariant(s)</td>
</tr>
<tr>
<td>TR_one3</td>
<td>one $\Rightarrow$ guard(s)</td>
</tr>
<tr>
<td>TR_loop1</td>
<td>loop $\Rightarrow$ guard(s)</td>
</tr>
<tr>
<td>TR_loop2</td>
<td>loop $\Rightarrow$ loop successor event(s) guard(s)</td>
</tr>
<tr>
<td>TR_loop3</td>
<td>loop $\Rightarrow$ resetting event</td>
</tr>
<tr>
<td>TR_weak1</td>
<td>weak sequencing flow $\Rightarrow$ invariant(s)</td>
</tr>
<tr>
<td>TR_weak2</td>
<td>weak sequencing flow $\Rightarrow$ guard(s)</td>
</tr>
</tbody>
</table>

Table 8.5: ERS Combinator-Related Translation Rules
In the following sub-sections we present the translation rules related to the extended ERS approach as presented in chapters 4 and 6. In defining the translation rules we follow the same tabular structure used in defining the original ERS rules. The structure is presented in Figure 8.7, where the first row represents rule signature which is of the form:

Rule Name: ABNF Element(s) (Source) → Event-B Element(s) (Target)

The second row, represents the ABNF element which is the source element. The third row uses some of the auxiliary functions defined in Section 8.3 to help in specifying the Event-B target element. The auxiliary functions will in turn apply pattern matching to extract the various sub-terms of the translation source, as shown in Section 8.3.1. The last row defines the target element in Event-B, i.e., the output. The last row can sometimes be divided into two rows to differentiate between the single instance case and the multiple instances case, if they are different.

An example ERS diagram is presented in Figure 8.1, which we will use to show how the new translation rules of the extended ERS are applied. The ERS diagram Figure 8.1 consists of an abstract level and two refinements, where the green nodes represent leaf events that are not decomposed.

8.4.1 Parameter Extension Translation Rules

In addition to changing the syntax of a parameter to include a range expression as shown in Table 8.2, we introduce two new translation rules related to adding the
range_expression property. The first is TR_rangeExpression which adds a guard to a replicator leaf. The second is TR_rep which maps a replicator to (an) invariant(s).

### 8.4.1.1 TR_rangeExpression: Mapping a replicator to (a) guard(s)

TR_rangeExpression presented in Figure 8.8, adds a guard (grd_Exp) to the first child leaves of a replicator (all, some, one, par-rep). This guard checks that the replicator parameter \( p_i \) remains in the range of the expression \( \text{range}_exp \), if it exists.

![Figure 8.8: TR_rangeExpression: Mapping a replicator to (a) guard(s)](image)

The rule applies the list_first_child_leaves function (Figure 8.3) to get the list of leaves of the first child of the replicator. The output of the rule adds a guard to each leaf event resulting from the list_first_child_leaves function. The added guard checks that the replicator parameter is an element of the range expression of the applied replicator. This rule is only applied when the replicator parameter has a defined range expression.

In Figure 8.9, we apply the rule TR_rangeExpression, to both the par-rep and the all at the second refinement level of Figure 8.1. In this case the first child of both replicators is the leaf \( K \), therefore two guards will be added to the event \( K \). The first guard is to ensure the par-replicator parameter \( p_1 \) is in the range expression \( P1 \), and the second is to ensure that the all replicator parameter \( p_2 \) is in the range expression \( P2 \). The guards are the same in both the single and the multiple instances cases.

### 8.4.1.2 TR_rep: Mapping a replicator leaf to (an) invariant(s)

The translation rule TR_rep in Figure 8.10, maps a replicator to one or more invariant(s) if the replicator defines a range expression. The replicator invariant ensures that the range of the replicator leaf variable remains within the range expression.
**Application of TR\_rangeExpression**

```latex
\begin{align*}
\text{flow}(\text{process\_name}, p, 1) \\
&\quad (\text{flow}(A, p, 1)) \\
&\quad (\text{leaf}(G)(0)) \\
&\quad \text{loop}(\text{flow}(H, p, 1)) \\
&\quad \quad (\text{and}(\text{leaf}(P), \text{leaf}(Q))(0), \\
&\quad \quad \text{leaf}(R)(1), \\
&\quad \quad \text{leaf}(I)(0)), \\
&\quad \text{leaf}(J)(1)) \\
&\quad \text{par-rep}(p_1)(\text{flow}(B, (p, p_1), 1)) \\
&\quad \quad (\text{all}(p_2)(\text{leaf}(K))(0), \\
&\quad \quad \text{leaf}(L)(1))(0), \\
&\quad \text{leaf}(C)(0), \\
&\quad \text{interrupt}(\text{flow}(D, p, 1)) \\
&\quad \quad (\text{retry}(\text{flow}(M, p, 1))(\text{xor}(\text{leaf}(S), \text{leaf}(T))(0), \text{leaf}(U)(1))) \\
&\quad \quad \quad \text{flow}(N, p, 1)(\text{leaf}(V)(1), \text{leaf}(W)(0))(0), \\
&\quad \quad \quad \text{leaf}(O)(1)) \\
&\quad \quad (\text{leaf}(E))(0), \\
&\quad \quad \text{leaf}(F)(0))
\end{align*}
```

**Applied Auxiliary Functions:**

1. **par-rep Case:**
   - `list\_first\_child\_leaves` (flow(B, (p, p1), 1), (p, p1)) = leaf(K, (p, p1, p2))

2. **all Case:**
   - `list\_first\_child\_leaves` (leaf(K), (p, p1, p2)) = leaf(K, (p, p1, p2))

**Output:**

```
event K
\begin{align*}
\text{any} & \quad p \quad p_1 \quad p_2 \quad \text{where} \\
& \quad \text{\texttt{@grd\_Exp}}\text{1} \quad p_1 \in P_1 \\
& \quad \text{\texttt{@grd\_Exp}}\text{2} \quad p_2 \in P_2
\end{align*}
```

Figure 8.9: Application of TR\_rangeExpression to the par-rep and all

**TR\_rep** adds an invariant for the leaf variables of the first child of the replicator. In the single instance case, where the replicator leaves resulting from `list\_first\_child\_leaves` function, have only the replicator parameter \((p_i)\), the invariant is simply that the leaf control variable is subset or equal to the range expression of the replicator \((\text{range\_exp})\).

In the multiple instances case where the leaf replicator leaves inherit the parent flow parameters \((p_1...p_n)\), the invariant specifies that each parameter is an element of its range expression if defined or its type otherwise, and finally it ensures that the leaf variable of the parent flow parameter maplets \((p_1 \mapsto .. \mapsto p_n)\) is subset or equal to the replicator range expression.

In Figure 8.11 we apply **TR\_rep** to the **all** replicator in the second refinement level showing the MI case, we also apply **TR\_rep** to the **par-replicator** but in the first refinement
Figure 8.10: TR_rep: Mapping a replicator leaf to an invariant

level where we assume the ERS diagram does not introduce the parameter \( p \) at the root of the diagram to show the result of applying this rule in the single instance case.

### 8.4.2 Par-Replicator Translation Rules

The `par-replicator` combinator is one of the replicator patterns as shown in the ERS language syntax (Table 8.2). Extending the ERS combinators with the `par-replicator`, results in a set of minor changes and a set of major changes. Minor changes include the modification of few translation rules (`TR_leaf3`, `TR_leaf10`, `TR_leaf12`) and some auxiliary functions. Here we only describe the major changes, since the minor ones only include the case of adding the `par-replicator` and no major changes take place.

The major changes include the introduction of three new translation rules: `TR_par1`, `TR_par2` and `TR_par3`. `TR_par1` maps a `par-replicator` to a guard that stops the execution of the `par-replicator` child once the next child of the `par-replicator` executes. `TR_par2` maps a refined `par-replicator` to a guard in its next child, to ensure that the
next child does not stop the *par-replicator* sub-events in the middle of their execution. *TR_par3* maps a refined *par-replicator* child to (an) invariant(s) to enforce the guard(s) generated by *TR_par2*.

### 8.4.2.1 TR_par1: Mapping a *par-replicator* to (a) guard(s)

The child of a *par-replicator* can execute zero or more times until the *par-replicator* successor child executes. In Event-B, this can be represented by adding a guard to the *par-replicator* child to check that the successor or the next child has not executed yet. Figure 8.12 presents the translation rule *TR_par1* which adds (a) guard(s) to the *par-replicator* child.
### TR\_par1: Mapping a par-rep to (a) guard(s)

<table>
<thead>
<tr>
<th><strong>TR_par1</strong>: par-rep</th>
<th>1*guard</th>
</tr>
</thead>
</table>
| flow(parent-name, (p_1,...,p_n), sw)(..., par-rep (p_i , par-rep-child)(0),...)
| list\_first\_child\_leaves(par-rep-child, (p_1,...,p_n, p_{i-1},..., p_{i+n}))* =
| leaf_i (leaf-name_{i-1}, 1*par_{i-1}) ... leaf_m (leaf-name_{i+n}, 1*par_{i+n}) (k \geq 0 & 1 \leq j \leq m)
| successor(par-rep, n) = (child, parnum)

* Where the first child of a flow is always a solid line, so the first child can be either a leaf, xor, or one.

<table>
<thead>
<tr>
<th>SI case (n = 0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>event leaf-name_i</td>
</tr>
<tr>
<td>@grd_par conjunction_of_leaves (child, 0)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MI case (n &gt; 0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>event leaf-name_i</td>
</tr>
<tr>
<td>any p_1...p_n where</td>
</tr>
<tr>
<td>@grd_par p_1\text{_}\text{_}..._p_n \in union_of_leaves (child, 0)</td>
</tr>
</tbody>
</table>

Figure 8.12: TR\_par1: Mapping a par-rep to (a) guard(s)

\textit{TR\_par1} uses the auxiliary functions, \textit{list\_first\_child\_leaves} and \textit{successor} in the traversing steps, where a guard is added to each leaf event resulting from \textit{list\_first\_child\_leaves} function. The guard is generated by applying the \textit{conjunction\_of\_leaves} in the single instance case and \textit{union\_of\_leaves} in the multiple instances case to the result of the \textit{successor} function. Note that we use zero for the parameter number when applying both the \textit{conjunction\_of\_leaves} and \textit{union\_of\_leaves}.

In Figure 8.13, we apply \textit{TR\_par1} to the par-replicator in the second refinement level of 8.1, which results in adding a guard, \textit{grd\_par}, to the leaf event \textit{K}.

### 8.4.2.2 TR\_par2: Mapping a refined par-rep to (a) guard(s) in the successor event(s)

When the child of a par-replicator is not a leaf, \textit{TR\_par2} will be applied. This rule will add a guard to the par-replicator successor first child leaves to disable them from interrupting the execution of the par-replicator sub-events if required. We do this by checking that both the first child and the last child of the par-replicator have executed. Figure 8.14 presents the translation rule \textit{TR\_par2} which requires introducing two new recursive functions \textit{build\_par\_ref\_grd} in Figure 8.4 and its inner function \textit{par\_ref\_grd} in Figure 8.5.

When the child of a par-replicator is not a leaf, \textit{TR\_par2} will execute by first getting the successor child of the par-replicator. Then, we apply \textit{list\_first\_child\_leaves} function
to the *par-replicator* successor child, to get the leaves of its first child which must be always a solid line. The output of this translation rule will be the addition of a guard `grd_par` to the leaf events, if the result of `build_par_ref_grd` is not an empty string.

*TR_par2* does not always result in adding a guard to the successor child leaves, that is why we say "*guard*" which means zero or more times. The cases that do not always require adding a guard to the *par-replicator* successor child is when refining the *par-replicator* child to only one solid leaf, or one solid xor or a one solid one. In these cases the successor of the *par-replicator* cannot interrupt them, that is why there is no need to add a guard.
Figure 8.14: TR_par2: Mapping a refined par-rep to (a) guard(s) in the successor event(s)

In Figure 8.15, we show how we applied TR_par2, which resulted in the addition of a guard (grd_par) to the successor leaf $C$ of the par-replicator.

8.4.2.3 TR_par3: Mapping a refined par-rep to an invariant

TR_par3 generates an invariant related to TR_par2. This invariant will enforce the guard generated by TR_par2, by ensuring that if the leaf event of the successor child has executed, then the refined par-replicator must have executed for the same number of instances for both its first and last child events.

In this translation rule we use the auxiliary function build_succ_SI_inv of Figure 8.6 to build the first part of the invariant in the single instance case, while in the multiple instances case, we use the union_of_leaves function. The second part of the invariant is similar to TR_par2, where we use build_par_ref_grd function.

In Figure 8.17, we apply TR_par3, which results in the addition of the invariant inv_par. This invariant will be only added if the result of applying build_par_ref_grd to the par-replicator child is not an empty string.

8.4.3 Loop Generalisation Translation Rules

Loop generalisation allows a loop to be applied to more than one child executed non-deterministically. Each loop child is independent of the other, so the loop translation rules (TR_loop1, TR_loop2, TR_loop3) are updated to include more than one child of
Figure 8.15: Application of TR_par2

<table>
<thead>
<tr>
<th>Application of TR_par2</th>
</tr>
</thead>
<tbody>
<tr>
<td>flow(process_name, p, 1)</td>
</tr>
<tr>
<td>(flow(A, p, 1)</td>
</tr>
<tr>
<td>(leaf(G))(0),</td>
</tr>
<tr>
<td>loop(flow(H, p, 1)</td>
</tr>
<tr>
<td>(and(leaf(P), leaf(Q))(0),</td>
</tr>
<tr>
<td>leaf(R)(1),</td>
</tr>
<tr>
<td>leaf(I)(0)),</td>
</tr>
<tr>
<td>leaf(J)(1)),</td>
</tr>
<tr>
<td>par-rep(p1)(flow(B, (p, p1), 1)</td>
</tr>
<tr>
<td>(all(p2)(leaf(K))(0),</td>
</tr>
<tr>
<td>leaf(L)(1))(0),</td>
</tr>
<tr>
<td>leaf(C)(0),</td>
</tr>
<tr>
<td>interrupt(flow(D, p, 1)</td>
</tr>
<tr>
<td>(retry(flow(M, p, 1)(xor(leaf(S), leaf(T))(0), leaf(U)(1)))</td>
</tr>
<tr>
<td>(flow(N, p, 1)(leaf(V)(1), leaf(W)(0))(0),</td>
</tr>
<tr>
<td>leaf(O)(1)))</td>
</tr>
<tr>
<td>(leaf(E))(0),</td>
</tr>
<tr>
<td>leaf(F)(0))</td>
</tr>
</tbody>
</table>

Applied Auxiliary Functions:

- successor(par-rep(p1, flow(B, (p, p1), 1)), 2) = leaf(C, 1)
- list_first_child_leaves (leaf(C), p) = leaf(C, p)

Output:

```
@event C
any p where
@grd_par dom(K) = L
```

Figure 8.16: TR_par3: Mapping a refined par-rep to an invariant

<table>
<thead>
<tr>
<th>TR_par3: par-rep --------&gt; invariant</th>
</tr>
</thead>
<tbody>
<tr>
<td>flow(parent-name, (p1,...,p_n), sw)(..., par-rep(p_i, par-child)(0),...)</td>
</tr>
<tr>
<td>*where par-child ≠ leaf and build_par_ref_grd (par-child) ≠ &quot;&quot;</td>
</tr>
</tbody>
</table>

**SI case (n = 0)**

**invariants**

```
@inv_par build_succ_SI_inv (child, 0) ⇒ build_par_ref_grd (par-child)
```

**MI case (n = 0)**

**invariants**

```
@inv_par ∀ p_i,..., p_n. p_i→...→p_n ∈ union_of_leaves (child, 0) ⇒ build_par_ref_grd (par-child)
```
**Application of TR_par3**

```
flow(process_name, p, 1)
(flow(A, p, 1)
  (leaf(G)(0),
  loop(flow(H, p, 1)
    (and(leaf(P), leaf(Q))(0),
     leaf(R)(1)),
    leaf(I))(0)),
  leaf(J)(1)),
par-rep(p1)(flow(B, (p, p1), 1)
  (all(p2)(leaf(K))(0),
   leaf(L)(1))(0),
  leaf(C)(0),
  interrupt(flow(D, p, 1)
    (retry(flow(M, p, 1)(xor(leaf(S), leaf(T))(0), leaf(U)(1))
     (flow(N, p, 1)(leaf(V)(1), leaf(W)(0))(0),
      leaf(O)(1))
    (leaf(E))(0),
    leaf(F)(0))
```

**Applied Auxiliary Functions:**

- **successor** \(\text{par-rep}(p1, \text{flow}(B, (p, p1), 1))\) = \(\text{leaf}(C, 1)\)
- **build_par_ref_gdrd** \(\text{flow}(B, (p, p1), 1))\) = \(\text{dom}(K) = L\)

**Output:**

```
invariants
@inv_par \forall p. p ∈ C \Rightarrow \text{dom}(K) = L
```

**Output in SI:** where we assume `Process_Name` does not introduce parameter \(p\)

```
invariants
@inv_par C = TRUE \Rightarrow \text{dom}(K) = L
```

Figure 8.17: Application of TR_par3

the loop. We also add a new translation rule \(TR_{loop4}\) which generates (an) invariant enforcing the guard(s) generated by \(TR_{loop2}\).

Some of the possible event traces of the loop in Figure 8.1 at the second refinement level are:

\[
< G(p), J(p), ... > \\
< G(p), P(p), Q(p), R(p), J(p), ... > \\
< G(p), I(p), J(p), ... > \\
< G(p), I(p), I(p), P(p), I(p), Q(p), R(p), J(p), ... > \\
< G(p), P(p), Q(p), R(p), P(p), Q(p), I(p), J(p), R(p), I(p), J(p), ... >
\]
8.4.3.1 TR_loop1: Mapping a loop to (a) guard(s)

TR_loop1, like TR_par1, adds a guard to the loop children to ensure that the loop next child has not executed yet. TR_loop1 is applied every time a loop exists and results in adding one or more guards. A loop can never be a refining event due to the single solid line rule, so the property ref is always zero. Figure 8.18 updates the original rule by allowing more than one child to a loop. This rule applies the list_first_child_leaves to each child of the loop, to get the leaf events of the first child of each loop. Then, the loop guard (grd_loop) is added to each leaf event resulting from the function list_first_child_leaves.

The guard grd_loop is defined in the same manner for each leaf event, first finding the next child of the loop by applying the successor function to the loop. In a single instance case, where the parent of the loop has no parameters (n=0), apply the conjunction_of_leaves function (Figure D.3) to the successor child. Otherwise, for a multiple instances case, apply the union_of_leaves function (Figure D.5) to the successor child and check that the maplet of the loop’s parent parameters (p_1 ↦ .. ↦ p_n) are not in the result of the union_of_leaves function.

Figure 8.19 shows an application of TR_loop1 in the second refinement level of Figure 8.1, resulting in adding the guard (grd_loop) to each first child leaf events, checking that the successor event J has not executed.

8.4.3.2 TR_loop2: Mapping a loop to (a) guard(s) in the successor child

TR_loop2 is applied when at least one of the loop children is not a leaf due to refinement. The rule ensures that the next child of the loop does not interrupt the execution of the
refined loop children.  TR_loop2 is updated as shown in Figure 8.20 to include the case of more than one loop child. The output of applying TR_loop2 is one or more guards added to the next leaf event(s) of the loop.

When applying TR_loop2, we find the next loop child by applying the successor function. Then list_first_child_leaves function is applied to the successor child of the loop, to get its leaf events. The guard grd_loop will then be added to each leaf event.

In a single instance case, if more than one child of the loop is not just a single leaf, then the loop guard (loop_grd) is the result of applying the conjunction_of_leaves function to the non-leaf child of the loop. In a multiple instances case, it is the result of applying
8.4.3.3 TR\textsubscript{loop3}: Mapping a loop to (a) resetting event(s)

Similar to TR\textsubscript{loop2}, TR\textsubscript{loop3} is applied when at least one of the loop children is not a single leaf. Refining a loop child results in adding control variables to ensure the proper sequencing, so TR\textsubscript{loop3} adds a resetting event for each child that is not a single leaf to enable the multiple execution of the loop child. The updated version of this rule is shown in Figure 8.22. The only change in the rule is, the possibility of generating more than one resetting event depending on the number of the loop children that are not single leaves.

The resetting event of the refined loop child is executed when its last leaf/leaves completes execution, this is ensured by the sequencing guard of the corresponding reset event. The sequencing guard of the reset event is the output of applying build\_seq\_grd function (Figure D.6) where the predecessor (Figure D.1) is the corresponding refined loop child.

The purpose of the resetting event is to allow multiple executions of the loop child, and this is ensured by resetting the control variables of the loop events. So, the action of

---

### Figure 8.20: TR\textsubscript{loop2}: Mapping a loop to (a) guard(s) in the successor child

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TR\textsubscript{loop2}</td>
<td>loop \rightarrow 1*guard</td>
</tr>
<tr>
<td>flow</td>
<td>(parent-name, (p\textsubscript{1}, ..., p\textsubscript{n}), sw)(..., loop (child\textsubscript{i}, ..., child\textsubscript{n})</td>
</tr>
<tr>
<td>*where</td>
<td>at least one (child, \neq leaf) (1 \leq i \leq m)</td>
</tr>
<tr>
<td>successor</td>
<td>(loop, n) = (child, parnum)</td>
</tr>
<tr>
<td>list_first_child_leaves</td>
<td>(child, (p\textsubscript{1}, ..., p\textsubscript{n})) =</td>
</tr>
<tr>
<td>leaf\textsubscript{j}</td>
<td>(leaf-name\textsubscript{j}, \ast \text{par}\textsubscript{j}), ..., leaf\textsubscript{k} (leaf-name\textsubscript{k}, \ast \text{par}\textsubscript{k}) (1 \leq j \leq k)</td>
</tr>
</tbody>
</table>

**SI case (n = 0)**

**event** leaf-name\textsubscript{j} | @grd\textsubscript{loop} conjunct\_of\_leaves (child\textsubscript{j}, 0) |

**MI case (n > 0)**

**event** leaf-name\textsubscript{j} | any p\textsubscript{1}...p\textsubscript{n} where |
| | @grd\textsubscript{loop} p\textsubscript{1}...p\textsubscript{n} \in union\_of\_leaves (child\textsubscript{j}, 0) |

---

The union\_of\_leaves function to the non-leaf child of the loop. The same will be repeated to each non-leaf child of the loop.

Applying TR\textsubscript{loop2} to the second refinement level of Figure 8.1 is shown in Figure 8.21. The loop child (H) is not a single leaf, therefore the rule TR\textsubscript{loop2} can be applied, resulting in the addition of the guard (grd\textsubscript{loop}) to the successor event J.
the reset event in a single instance case, will set each leaf of the refined child to false. This is the case when no new parameters are added during refinement \((ni = 0)\). In case a parameter is added during refinement as a result of applying a replicator \((ni \neq 0)\), it sets each leaf of the refined child to the empty set.

In a multiple instances case, in case the parameter list is the same as the loop \((ni = n)\), the resetting event removes the maplet of the loop parent parameters \((p_1 \mapsto \ldots \mapsto p_n)\) from each leaf. Otherwise, if the parameter list is longer \((ni > n)\), due to applying a replicator during refinement, the resetting is done using the domain subtraction of the loop parameters from each leaf control variable.
**Figure 8.22:** TR_loop3: Mapping a loop to (a) resetting event(s)

<table>
<thead>
<tr>
<th>TR_loop3: loop</th>
<th>1* resetting event(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>flow(parent-name, (p₁,..., pₙ), sw)(..., loop (child₁,..., childₖ)(0), ...)</td>
<td>*where (childⱼ ≠ leaf) (1 ≤ j ≤ k)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>list_child_leaves</th>
<th>(childᵢ, (p₁,..., pₙ))</th>
</tr>
</thead>
<tbody>
<tr>
<td>leaf₁(name₁₁, (pᵢ₁,..., pᵢ₁ₙ)), ..., leafᵢₘ(nameᵢₘ, (pᵢₘ₁,..., pᵢₘₙ))</td>
<td>(1 ≤ i ≤ m)</td>
</tr>
</tbody>
</table>

In Figure 8.23, the loop child (H) is not a single leaf, in this case applying TR_loop3 at the second refinement level, will result in the resetting event reset_loop.

### 8.4.3.4 TR_loop4: Mapping a loop to (an) invariant(s)

TR_loop4, which is presented in Figure 8.24, is a new translation rule that is added to enforce the guard(s) resulting from TR_loop2 using invariant(s). This rule, similar to TR_loop2 only applies to the refined child of the loop.

This rule will generate (an) invariant(s) that ensures that the loop successor can only occur if the first child of the refined loop has not occurred, i.e., the refined child of the loop must be reset first to ensure that its execution is not interrupted.

The application of TR_loop3 is shown in Figure 8.25, which results in the addition of the invariant (inv_loop).
8.4.4 Translation Rules for Exception Handling Combinators

As explained earlier the child nodes of the exception handling combinators, unlike the other combinators, are not symmetrical, because the interrupting child can block the normal child execution at any time before completion.
The new translation rules *TR_interrupt*, *TR_disable_interrupt* and *TR_interrupt_inv* apply to both the interrupt and retry combinators, whereas *TR_retry* and *TR_retry_reset* only apply to the retry combinator.

### 8.4.4.1 TR_interrupt: Mapping the Interrupt and Retry Combinators to Disabling Guard(s) in the Normal Flow

Both the *Interrupt* and the *Retry* combinators will add a disabling guard to each leaf event of the normal flow. This guard will disable the execution of the normal flow, once the interrupting event executes.

Figure 8.26 presents the rule to get the Event-B output, which is in this case one or more guards. The rule will be applied to any interrupt or retry combinator, which will always has the refining property set to zero. The traversing steps of the rule requires getting every leaf event of the normal sub-flow of the exception handling combinator, which is done by applying the *list_child_leaves* function (Figure 8.2) to the *normal-child*. We also need the first child of the interrupting sub-flow.

The output will be a guard (*grd_interrupt*) added to each leaf event of the normal sub-flow, which is computed by applying the *conjunction_of_leaves* function to the first child (*child*$_1$) of the interrupting sub-flow in the single instance case. In the multiple instances case the guard is the flow parameters not in the *union_of_leaves* of the first child of the interrupting sub-flow.

In Figure 8.27, we show the application of *TR_interrupt* to both the interrupt and retry combinators at the second refinement level of Figure 8.1. The *interrupt-combinator* is applied at a higher level of the *retry-combinator*, that is why we can see the interrupting
leaves of the \textit{retry-combinator} ($V$ and $W$) have the interrupting guard of the \textit{interrupt-combinator}.

\subsection{TR\_disable\_interrupt: Mapping the Interrupt and Retry Combinators to a Disabling Guard(s) in the Interrupting Event}

This is to define the scope of the interrupting sub-flow of both the \textit{interrupt} and \textit{retry} combinators, where the interrupting sub-flow will not be applicable when the normal sub-flow completes successfully. As explained earlier in the refinement of \textit{interrupt} and \textit{retry} in sections 6.2.1 and 6.2.2, the refining child of the normal sub-flow must be always the last child of the normal sub-flow, while the solid child of the interrupting sub-flow must be the first child.
Chapter 8 ERS: Formal Specification and Tool Support

8.4.4.3 \( TR_{interrupt} \text{ inv} \): Mapping the Interrupt and Retry Combinators to an Invariant

The output of this rule is an invariant, which ensures that the last child of the normal flow and the first child of interrupting flow are mutually exclusive. The guards generated by \( TR_{interrupt} \) and \( TR_{disable\_interrupt} \) translation rules, when applied to the last child of the normal flow and the first child of the interrupting flow respectively, ensures the mutual exclusive property.

Figure 8.30 presents the translation rule, where in the single instance case, we apply the disjunction_of_leaves function (Figure D.4), while in the multiple instances case we apply the union_of_leaves function. The xor operator in the single instance invariant is not implemented yet in the Event-B language, we use it here to show the need of a
Figure 8.27: Application of TR\_interrupt to the interrupt-combinator

mutual exclusive operator in the single instance case. While the partition operator in Event-B can describe the mutual exclusive relationship in the multiple instances case.
**Figure 8.28: TR_disable_interrupt: Mapping the interrupt/retry to Guard(s) in the Interrupting Flow**

The application of TR_disable_interrupt is done in Figure 8.31, resulting in interrupt1_inv invariant for the interrupt-combinator, and interrupt2_inv invariant for the retry-combinator.

### 8.4.4.4 TR_retry: Mapping the Retry Combinator to Resetting Actions

The translation rule, TR_retry (Figure 8.32), will reset the control variables for each leaf event in the normal sub-flow of the retry combinator, this is to enable the execution of the normal sub-flow again.

The resetting actions of the control variables of the normal sub-flow will be added to the first child leaf events of the interrupting sub-flow. Similar to the resetting actions in the loop resetting event, in the single instance case where no parameters are introduced due to a replicator, the resetting action will set the leaf control variable to false, however if there are parameters added due to an application of a replicator, the leaf control variable will be reset to the empty set. In the multiple instances case, if the number of parameters of the leaf control variable is the same as its parent flow, then we just remove the flow parameters from the control variable, whereas if its parameters are more due to an application of a replicator, then the parent parameters will be removed from the domain of the leaf control variable.

In Figure 8.33, we apply TR_retry to the retry-combinator at the second refinement level. In this case the number of parameters is the same as the parent flow, because we have not introduced a replicator for any leaf event of the normal-child.
Figure 8.29: Application of TR\_disable\_interrupt

8.4.4.5 TR\_retry\_reset: Mapping the Retry Combinator to a Reset Event

In the TR\_retry\_reset translation rule (Figure 8.34), the retry-combinator will be mapped to an event, that resets the control variables of the interrupting child, after it completes execution. This is to enable the normal sub-flow to execute again. As explained in Chapter 6, the normal-child of the retry-combinator can only complete execution once, while the interrupting-child can complete execution more than once.
**Figure 8.30: TR_interrupt_inv: Mapping the interrupt/retry to an Invariant**

The sequencing guard of this resetting event, will be after the interrupting sub-flow completes execution, that is why we use the `build_seq grd` function where the predecessor child is the `interrupting-child` of the `retry-combinator`. The resetting actions of this event are done in the same way as in `TR_retry`, but this time the resetting is done for the `interrupting-child` control variables.

In Figure 8.35, we apply `TR_retry_reset`, where a new resetting event (`reset_retry`) is generated with the appropriate sequencing guard and resetting actions.

### 8.5 ERS Tool Support

The openness of the Event-B tool, Rodin (Section 3.3.4), allows us to support the ERS approach by a plug-in that provides an automatic generation of part of the Event-B model related to the ordering of events and their relationships at different refinement levels.

The Eclipse Modelling Framework (EMF) (Steinberg et al., 2008) provides a modelling and code generation facility based on structured data models, where an emf meta-model can be used to define the different entities, attributes and relations between the entities of a model. The ERS language, introduced in Section 8.2, is defined using the Eclipse Modelling Framework (EMF) meta-model, and then transformed into an Event-B EMF meta-model.

In the earlier version of the tool (Fathabadi et al., 2014), an ERS diagram was defined in an EMF tree structure. The transformation from the ERS language to Event-B was performed using the Epsilon Transformation Language (ETL) (Kolovos et al., 2014).
In the updated version of the ERS plug-in (Dghaym et al., 2016), we provide a graphical environment for ERS, and we apply a different approach to transform from the ERS language to Event-B. The new followed approach is based on the generic Diagram Extensions framework for Event-B (Savicks and Snook, 2012). The framework used, is built specifically to support Event-B providing lots of helpful functionalities. In addition to supporting model transformation to Event-B, the generic Diagram Extensions framework provides graphical and validation support.

Figure 8.31: Application of TR_interrupt_inv
Most of the new ERS translation rules described in this chapter and the updates to the original translation rules to include the ERS extensions, are added to the improved ERS plug-in. Generating the Event-B elements from the ERS diagram is based on the generator framework which is part of the generic Diagram Extension framework (Savicks and Snook, 2012; Wiki.event-b.org, 2015). Each rule transforming an ERS element to an Event-B element implements the IRule interface of the generator framework and defines the methods enabled, dependenciesOK, and fire. The enabled method checks when the translation rule should be applied, e.g., the rule transforming a leaf to a variable is enabled if the ERS source element is a leaf that is not a child of a loop. The dependenciesOK method checks if there are any dependencies required like other elements that need to be generated first, e.g., generating a sequencing guard to an event requires the event to be generated first or already existing in the machine, so if dependencies are not satisfied firing the rule will be postponed until all dependencies are satisfied. The fire method is the main method where the mapping of ERS elements to Event-B elements takes place. The fire method returns a list of GenerationDescriptors describing what should be generated, e.g., an ERS leaf is mapped to an event in Event-B.

In figures 8.36 and 8.37, we show how the translation rule TR\_retry (Section 8.4.2.1) is implemented. The enabled method (Figure 8.36) checks that the rule can be only enabled when the source is a leaf with a par-replicator ancestor. The dependenciesOK
checks that all the required elements are satisfied, in this case the leaf event to which the guard will be added should be already generated before firing the rule, otherwise the application of the rule will be postponed.

Figure 8.37 is the main part of the rule where the mapping between the ERS elements and the Event-B elements takes place. The function `getAncestorsAncestorsOfClass`, which is
used in both the enabled and the fire methods, traverses down the tree to get a list of the first child elements of the specified class. In this case getAncestorsAncestorsOfClass has the Par class which represents the par-replicator as an argument, and we also check that the source is a leaf that is not decomposed, therefore, this will play the role of get_first_child_leaves which is used in Figure 8.12. The index, used in naming the par-replicator guard, is needed in the case more than one par-replicator is in the leaf parent chain. The fire method returns a generation descriptor, which will be used by the Generator Framework to generate the par-replicator guard(s).

All the generated Event-B elements from the ERS diagram are read only so that the modeller cannot change the control flow described by the ERS diagram, except the events in order to allow the user to add data related guards and actions.

8.5.1 User Interface

Creating an ERS Diagram in the ERS plug-in requires an existing Event-B machine, where the user can right click and add an ERS flow-diagram. The graphical environment of the ERS plug-in (Figure 8.38) consists of a diagram editor where we can add the ERS elements, a palette which contains the different ERS elements, which are the different

---

**Figure 8.34: TR_retry-reset:** Mapping the retry Combinator to a Resetting Event to Reset the Interrupting Flow

---

<table>
<thead>
<tr>
<th>TR_retry_reset: retry → 1 resetting event</th>
</tr>
</thead>
<tbody>
<tr>
<td>flow(parent-name, (p₁,…, pₙ), sw)((…, retry (normal-child)(interrupting-child)(0),…))</td>
</tr>
<tr>
<td>list_child_leaves (interrupting-child, (p₂, …, pₙ)) =</td>
</tr>
<tr>
<td>leaf₁(name₁, *par₁), …, leafₘ(nameₘ, *parₘ)</td>
</tr>
<tr>
<td>SI case (n = 0) (1 ≤ i ≤ m)</td>
</tr>
<tr>
<td>event reset_retry</td>
</tr>
<tr>
<td>where</td>
</tr>
<tr>
<td>@grd_reset build_seq_grd (interrupting-child, null, null, null)</td>
</tr>
<tr>
<td>then</td>
</tr>
<tr>
<td>@actᵢ_reset [ni = 0: nameᵢ := FALSE</td>
</tr>
<tr>
<td>ni ≠ 0: nameᵢ := ∅</td>
</tr>
<tr>
<td>MI case (n &gt; 0) (1 ≤ i ≤ m)</td>
</tr>
<tr>
<td>event reset_retry</td>
</tr>
<tr>
<td>any p₁…pₙ where</td>
</tr>
<tr>
<td>@grd_reset build_seq_grd (interrupting-child, (p₁, …, pₙ), null, (p₁, …, pₙ))</td>
</tr>
<tr>
<td>then</td>
</tr>
<tr>
<td>@actᵢ_reset [ni = n: nameᵢ := nameᵢ \ {p₁, …, pₙ}</td>
</tr>
<tr>
<td>ni &gt; n: nameᵢ := {p₁, …, pₙ} \ nameᵢ</td>
</tr>
</tbody>
</table>

**Application of TR_retry_reset**

```
flow(process_name, p, 1)
  (flow(A, p, 1)
    (leaf(G)(0),
     loop(flow(H, p, 1)
       (and(leaf(P), leaf(Q))(0),
        leaf(R)(1)),
       leaf(I)(0)),
     leaf(J)(1)),
    par-rep(p1)(flow(B, (p, p1), 1)
      (all(p2)(leaf(K))(0),
       leaf(L)(1)))(0),
  leaf(C)(0),
interrupt(flow(D, p, 1)
  (retry(flow(M, p, 1)(xor(leaf(S), leaf(T))(0), leaf(U)(1))
    (flow(N, p, 1)(leaf(V)(1), leaf(W)(0)))(0),
    leaf(O)(1))
  (leaf(E))(0),
  leaf(F)))(0)
```

**Applied Auxiliary Functions:**

- `list_child_leaves(flow(N, p, 1), p) = leaf(V, p), leaf(W, p)`

**Output:**

```
event reset_retry any p where
  @grd_reset p ∈ W
then
  @act1_reset V := V\{p}
  @act2_reset W := W\{p}
```

**Output in SI:** where we assume `Process_Name` does not introduce parameter `p`

```
event reset_retry
  where
    @grd_reset W = TRUE
then
  @act1_reset V := FALSE
  @act2_reset W := FALSE
```

Figure 8.35: Application of TR_retry_reset

combinators (all, xor,...), leaf and the connections, in addition to the toolbar buttons which allows the user to validate the diagram and generate Event-B.

Refining an ERS diagram, requires refining the machine first, the ERS diagram will be automatically copied to the refined machine, the user then can refine the leaves of the new copied ERS diagram by right clicking the leaf and selecting refine, as shown in Figure 8.39. The manually added specification elements, which in our case are any
requirement specifications that are not related to control flow, are automatically copied to the solid child after refining a leaf.

### 8.6 Conclusion

In addition to the original twenty three translation rules (Salehi Fathabadi, 2012), we added eleven new translation rules, in addition to updating some of the translation rules (e.g. TR\textunderscore loop1) and functions. The newly added rules are summarised in Table 8.6.

<table>
<thead>
<tr>
<th>Rule Name</th>
<th>Rule Signature: ERS $\Rightarrow$ Event-B</th>
</tr>
</thead>
<tbody>
<tr>
<td>TR_rangeExpression</td>
<td>replicator $\Rightarrow$ guard(s)</td>
</tr>
<tr>
<td>TR_rep</td>
<td>replicator $\Rightarrow$ invariant(s)</td>
</tr>
<tr>
<td>TR_par1</td>
<td>par-rep $\Rightarrow$ guard(s)</td>
</tr>
<tr>
<td>TR_par2</td>
<td>par-rep $\Rightarrow$ par-rep successor event(s) guard(s)</td>
</tr>
<tr>
<td>TR_par3</td>
<td>par-rep $\Rightarrow$ invariant</td>
</tr>
<tr>
<td>TR_loop4</td>
<td>loop $\Rightarrow$ invariant(s)</td>
</tr>
<tr>
<td>TR_interrupt</td>
<td>interrupt/retry $\Rightarrow$ guard(s)</td>
</tr>
<tr>
<td>TR_disable_interrupt</td>
<td>interrupt/retry $\Rightarrow$ guard(s) in interrupting sub-flow</td>
</tr>
<tr>
<td>TR_interrupt_inv</td>
<td>interrupt/retry $\Rightarrow$ invariant</td>
</tr>
<tr>
<td>TR_retry</td>
<td>retry $\Rightarrow$ action(s)</td>
</tr>
<tr>
<td>TR_retry_reset</td>
<td>retry $\Rightarrow$ resetting event</td>
</tr>
</tbody>
</table>

Table 8.6: New ERS Translation Rules
The formal definition of the ERS syntax and its translation rules have made it easier to
automate the translation of the ERS language to Event-B language using the ERS plug-in, which extends the Event-B tool, Rodin. Having a tool that automatically generates Event-B models from ERS diagrams can encourage applying ERS for workflow modelling and also reduce the human error.

Moreover, the way the ERS tool can be used, which starts from an existing Event-B machine, makes it easier to combine ERS with other approaches based on Event-B semantics, such as the iUML-B statemachines as we have seen in Chapter 6. In addition to allowing the user to manually extend the model with manually added specification elements, which as explained earlier are not lost but will be reserved, even if they are added to an ERS element, they will be still copied to the solid child if refinement is done, and the user can still edit the manually added specification elements.
Chapter 9

Conclusions

In this chapter we present a summary of our main contributions and the lessons learned during the development of our case study, and identify the limitations in our work. We also evaluate our approach against other approaches that focus on workflow modelling and integration of diagrammatic approaches with formal modelling. Finally we present some directions for future work.

9.1 Contributions

The main contributions of this thesis can be summarised as follows:

- **Defining a workflow modelling approach** that combines both formal and semi-formal (graphical) modelling. This approach uses the formal method, Event-B for data handling and the Event Refinement Structures (ERS) diagrams for control flow.

- **Extending the ERS approach** to provide better support for workflow modelling, these extensions are divided into two types:
  - **Dynamic Modelling**: Includes defining a new combinator (par-replicator) that provides flexibility in modelling dynamic changes of data-dependent workflows. This new combinator allows us to model behaviours like the parallel producer-consumer pattern. We have also extended the parameter definition to include sets that can be defined dynamically as well as statically. Finally, we generalised the loop definition allowing the non-deterministic execution of the loop child events.
  - **Exception Handling**: Introduces two new combinators (interrupt, retry) that provide a mechanism to support exception handling. Both combinators allow one sub-flow (representing the exception) to stop the execution of
Chapter 9 Conclusions

another sub-flow (the normal behaviour) before completion. The retry combinator, in addition to the interruption feature, resets the control variables of the normal sub-flow in an attempt to execute the normal sub-flow again.

- Validation and evaluation of our workflow modelling approach and our ERS extensions through application to two case studies:
  - Fire dispatch workflow
  - On-line booking workflow of the travel agency

- Formal specification of our extensions through translation rules that map the ERS elements to Event-B. In addition to updating the original ERS translation rules and functions to include our extensions, we define an additional eleven translation rules that are exclusively related to our ERS extensions.

- Tool support: Extending the ERS language and adding the ERS translation rules to an improved version of the ERS plug-in that provides graphical support in addition to some validation of the diagrams.

9.2 Limitations

In Chapter 3 and Chapter 8, we have explained some of the restrictions of the ERS diagrams. Some of these restrictions are necessary to ensure the proper behaviour of the ERS diagrams like the case of the solid last child of the normal sub-flow of an exception handling combinator. Other restrictions are related to technical issues and it might be possible to remove them if we find cases that requires so, such as:

- A loop cannot be the first child of a flow.
- The loop and the par-replicator cannot be directly followed by an exception handling combinator
- The loop and the par-replicator if followed by a flow (i.e. a refined leaf) then the first child must be a solid child.
- The first child of the interrupting sub-flow of a an exception handling combinator must be a solid child.

Other restrictions such as the single solid line rule and applying a combinator to a leaf only have proved through different case studies, to be necessary to simplify the ERS diagrams and their generated models. Having simpler models will in turn facilitate the automation of the proofs. Similarly the restriction that does not allow the same leaf
name in different branches is important due to dependency between the events, which if removed could lead to a deadlock. Take the simplest case which is leaf $A$ followed by leaf $A$. The second leaf event $A$ requires the execution of $A$ and at the same time checks that $A$ has not executed. If the modeller requires the same event at different branches, then the solution is to define similar events, but with different names.

## 9.3 Lessons Learned

In this section we present the lessons learned while using ERS in modelling the case studies:

### 9.3.1 Separation of Control Flow and Data Handling

In our workflow modelling approach, we adopt the separation of control flow from data handling, where we model control flow using ERS diagrams and leave the data handling to be done manually using Event-B. However, in our case the separation does not mean the independence of the control flow from the data flow, this is because the ERS diagrams semantics are defined by transforming the diagrams into Event-B. Moreover defining explicit control flow variables in Event-B, have made it possible to relate the data variables to the control flow variables.

In both case studies, we have explicitly defined, using invariants, the relation between manually defined data variables and the ERS automatically generated control variables. By defining such invariants, we managed to define the scope of the data variables, i.e., which events they are relevant to. In addition to that, in many cases we used a combination of data variables and control variables to model the requirements, e.g., the requirements related to closing an incident in the fire dispatch case study.

An advantage of the separation is having simpler diagrams without complicating them with lots of specification elements concerned with data. The simpler the ERS diagrams are, the simpler the Event-B models are; hence easier verification as we have seen most of the proof obligations of our case studies were discharged automatically.

### 9.3.2 Integration of ERS with Other Event-B Approaches

Using the ERS plug-in we have seen how we used ERS with iUML-B statemachines, where we used ERS to model the overall dispatch workflow related to an incident and used statemachines to model the state changes of a physical resource. The common events between the ERS diagram and the statemachine were extended by the additional
guards and actions resulting from the statemachine. We have also extended the ERS diagram in the travel agency case study with timing patterns.

The integration of ERS with other approaches generating Event-B models, such as iUML-B statemachines, is similar to the shared event composition approach (Silva and Butler, 2010), where the composed machine includes variables, invariants and events from both approaches. The common events are synchronised by the conjunction of the guards and the combination of their actions. Consider, for example Figure 9.1(a) representing a model generated by ERS and Figure 9.1(b) representing a model generated by a different approach or even a different ERS diagram. The integration of both models is represented in Figure 9.2 showing how the common event is synchronised.

![Machine M](a) ERS

![Machine M](b) Other Approach

Figure 9.1: Machine $M$ Generated by Applying Different Approaches

Sometimes the integration may cause a deadlock if the shared events constraints generated by the different approaches or diagrams are inconsistent, that is why it is always required to check for deadlock freedom using ProB (Leuschel and Butler, 2003; Hallerstede and Leuschel, 2011). Consider the two ERS flow diagrams in Figure 9.3, integrating such diagrams will cause a deadlock, because $B$ requires the execution of $A$ in $ERS1$, and $A$ requires the execution of $B$ in $ERS2$. However, such deadlock can be easily detected using ProB model checking.

### 9.3.3 Proper Behaviour of the Model

In addition to supporting control flow, in many cases we have seen how the ERS translation rules, which are related to refinement have ensured that requirements are satisfied by the model. For example, the `par-replicator` refinement rule ensures that the same number of instances have executed for both the first and last events of the `par-replicator`
sub-flow, before it can be interrupted by the follow-on event. In the fire dispatch case study (Chapter 5), the \textit{par-replicator} refinement rule ensured that any allocated resource to the incident must be deallocated before considering the incident as managed. Likewise, in the travel agency case study (Chapter 7) the \textit{par-replicator} refinement rule ensured that any selected flight/car/hotel must be either booked or removed from the itinerary before completing the itinerary.

### 9.3.4 Modelling New Behaviours and Features

The addition of the new ERS extensions made it possible to model some behaviours that were difficult or even not possible to represent in the original ERS. The \textit{parallel producer-consumer} pattern is one of these behaviours that could not be represented by the original ERS, while the addition of the \textit{par-replicator} and the extension of the parameter definition made it possible to explicitly model this behaviour using ERS.

Before introducing the exception handling combinators, it was difficult to model one flow interrupting another, as we have seen in the duplicate case of the fire dispatch case study (Chapter 5), where we had to change the requirement or else using the original ERS
would have ended with a complex ERS diagram with many alternative branches. After introducing the *interrupt* combinator we managed to model the original requirement (Chapter 6) in a simple ERS diagram.

The new *retry* combinator, has made it possible to model cancellation behaviours. Although this combinator only resets control variables, we have shown in both case studies how we can manually undo data related actions. For example, in the case of a lost allocation of a physical resource, we used the *retry* combinator to make it possible to get a replacement, where we manually cancelled the allocation of the physical resource to the incident.

### 9.3.5 Role of Invariants

The ERS approach generates invariants to ensure the behaviour of the model is maintained all the time. In refinement some of the ERS generated invariants also help to discharge proof obligation by avoiding having repeated guards in events. This is in particular useful when the first child is not the solid child, and normally in ERS we add the guards to the first child of the flow in refinement, therefore by having explicit invariants we can discharge *GRD* proof obligations and avoid having unnecessarily repeated guards.

Take for example the refinement of *IncidentHandling* in Chapter 5. As shown in Figure 9.4, the refining event of *IncidentHandling* is the last event *DeallocateRes*. Applying the *par-replicator* to *IncidentHandling* will result in the guard \( i \notin \text{IncidentManaged} \) added to the event *IncidentHandling*. After refining the *IncidentHandling* event, the *par-replicator* guard will be added to the first event *AllocateRes* to interrupt the execution of the *par-replicator* flow, once the *IncidentManaged* event occur. However *DeallocateRes* is the refining event of *IncidentHandling*, hence according to the *GRD* proof obligation we need the *par-replicator* guard in *DeallocateRes*. In this case the following invariant resulting from the *par-replicator* refinement will ensure that the *GRD* proof obligation is satisfied:

\[
\forall i \cdot i \in \text{IncidentManaged} \implies \text{AllocateRes}[\{i\}] = \text{DeallocateRes}[\{i\}]
\]

### 9.3.6 Role of ERS Diagrams in Refinements and Proofs

In ERS, applying more than one combinator to the same event at the same refinement level is not permitted. Such restriction has the advantage of imposing smaller changes at each refinement level. Some studies have shown that such approach, although resulting in more refinements, helps in reducing the proving effort and help achieve more automation.
of the proofs (Butler and Yadav, 2008; Butler, 2009b), and we have seen in both case studies that most of the proofs were done automatically.

### 9.4 Comparing ERS with Other Workflow Modelling Approaches

#### 9.4.1 Workflow Control Flow Patterns

Russell et al. (2006a) describe forty three control flow patterns for business process modelling. This is an extension to their work which initially described twenty control flow patterns in van der Aalst et al. (2003). They use coloured Petri-Nets to describe the patterns and they do an evaluation to different control flow offerings including UML activity diagrams.

One of the important features of ERS is multiple instance modelling, which can be done using the different replicators (*all, some, one, par-replicator*), among the forty three patterns, they identify seven multiple instance patterns. The patterns “Multiple Instances with a Priori Design-Time Knowledge” and “Multiple Instances with a Priori Run-Time Knowledge”, as defined by the Workflow Initiative allow the concurrent execution of multiple instances, where any subsequent tasks cannot start before the completion of all the task instances. The different between the two patterns is that, the number of instances is in the former pattern is known at design time, while in the latter it is known during runtime. Both patterns can be represented by the ERS *all-replicator*, where in the first one the type of the set is constant, while in the second one, the type of the set is a variable.

The pattern “Multiple Instances without a Priori Run-Time Knowledge” allows the concurrent execution of multiple instances where the number of instances is not known
beforehand and new instances can be initiated during execution. The pattern “Dynamic Partial Join for Multiple Instances” is a variant of “Multiple Instances without a Priori Run-Time Knowledge” pattern, but does not require the completion of all created instances to trigger the next task. Both patterns can be represented by the par-replicator and the parallel producer-consumer pattern, by defining the required completion criteria which is determined by the par-replicator next event.

The remaining multiple instances patterns are indirectly captured by the some and one replicators. The pattern “Multiple Instances without Synchronisation”, allows the creation of multiple instances of an activity that can run concurrently and do not require synchronisation upon completion. This could possibly be captured by the some-replicator. However, in the case of the some-replicator, if there is an event following the some-replicator, then its execution requires the execution of at least one instance of the some-replicator. One possibility to avoid the one instance execution requirement, can be achieved by applying weak sequencing when introducing the some-replicator.

The pattern “Static Partial Join for Multiple Instances”, allows the concurrent execution of multiple instances of a task, where \( m \) instances are created. Once \( n \) instances (\( n < m \)) have completed, the next task is triggered. The pattern “Static Partial Join for Multiple Instances” is similar to the some-replicator, where \( n = 1 \) is always the case for the some-replicator. If we add an additional guard to the some-replicator follow-on event specifying the required \( (n) \) number using Event-B, then this pattern can be captured by ERS.

The pattern “Cancelling Partial Join for Multiple Instances” is similar to “Static Partial Join for Multiple Instances”. The only difference is that once the required \( n \) instances have executed, the “Cancelling Partial Join for Multiple Instances” pattern will cancel the execution of the remaining \( m - n \) instances. The one-replicator represents a special case of the pattern “Cancelling Partial Join for Multiple Instances”, where \( n = 1 \) is always the case. However, adding an additional guard using Event-B to the some-replicator to disable the execution of the some-replicator after a specified number \( (n) \) can cover this pattern. The par-replicator can also represent this pattern.

This shows how the ERS four replicators captured the behaviour of all the seven multiple instances patterns. The same applies to most of the other control flow patterns of the workflow patterns initiative. Here we focused on the multiple instances patterns to show the need for a dynamic pattern like the par-replicator which in this case managed to capture the behaviour of three of the multiple instances patterns, and also the addition of range-expression helped to cover these patterns.

The long list of patterns defined by Russell et al. (2006a) is criticised by Börger (Börger, 2007a,b), where the patterns are simplified into eight general patterns, described using abstract state machines (ASM) (Börger, 2005), which is a rule-based modelling approach. The rest of the patterns can be derived from the basic eight, using parameter
instantiation, refinement and composition. ASM and the eight control flow patterns are described in more detail in chapters 2 and 3.

In Table 9.1, we evaluate the extended ERS patterns against the eight generic patterns, in the same way Russell et al. (2006a) evaluated different workflow approaches against the forty three patterns. The first four patterns represent the sequential category, while the last four represent the parallel category. The “+” score means the pattern is explicitly supported by ERS, while the “−” scoring means, it is not explicitly supported by ERS.

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Score</th>
<th>Motivation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequence</td>
<td>+/−</td>
<td>Directly supported by the left-to-right arrangement of events, the “−” is because of the related milestone pattern which can be supported by additional Event-B guards to check the specified state.</td>
</tr>
<tr>
<td>Iteration</td>
<td>+/−</td>
<td>Structured loops are supported, but recursion and arbitrary cycles, i.e., cycles having more than one entry or exit point, are not supported.</td>
</tr>
<tr>
<td>Begin/Termination</td>
<td>+/−</td>
<td>Can be explained by the enabling/disabling conditions in Table 4.1 and Table 6.1, where some combinatorors have explicit termination like the loop and the par-replicator, while others termination can be implicitely described by their disabling conditions. The cancellation patterns are classified under termination, can now be supported by the interrupt-combinator. However, we still give a “−” because of the CancelRegion where interrupt cannot cancel any set of unconnected activities, which is still doable using Event-B.</td>
</tr>
<tr>
<td>Selection</td>
<td>+</td>
<td>Supported using or, xor combinators and their generalisations into some, one replicators.</td>
</tr>
<tr>
<td>Parallel Split</td>
<td>+</td>
<td>Supported using and, or combinators and the all, some, par replicators, where the par-replicator supports dynamic sets.</td>
</tr>
<tr>
<td>Merge</td>
<td>+/−</td>
<td>merging with synchronisation is supported by and/all, while merging without synchronisation is supported by or/some. The – because some variants of the merge pattern described by the workflow patterns initiative are not supported</td>
</tr>
<tr>
<td>Interleaving</td>
<td>+/−</td>
<td>No explicit ERS combinator, but parallelism in Event-B is only supported through interleaving. Concurrency supported by and, all, etc. can only be done through interleaving in Event-B</td>
</tr>
<tr>
<td>Trigger</td>
<td>+/−</td>
<td>Can be implicitly represented using monitored guards in Event-B, however an enabled event in Event-B is permitted to execute but cannot be forced to execute.</td>
</tr>
</tbody>
</table>
Regarding the interleaving pattern, there is no special ERS combinator that only supports interleaving, the ERS combinators (and, or, all, some, par) support concurrency in general including interleaving. On the other hand, Event-B only allows the non-deterministic execution of one event if more than one event is enabled, so the actual parallel execution of the Event-B events is done through the interleaving of events. Therefore since ERS semantics is based on Event-B, then concurrency can only be achieved by the interleaving of events. However, the author in Börger (2007b) mentions that interleaving might be better treated as a different form of parallelism rather than a pattern by itself, i.e., interleaving can be another form of the parallel split pattern. Similarly the trigger pattern can be a form of the start patterns.

In the merge pattern we gave ERS partial support, because of variants like the Multi-Merge are not supported by ERS. For example, the Multi-Merge pattern allows the convergence of multiple branches into a single branch. However, unlike the ERS or-combinator, the execution of each incoming branch will result in enabling the execution of the subsequent branch.

In Table 9.1, most of the patterns if not supported explicitly by ERS can be supported implicitly by adding some additional Event-B guards and actions to the existing ERS combinators. In most of the patterns we have seen partial support instead of a full support, due to the specific variants specified by the workflow patterns initiative. However, if we treat the patterns in general without getting into the detailed specification as the workflow patterns initiative does, most of the partial support can be changed into full support.

By comparing the ERS constructors with Börger’s approach, we can see that ERS can be used for workflow modelling, but one of the main differences is our distinction between the constructors that introduce dynamic and static parameters like the difference between the all-replicator and the par-replicator, where the all parameter value, unlike the par-replicator parameter, should be static i.e. known before executing the all event and cannot be changed after its execution. The ASM eight patterns does not distinguish between the static and dynamic character of the parameter and leave it as a matter of how the parameter is declared.

Petri Nets (Murata, 1989) are also widely used as a graphical formal approach for workflow modelling. However, Petri Nets are criticised by van der Aalst and ter Hofstede (2005) for their limitations in supporting multiple instances patterns, cancellation and advanced synchronisation patterns, which is why they present YAWL (Section 2.6), inspired by Petri Nets but extended to overcome these limitations. ERS, on the other hand supports multiple instances through its different replicators, also cancellation is supported by the new extensions interrupt and retry, where cancellation in this sense means aborting the execution of subsequent tasks when cancellation happens, or in other
words removing the tokens of execution. Advanced synchronisation is supported by the *or-combinator* of ERS.

### 9.4.2 UML Activity Diagram

Activity diagrams are one of the Unified Modelling Language (UML) diagrams (Rumbaugh et al., 1998), that describe the behavioural part of systems. Activity Diagrams are widely used to describe control flow (omg.org, 2012). In Dghaym et al. (2013), we have done a comparison between activity diagrams and ERS, and evaluated both approaches by applying them to a fire dispatch case study.

Figure 9.5 from Dghaym et al. (2013), represents the various ERS non-replicator combinators and their counterparts in activity diagram. As shown in the diagram all the non-replicator combinators of ERS (*and, xor, or, sequencing, loop*), can be represented using activity diagrams. The *and-combinator* in ERS is represented as a fork to describe the concurrent execution of Eve1 and Eve2, and then they are joined or synchronised into one flow before Eve3 can be executed. The *xor-combinator* is similar to an activity diagram decision-node with mutually exclusive guards.

The *or-combinator* means at least one of the sub-events is executed before the next event (*Eve3*) can be enabled, that is why we use a merge-node in the activity diagram which requires at least one flow to proceed and does not synchronise the flows like the join-node. However, in the ’or’ case in Figure 9.5, *Eve3* in ERS can be executed only once even if both *Eve1* and *Eve2* have executed, whereas in activity diagram according to the token flow translation, *Eve3* will be executed twice if both *Eve1* and *Eve2* have executed.

The left to right sequencing in ERS is shown by the direction of the arrow in activity diagrams. Finally, the star symbol, which indicates looping, can be described by using a decision-node and a merge-node that bring all flows together without synchronisation. Here the decision-node is added before the loop event (*Eve1*) to describe the zero iteration. UML 2.0 has introduced a structured node to represent a *loop*, but it is still a syntactic sugar that is equivalent to applying the decision and merge nodes as shown in Figure 9.6.

UML 2.0 has also introduced a new construct called expansion region. The expansion region, shown in Figure 9.7 (omg.org, 2012), is a structured activity node that takes collections of elements as input and can return output collections described in expansion nodes. The execution of elements can be described in three different modes (parallel, iterative, stream) according to a keyword at the top of the region. In the parallel mode, the elements execute concurrently and can overlap in time. In the iterative mode, execution is done in sequence, so the execution of one element can not start until the previous one completes its execution and if ordering is applied to the input elements, the
same order must be followed during execution. In a stream mode, there is exactly one execution in the region, where elements are offered to execution in a stream, preserving ordering if applied. That is, a stream mode execution is continuous and elements are processed as soon as one is available.

Regarding the ERS replicators (all, some, one), there are no explicit counterparts for the some and the one replicators, but the all replicator can be equivalent to the expansion region in the parallel mode. Similar to the ERS replicators (all, some, one), the expansion region cannot change its input during its execution and the input nodes must be determined in advance, hence activity diagrams cannot support dynamic behaviour.
like the *par-replicator*. However, ERS does not have an equivalent representation of the expansion region in the iterative and the stream modes.

When it comes to exceptional behaviour, we have explained in Chapter 6, how the *interrupt* combinator can support behaviour similar to the interruptible activity regions of activity diagrams. However the main difference between the ERS *interrupt* and the activity diagram interruptible region is that ERS interrupted tasks must be within the same subactivity, whereas the interruptible region can interrupt arbitrary tasks that are not related or not connected.

Regarding the resource perspective, which is one of the four perspectives of workflow modelling (Control, Data, Exception Handling, Resource) as described by the *Workflow Patterns Initiative*, activity diagrams support activity partitions or swimlanes where actions and activities are grouped according to whom or what is performing them. In Dghaym et al. (2013), we have done the resource grouping in ERS using colour coding of the ERS events according to whom/what is performing the event.

### 9.4.3 iUML-B Statemachines

iUML-B (Snook, 2014; Wiki.event-b.org, 2014) is another form of UML-B (presented in Section 3.3.5), but with stronger integration with Event-B. Some of the features supported by iUML-B state-machines include junctions, forks, joins and multiple instances.

Junctions in iUML-B result in the addition of a disjunctive guard to the events. A new pseudo state called “*any*” is also introduced to iUML-B, which represents all the states at the same level of the statemachine, where a transition with source *any* can fire irrespective of the current state of the statemachine. Lifting is another feature introduced to iUML-B, where a state-machine can be lifted to multiple instances.

However, the main difference between ERS and iUML-B is what is the purpose of the diagram, where ERS diagrams focus on events, iUML-B statemachines focus on the state and the state transitions of the model. Therefore, choosing between the approaches depends on the nature of the problem, like in Chapter 6, we have shown how we used ERS to represent the dispatch system of the incident, but when it came to modelling the state changes of the resource, we used iUML-B statemachines, and both approaches complemented each other.
9.5 Other Work on Integration of Formal Methods and Graphical Representations

9.5.1 Tree-Like Structures Integrated with Formal Methods

Matoussi et al. (2011) describe a transformation of the KAOS goal model to the Event-B formal model. KAOS (Keep All Objectives Satisfied) is a goal based requirement engineering method, providing an overall representation of the system’s goals, objects, agents, operations and behaviours (van Lamsweerde, 2008). Matoussi et al. focus on the goal KOAS model in their work, where each goal is translated into an event in the Event-B model, with actions describing the goal achievement and guards are the pre-conditions of the goal. Regarding refinement of the Event-B model, they follow the same approach of goal refinement where goals are refined into AND/OR constrained sub-goals. AND-refined means all the sub-goals need to be achieved in order to achieve the parent goal, whereas OR-refined means achieving one sub-goal is enough to achieve the parent goal (Darimont and van Lamsweerde, 1996). Regarding the transformation to Event-B, they describe the sequencing, the AND and the OR, which is described as mutually exclusive. There is no mention of the AND and OR generalisations into some and all, and unlike ERS the type, ordering and gluing invariants are not described and must be done manually by the modeller. Figure 9.8 shows a safe transportation example (Darimont and van Lamsweerde, 1996) using KAOS goal model. In this example AND-refinement of the sub-goals is used at the first level, while OR-refinement is applied in the refinement of “No Collision” goal at the second level.

![Figure 9.8: Safe Transportation (van Lamsweerde, 2008): Example of KAOS Goal Model](image-url)

Laleau et al. (2010) also use the KAOS goal model to reduce the gap between requirements and formal methods. They use the KAOS goal model to extend the SysML requirements model, and then map this model to the B method. Each goal is mapped
to a B machine with an associated operation describing the goal, the refinement is then described similar to the sub-goals refinement AND/OR and milestone or sequence of sub-goals, in a way that each sub-goal is defined in a new B-machine with a corresponding operation, and the refinement is described by a new machine refining the abstract goal and including the sub-goals machines; the refinement behaviour (AND/OR/Sequence) is described within an operation in the refining B machine. This approach includes lots of machines in the B method and they also do not describe invariants related sequencing, typing and gluing like ERS, and there is no notion of multiple instances representations.

9.5.2 BPMN and Formal Methods

BPMN, which is adopted as the de-facto for business process modelling, has been extended with formal semantics using Event-B in Bryans and Wei (2010). In the translation of BPMN to Event-B, the interpretation of control flow is based on tokens, where a function is defined mapping each process instance to the number of tokens in that instance. Each flow is then guarded by the number of instances required to commence execution. The ERS approach uses subset relationships with events having corresponding control variables with the same names. The advantage of using subset relationships appears in the ability of defining control flow, not only using guards but also using invariants, ensuring that ordering is strongly specified and maintained by invariants. Moreover, the naming convention used by ERS, makes it easier to understand and track the ordering of events. Another key advantage of applying ERS is the explicit refinement relationships between events at different refinement levels

In Börger and Sörensen (2011), a formal semantics of BPMN is described using Abstract State Machines (ASM). Both previous works cover control and data flow of BPMN. In ERS data flow is left to be specified by Event-B. However, regarding the dynamic change of multiple instances, BPMN does not support the addition of new instances once a task has commenced execution, unlike the par-replicator, which gives the flexibility to support dynamic parameters. On the other hand, BPMN supports both structured and arbitrary cycles, whereas ERS only supports structured iteration and replication.

9.5.3 UML Activity Diagrams and Formal Methods

de Sousa et al. (2011) investigate the translation of IOD or interaction overview diagrams into Event-B, to describe the behavioural part of the model. IOD is a variant of UML activity diagrams, but with more restrictive syntax that does not allow the use of object nodes and object flows and instead of actions, IOD uses interactions. Regarding the static part, they extend the UML-B class diagram with BCE or boundary-control-entity class diagram. In their translation, they define a boolean variable for each control flow including decisions, they also define variables for each class and association in the
class diagram. Then each interaction of the IOD is associated with an event, and each construct (join, fork, decision, merge) is associated with a special event. Compared to the ERS approach, the ERS combinators are not defined as separate events but their effect is shown by the sequencing invariants and guards in the events. They also associate a new boolean variable for each control flow edge, whereas ERS describes the control flow using the defined event control variables which can be easier to follow. As mentioned earlier, IOD does not allow the use of objects so there is no notion of multiple instances in their representation.

Refinement is described in de Sousa et al. (2012), but not in one type of diagrams like ERS. It is based on the ICONIX process that includes four types of UML diagrams (class, use cases, sequence and robustness which is a hybrid of the activity and class diagrams). They start with identifying the domain of the model which is mapped to sets and relations, then derive the use cases diagram which is mapped into events in the Event-B model. Then in the first refinement, they derive a robustness diagram from each use case which adds new events, invariants and new class associations to update the domain model. Lastly in the second refinement, they derive a sequence diagram from each robustness diagram updating the control, entity and boundary classes and the associations between them, however the translation of the sequence diagram into Event-B is not described yet. This process of refinement helps in understanding the system using different views of UML diagrams, but it does not describe exactly how events are refined and how they relate to abstract ones.

Ben Younes and Ben Ayed (2008) also describe the translation of UML activity diagram into Event-B. In their translation they describe refinement of Event-B machines in each nested activity, and they determine the ordering of events by a decreasing variant. Regarding their translation. They extend their work in Ben Younes et al. (2012) to describe the refinement patterns, where they apply a similar approach to goal refinement using sequence/AND/OR refinements. However, they do not describe how to handle new constructs of activity diagrams like expansion regions which is related to multiple instances modelling.

As mentioned in Section 2.3.1, version 2.0 of UML describes the semantics of activity diagrams based on token flow inspired by Petri nets. This resulted in many attempts to formalise the semantics of activity diagrams using Petri nets and its variants. However, most of these translations are based on earlier versions of UML activity diagram or only concentrated on the basic structures of UML 2.0 activity diagrams. Storrle (2004) is one of the few that concentrated on newly introduced structures of activity diagrams like expansion regions. Storrle describes the semantics of structured nodes like loop nodes and conditional nodes as *syntactic sugar*, since they can be defined by other basic nodes. However, the situation is not that simple when it comes to expansion regions. So the author defines the formal semantics of expansion regions using procedural Petri nets. By this, he means translating the expansion regions as refined actions, where various calls
to the same refined transitions are executed into its own space. The different modes of
the expansion region implies different types of refinement nets.

Syriani and Ergin (2012) describe model transformation rules mapping constructs of
UML 2.4.1 activity diagrams to Petri nets. In these transformation rules they describe
the preconditions that must be found, and the negative application conditions (NACs)
which are the precondition patterns that must not be found and prevent the application
of the rules, and finally the postconditions which are the results of the transformation.
These rules are divided into 11 phases to describe execution order. Each phase consists
of a set of rules that can be applied at any order. An example of transformation rule, is
the mapping of an action node into an entry transition, a processing transition and an
exit transition. For expansion regions they introduce the concept of inhibitor arcs, that
ensure all the inputs are processed before sending them to output. In this work, there
is no notion of refinement.

9.5.4 Other UML Diagrams and Formal Methods

Idani et al. (2006) follow a reverse engineering approach where they derive a UML class
diagram from B specifications, in order to help the customers understand the formal
model and also help the modellers better understand the structure of their models.
From each refinement of the B specifications, they derive a pertinent context where
they ensure that each set and operation appears only once and then the contexts are
automatically transformed into class diagrams, but this tool is still limited because
it requires the interference of the modeller to select the sets. This approach helps
understanding the formal model and provides good documentations of the formal model,
but still cannot bridge the gap between requirements and formal methods, since it is
applied after finishing the formal model.

Lausdahl et al. (2009) describe a mapping between UML and VDM++, which extends
VDM, explained in Section 3.2.1, to support object oriented modelling and concurrency.
They do two types of mappings, a bidirectional mapping between UML class diagrams
and VDM++, and a one direction mapping from UML sequence diagrams to VDM++. The
class diagrams describe the static part of the VDM++ model, whereas sequence
diagrams describe the dynamics of the model. Here, they transform the sequence di-
agrams into trace definitions, in order to generate automatic test cases. While in our
case, we use the ERS diagrams to build the dynamic part of the model.

Amálio et al. (2010) defines a framework for modelling using UML and Z. In this frame-
work they use three types of UML diagrams, which are class, state and object diagrams
that help later in generating and analysing a Z formal model. They start with a class
diagram that describes the static part of the model. Then a UML state diagram to
represent the dynamics of the model. These diagrammatic representations are then instantiated into templates (Amálio et al., 2007), which describe general structures and patterns of the formal language in this case Z, to generate the Z model. Finally they use object diagrams to validate the system against the requirements. They also apply object oriented structuring (Amálio et al., 2005, 2010) based on five different views, which are structural, intensional, extensional, relational and global, when generating the Z model from the instantiated templates. This framework helps in generating, analysing and validating models, but unlike ERS, it does not represent the refinement relations explicitly between the different operations, and do not describe explicitly how the different construct representations like all, and etc. are presented both diagrammatically and formally.

In Murali et al. (2015), they use UML use cases with Event-B. They define an accident use case which is an extension use case to model safety accidents which can result in an undesired loss. The accident use case is introduced through refinement describing the alternate flow if the accident is allowed to happen. This is similar to our approach where we introduce explicit exception handling combinators to explicitly define possible disruptions to the planned flow.

9.6 Overview of ERS in Comparison to Other Approaches

In this section we highlight the main strengths and weaknesses of the ERS approach in comparison to the previously discussed related work.

- One of the key strengths of the ERS approach is the support to multiple instance modelling through different combinators (par, all, some, one) and the or-combinator which is not supported by some of the workflow techniques. Moreover based on the evaluation with the control flow patterns, ERS supports most of these patterns either explicitly or implicitly when combined with additional Event-B guards, which is done using a reasonable number of combinators. However the ERS approach lacks the flexibility of modelling arbitrary jumps due to its structured nature.

- The models generated by ERS, are verified using the Event-B formal method, and we can also apply ProB for model checking and validation. This also applies to the models generated by iUML-B, however ERS provides a different viewpoint focusing on the events rather than the state changes.

- The diagrammatic nature of ERS showing the different refinement levels, also helps in making modelling decisions that facilitate the model verification such as the decision of when to introduce a data variable that can be modified by different events, in order to discharge EQL proof obligations generated during refinement.
• ERS rules, such as the single solid line rule, prompt doing smaller changes at each refinement level and hence facilitate the automation of discharging proof obligations.

• On the other hand, having lots of refinement levels can result in larger models and the hierarchical nature of ERS can result in additional events especially that the same event names cannot appear in different ERS branches.

• In addition to explicitly representing control flow, ERS structures the Event-B refinement and shows explicitly the relationship between events at different refinement levels.

• Most of the approaches that integrate formal modelling with graphical approaches aim at giving formal semantics to the graphical representations. ERS on the other hand extends the formal method Event-B to provide explicit support for control flow in Event-B and most importantly providing a systematic refinement approach.

• Control flow in ERS is not only expressed in terms of enabling conditions as most formalisms of workflow approaches do, but also as invariants that have to be maintained by the model.

9.7 Conclusion and Future Work

In this thesis we have presented a workflow modelling approach that combines formal modelling (Event-B) with a hierarchical graphical representation (ERS). This combination facilitates the incremental modelling of the workflows, providing explicit control flow representation which makes communication and validation easier. In addition to the advantages of formal modelling in verification.

Our approach extended the original ERS with combinators that facilitate dynamic modelling and support exception handling, thus expanding the usability of ERS to include more complex cases, as we have shown by the application to complex dispatch system of emergency services.

We have also presented a tool that generates Event-B models from ERS diagrams based on a well defined set of transformation rules. In the future we would like to improve our tool support by adding all the translation rules to the tool, providing more validations and trying to overcome some of the restrictions described in Section 9.2, if possible.

Another possible improvement that can be done to the tool is enhancing the graphical environment by allowing the user to add data in the ERS graphical editor without having to switch to the Event-B editor. We would like to focus on the improvement of the tool support which encourage users to apply and try our approach.

In addition to improving the tool support of our approach, our future work will include:
**Application to Case Studies:** Apply our approach to other industrial case studies, with special focus on exception handling and dynamic modelling.

**Timing Patterns:** In the travel agency case study, we have extended the ERS diagrams with the expiry timing pattern, we would also like to explore other timing patterns and how they can be used in conjunction with ERS diagrams.

**Decomposition:** The ERS approach helps in structuring the Event-B refinement which aims at tackling the modelling complexity. However having different refinement levels sometimes result in complex and large models and in some cases only parts of the model need to be refined, hence the need for model decomposition, which is another key concept in Event-B. Our approach transforms the ERS diagrams to Event-B models, therefore anything that applies to Event-B can be applied to the generated Event-B models. There are two approaches that support model decomposition in Event-B: shared variable (Abrial and Hallerstede, 2007; Abrial, 2009) and shared event (Butler, 2006, 2009a). Both approaches are supported by the Decomposition plug-in in the Rodin platform (Silva et al., 2010).

Applying model decomposition will generate new sub-models, although the new generated sub-models can be further refined independently using ERS, the original ERS diagrams will be lost and new ERS diagrams must be added to be able to apply the ERS approach in refinement. In the future we would like to integrate ERS with model decomposition by also allowing the ERS diagram decomposition based on the model decomposition approaches.

Such integration is feasible and in Butler (2009a) and Salehi Fathabadi et al. (2011), ERS and shared event decomposition were both applied, where ERS helped in preparing the model for decomposition. Model decomposition can benefit from the event visualisation provided by ERS when deciding how to split the events between the decomposed models.

For example the travel agency case study can be decomposed to three models: flight itinerary, hotel itinerary and car itinerary. Currently we can have three separate ERS diagrams for each itinerary and apply the shared event composition approach presented in Section 9.3.2, with the shared events `start itinerary` and `checkout`. However the three ERS diagrams will generate Event-B encodings in the same model, while decomposition will lead to three separate Event-B models.

**Transformation Correctness:** In Chapter 8 we have defined the translation rules which map the ERS elements to Event-B. The ERS translation rules are built using recursive functions to map the ERS operators and their possible combinations to Event-B. Currently the correctness of these transformations rely on the verification capabilities
of Event-B, by trying to prove all the proof obligations generated by the Event-B models and validate the generated models using ProB. The difficulty in proving the correctness of the ERS transformations to Event-B is that the ERS operators are not compositional, like trace semantics for example, where in ERS, we need to traverse down the hierarchy to define the operators. As a future work, we would like to prove the correctness of the transformation rules generating Event-B from ERS.

One possibility is to define the semantics of both ERS and Event-B within one framework and prove the correspondence between the two semantic models of the source (ERS) and target (Event-B). A possible framework is the Prototype Verification System (PVS) (Owre et al., 1992), which provides a framework for formal specification and verification with a powerful theorem proving mechanisms. PVS is a good candidate where in Bodeveix et al. (1999), a formalisation of the B-method (an ancestor of Event-B) using PVS was done, and in Ripon and Butler (2009), the different semantics of compensating CSP (cCSP) and their equivalence were mechanised using PVS.

Another possibility that we can start with is to study the proof obligations generated by the different combinations of the ERS combinators, and provide proof tactics for the proofs that cannot be automatically proved.
Appendix A

Event-B Model of the Fire Dispatch Workflow

In this chapter we present the complete Event-B version of the fire dispatch system presented in Chapter 5. We also present all the ERS diagrams of the model, which are done by the improved version of the ERS plug-in. An interim update site of the ERS tool is available on http://users.ecs.soton.ac.uk/dd4g12/.

A.1 Abstract Level

A.1.1 Context C0

CONTEXT C0
SETS INCIDENT
END

A.1.2 ERS Diagram: M0

Figure A.1: ERS Diagram at Abstract Level: M0
A.1.3 Machine M0

MACHINE M0
SEES C0
VARIABLES
  CreateIncident
  GatherInfo
  Duplicate
  NonDuplicate

INVARIANTS
  inv_createincident_type: CreateIncident ⊆ INCIDENT
  inv_gatherinfo_fd0_seq: GatherInfo ⊆ CreateIncident
  inv_duplicate_fd0_seq: Duplicate ⊆ GatherInfo
  inv_nonduplicate_fd0_seq: NonDuplicate ⊆ GatherInfo
  inv1_xor_fd0: partition((Duplicate ∪ NonDuplicate), Duplicate, NonDuplicate)

EVENTS

Initialisation
begin
  act_createincident: CreateIncident := ∅
  act_gatherinfo: GatherInfo := ∅
  act_duplicate: Duplicate := ∅
  act_nonduplicate: NonDuplicate := ∅
end

Event CreateIncident (ordinary) ≜ any
  i
  where
    grd_self: i /∈ CreateIncident
  then
    act: CreateIncident := CreateIncident ∪ {i}
end

Event GatherInfo (ordinary) ≜ any
  i
  where
    grd_seq_fd0: i ∈ CreateIncident
    grd_self: i /∈ GatherInfo
  then
    act: GatherInfo := GatherInfo ∪ {i}
end

Event Duplicate (ordinary) ≜
any
i
where
grd_seq_fd0: \ i \in \text{GatherInfo} 

grd_self: \ i \notin \text{Duplicate} 

grd1_xor: \ i \notin \text{NonDuplicate} 

then
act: \ Duplicate := \ Duplicate \cup \{i\}

end

Event NonDuplicate (ordinary) ≜
any
i
where
grd_seq_fd0: \ i \in \text{GatherInfo} 

grd_self: \ i \notin \text{NonDuplicate} 

grd1_xor: \ i \notin \text{Duplicate} 

then
act: \ NonDuplicate := \ NonDuplicate \cup \{i\}

end

END

A.2 First Refinement

A.2.1 ERS Diagrams: M1

![ERS Diagram at First Refinement: M1](image)

Figure A.2: ERS Diagram at First Refinement: M1

A.2.2 Machine M1

MACHINE M1
REFINES M0
SEES C0

VARIABLES
CreateIncident
GatherInfo
IsDuplicate
CloseDuplicate
NotDuplicate
SetInitialAP
IncidentManagement

INVARIANTS

inv_CreateIncident_type: CreateIncident ⊆ INCIDENT
inv_GatherInfo_fd0_seq: GatherInfo ⊆ CreateIncident
inv_IsDuplicate_Duplicate_seq: IsDuplicate ⊆ GatherInfo
inv_IsDuplicate_glu: IsDuplicate = Duplicate
inv_CloseDuplicate_Duplicate_seq: CloseDuplicate ⊆ IsDuplicate
inv_NotDuplicate_NonDuplicate_seq: NotDuplicate ⊆ GatherInfo
inv_NotDuplicate_glu: NotDuplicate = NonDuplicate
inv_SetInitialAP_NonDuplicate_seq: SetInitialAP ⊆ NotDuplicate
inv_IncidentManagement_NonDuplicate_seq: IncidentManagement ⊆ SetInitialAP

inv1_xorfd0: partition((IsDuplicate ∪ NotDuplicate), IsDuplicate, NotDuplicate)

EVENTS

Initialisation

begin
act_CreateIncident: CreateIncident := ∅
act_GatherInfo: GatherInfo := ∅
act_IsDuplicate: IsDuplicate := ∅
act_CloseDuplicate: CloseDuplicate := ∅
act_NotDuplicate: NotDuplicate := ∅
act_SetInitialAP: SetInitialAP := ∅
act_IncidentManagement: IncidentManagement := ∅
end

Event CloseDuplicate ⟨ordinary⟩ ≝

any

where

grd_seq_Duplicate: i ∈ IsDuplicate
grd_self: i \notin \text{CloseDuplicate}

\textbf{then}

\textbf{act}: \text{CloseDuplicate} := \text{CloseDuplicate} \cup \{i\}

\textbf{end}

\textbf{Event} SetInitialAP \langle \text{ordinary} \rangle \equiv

\text{any}

i

\textbf{where}

grd_seq_{NonDuplicate}: i \in \text{NotDuplicate}
grd_self: i \notin \text{SetInitialAP}

\textbf{then}

\textbf{act}: \text{SetInitialAP} := \text{SetInitialAP} \cup \{i\}

\textbf{end}

\textbf{Event} IncidentManagement \langle \text{ordinary} \rangle \equiv

\text{any}

i

\textbf{where}

grd_seq_{NonDuplicate}: i \in \text{SetInitialAP}
grd_self: i \notin \text{IncidentManagement}

\textbf{then}

\textbf{act}: \text{IncidentManagement} := \text{IncidentManagement} \cup \{i\}

\textbf{end}

\textbf{Event} CreateIncident \langle \text{ordinary} \rangle \equiv

\textbf{refines} CreateIncident

\text{any}

i

\textbf{where}

grd_self: i \notin \text{CreateIncident}

\textbf{then}

\textbf{act}: \text{CreateIncident} := \text{CreateIncident} \cup \{i\}

\textbf{end}

\textbf{Event} GatherInfo \langle \text{ordinary} \rangle \equiv

\textbf{refines} GatherInfo

\text{any}

i

\textbf{where}

grd_seq_{fd0}: i \in \text{CreateIncident}
grd_self: i \notin \text{GatherInfo}

\textbf{then}

\textbf{act}: \text{GatherInfo} := \text{GatherInfo} \cup \{i\}
Event IsDuplicate (ordinary) ≡
refines Duplicate
  any
    i
  where
    grd_seq_Duplicate: i ∈ GatherInfo
    grd_self: i ∉ IsDuplicate
    grd1_xor: i ∉ NotDuplicate
  then
    act: IsDuplicate := IsDuplicate ∪ {i}
  end
Event NotDuplicate (ordinary) ≡
refines NonDuplicate
  any
    i
  where
    grd_seq_NonDuplicate: i ∈ GatherInfo
    grd_self: i ∉ NotDuplicate
    grd1_xor: i ∉ IsDuplicate
  then
    act: NotDuplicate := NotDuplicate ∪ {i}
  end
END

A.3 Second Refinement

A.3.1 ERS Diagram: M2

A.3.2 Machine M2

MACHINE M2
REFINES M1
SEES C0
VARIABLES
  CreateIncident
  GatherInfo
  IsDuplicate
  CloseDuplicate
NotDuplicate
SetInitialAP
UpdateActionPlan
ManageIncident
CloseIncident

INVARINTS

inv_CreateIncident_type: CreateIncident ⊆ INCIDENT
inv_GatherInfo_fd0_seq: GatherInfo ⊆ CreateIncident
inv_IsDuplicate_Duplicate_seq: IsDuplicate ⊆ GatherInfo
inv_CloseDuplicate_Duplicate_seq: CloseDuplicate ⊆ IsDuplicate
inv_NotDuplicate_NotDuplicate_seq: NotDuplicate ⊆ GatherInfo
inv_SetInitialAP_NotDuplicate_seq: SetInitialAP ⊆ NotDuplicate
inv_UpdateActionPlan_seq: UpdateActionPlan ⊆ SetInitialAP
inv_ManageIncident_seq: ManageIncident ⊆ SetInitialAP
inv_CloseIncident_seq: CloseIncident ⊆ (UpdateActionPlan ∩ ManageIncident)
inv_CloseIncident_glu: CloseIncident = IncidentManagement
inv_xorfd0: partition((IsDuplicate ∪ NotDuplicate), IsDuplicate, NotDuplicate)

EVENTS

Initialisation
begin
  act_CreateIncident: CreateIncident := ∅
  act_GatherInfo: GatherInfo := ∅
  act_IsDuplicate: IsDuplicate := ∅
  act_CloseDuplicate: CloseDuplicate := ∅
  act_NotDuplicate: NotDuplicate := ∅
  act_SetInitialAP: SetInitialAP := ∅
Appendix A Event-B Model of the Fire Dispatch Workflow

```plaintext
act_UpdateActionPlan: UpdateActionPlan := ∅
act_ManageIncident: ManageIncident := ∅
act_CloseIncident: CloseIncident := ∅
end

Event UpdateActionPlan \langle \text{ordinary} \rangle \triangleq
any
  i
where
  grd_{seq}:: i \in SetInitialAP
  grd_{self}:: i \notin UpdateActionPlan
then
  act:: UpdateActionPlan := UpdateActionPlan \cup \{i\}
end

Event ManageIncident \langle \text{ordinary} \rangle \triangleq
any
  i
where
  grd_{seq}:: i \in SetInitialAP
  grd_{self}:: i \notin ManageIncident
then
  act:: ManageIncident := ManageIncident \cup \{i\}
end

Event CreateIncident \langle \text{ordinary} \rangle \triangleq
refines CreateIncident
any
  i
where
  grd_{self}:: i \notin CreateIncident
then
  act:: CreateIncident := CreateIncident \cup \{i\}
end

Event GatherInfo \langle \text{ordinary} \rangle \triangleq
refines GatherInfo
any
  i
where
  grd_{seq_fd0}:: i \in CreateIncident
  grd_{self}:: i \notin GatherInfo
then
  act:: GatherInfo := GatherInfo \cup \{i\}
```

end

**Event** IsDuplicate (ordinary) ≜
**refines** IsDuplicate

any

i

where

grd_seq.Duplicate: \( i \in \text{GatherInfo} \)
grd_self: \( i \notin \text{IsDuplicate} \)
grd1_xor: \( i \notin \text{NotDuplicate} \)

then

act: \( \text{IsDuplicate} := \text{IsDuplicate} \cup \{i\} \)

end

**Event** CloseDuplicate (ordinary) ≜
**refines** CloseDuplicate

any

i

where

grd_seq.Duplicate: \( i \in \text{IsDuplicate} \)
grd_self: \( i \notin \text{CloseDuplicate} \)

then

act: \( \text{CloseDuplicate} := \text{CloseDuplicate} \cup \{i\} \)

end

**Event** NotDuplicate (ordinary) ≜
**refines** NotDuplicate

any

i

where

grd_seq.NonDuplicate: \( i \in \text{GatherInfo} \)
grd_self: \( i \notin \text{NotDuplicate} \)
grd1_xor: \( i \notin \text{IsDuplicate} \)

then

act: \( \text{NotDuplicate} := \text{NotDuplicate} \cup \{i\} \)

end

**Event** SetInitialAP (ordinary) ≜
**refines** SetInitialAP

any

i

where

grd_seq.NonDuplicate: \( i \in \text{NotDuplicate} \)
grd_self: \( i \notin \text{SetInitialAP} \)
Appendix A Event-B Model of the Fire Dispatch Workflow

then
  \textit{act}: \textit{SetInitialAP} := \textit{SetInitialAP} \cup \{i\}
end

\textbf{Event} \textit{CloseIncident (ordinary) =}

\textbf{refines} \textit{IncidentManagement}

\textbf{any}

\textbf{i}

\textbf{where}

- \textbf{grd_seq}: \quad (i \in \textit{UpdateActionPlan} \land i \in \textit{ManageIncident})
- \textbf{grd_self}: \quad i \notin \textit{CloseIncident}

then
  \textit{act}: \textit{CloseIncident} := \textit{CloseIncident} \cup \{i\}
end

END

A.4 Third Refinement

A.4.1 Context C1

\textbf{CONTEXT} \textit{C1}
\textbf{EXTENDS} \textit{C0}
\textbf{SETS}
  \textit{LR}
END

A.4.2 ERS Diagram: M3
Figure A.4: ERS Diagram at Third Refinement: M3
Appendix A Event-B Model of the Fire Dispatch Workflow

A.4.3 Machine M3

MACHINE M3
REFINES M2
SEES C1

VARIABLES
CreateIncident
GatherInfo
IsDuplicate
CloseDuplicate
NotDuplicate
SetInitialAP
RequestAdditionalRes
SendStopMsg
HandleIncident
IncidentManaged
CloseIncident
ap added manually
notRequired_ap added manually

INVARIANTS

inv_CreateIncident_type: CreateIncident ⊆ INCIDENT
inv_RequestAdditionalRes_type: RequestAdditionalRes ⊆ INCIDENT × LR
inv_HandleIncident_type: HandleIncident ⊆ INCIDENT × LR
inv_GatherInfo_fd0_seq: GatherInfo ⊆ CreateIncident
inv_IsDuplicate_Duplicate_seq: IsDuplicate ⊆ GatherInfo
inv_CloseDuplicate_Duplicate_seq: CloseDuplicate ⊆ IsDuplicate
inv_NotDuplicate_NonDuplicate_seq: NotDuplicate ⊆ GatherInfo
inv_SetInitialAP_NonDuplicate_seq: SetInitialAP ⊆ NotDuplicate
inv_RequestAdditionalRes_seq: dom(RequestAdditionalRes) ⊆ SetInitialAP
inv_SendStopMsg_seq: SendStopMsg ⊆ SetInitialAP
inv_SendStopMsg_glu: SendStopMsg = UpdateActionPlan
inv_HandleIncident_seq: dom(HandleIncident) ⊆ SetInitialAP
inv_IncidentManaged_seq: IncidentManaged ⊆ SetInitialAP
inv_IncidentManaged_glu: IncidentManaged = ManageIncident
inv_CloseIncident_seq: CloseIncident ⊆ (SendStopMsg ∩ IncidentManaged)
inv1_xorfd0: partition((IsDuplicate ∪ NotDuplicate), IsDuplicate, NotDuplicate)
inv1: ap ∈ INCIDENT ↵ LR
inv2: dom(ap) = SetInitialAP
in \( \text{Handle\_Rep} \): \( \forall i \in \text{INCIDENT} \Rightarrow \text{HandleIncident}[\{i\}] \subseteq \text{ap}[\{i\}] \)

\( \text{inv\_RequestAdditionalRes\_rep} \): \( \forall i \in \text{INCIDENT} \Rightarrow \text{RequestAdditionalRes}[\{i\}] \subseteq \text{LR} \)

\( \text{inv\_notRequired\_ap\_type} \): \( \text{notRequired\_ap} \subseteq \text{ap} \)

manually added data variable to describe the actions plan items that are not required after stop message is sent

**EVENTS**

**Initialisation**

begin

act\_CreateIncident: \( \text{CreateIncident} := \emptyset \)

act\_GatherInfo: \( \text{GatherInfo} := \emptyset \)

act\_IsDuplicate: \( \text{IsDuplicate} := \emptyset \)

act\_CloseDuplicate: \( \text{CloseDuplicate} := \emptyset \)

act\_NotDuplicate: \( \text{NotDuplicate} := \emptyset \)

act\_SetInitialAP: \( \text{SetInitialAP} := \emptyset \)

act\_RequestAdditionalRes: \( \text{RequestAdditionalRes} := \emptyset \)

act\_SendStopMsg: \( \text{SendStopMsg} := \emptyset \)

act\_HandleIncident: \( \text{HandleIncident} := \emptyset \)

act\_IncidentManaged: \( \text{IncidentManaged} := \emptyset \)

act\_CloseIncident: \( \text{CloseIncident} := \emptyset \)

act1: \( \text{ap} := \emptyset \)

act\_notRequired\_ap: \( \text{notRequired\_ap} := \emptyset \)

end

Event RequestAdditionalRes \( (\text{ordinary}) \) \( \hat{=} \)

any

i

lr

where

\( \text{grd\_seq}: i \in \text{SetInitialAP} \)

\( \text{grd\_self}: i \mapsto lr \notin \text{RequestAdditionalRes} \)

\( \text{grd\_InputExpression1}: lr \in \text{LR} \)

\( \text{grd0\_par}: i \notin \text{SendStopMsg} \)

\( \text{grd1}: lr \notin \text{ap}[\{i\}] \)

then

act: \( \text{RequestAdditionalRes} := \text{RequestAdditionalRes} \cup \{i \mapsto lr\} \)

act1: \( \text{ap} := \text{ap} \cup \{i \mapsto lr\} \)

//manually added

end

Event HandleIncident \( (\text{ordinary}) \) \( \hat{=} \)

any
where

\text{grd}\_self: \ i \mapsto lr \notin \text{HandleIncident}
\text{grd}\_InputExpression1: \ lr \in ap\{i\}
\text{grd0}\_par: \ i \notin \text{IncidentManaged}
\text{grd}\_seq: \ i \in \text{SetInitialAP}

\text{then}

\text{act}: \text{HandleIncident} := \text{HandleIncident} \cup \{i \mapsto lr\}

\text{end}

\textbf{Event} \ \text{CreateIncident} \ \text{(ordinary)} \ \overset{\Delta}{=} \ \text{refines} \ \text{CreateIncident}
\text{any}

\text{any}

\text{where}

\text{grd}\_self: \ i \notin \text{CreateIncident}

\text{then}

\text{act}: \text{CreateIncident} := \text{CreateIncident} \cup \{i\}

\text{end}

\textbf{Event} \ \text{GatherInfo} \ \text{(ordinary)} \ \overset{\Delta}{=} \ \text{refines} \ \text{GatherInfo}
\text{any}

\text{any}

\text{where}

\text{grd}\_self: \ i \notin \text{GatherInfo}
\text{grd}\_seq\_{fd0}: \ i \in \text{CreateIncident}

\text{then}

\text{act}: \text{GatherInfo} := \text{GatherInfo} \cup \{i\}

\text{end}

\textbf{Event} \ \text{IsDuplicate} \ \text{(ordinary)} \ \overset{\Delta}{=} \ \text{refines} \ \text{IsDuplicate}
\text{any}

\text{any}

\text{where}

\text{grd}\_self: \ i \notin \text{IsDuplicate}
\text{grd1}\_xor: \ i \notin \text{NotDuplicate}
\text{grd}\_seq\_{Duplicate}: \ i \in \text{GatherInfo}

\text{then}

\text{act}: \text{IsDuplicate} := \text{IsDuplicate} \cup \{i\}

\text{end}
Appendix A Event-B Model of the Fire Dispatch Workflow

**Event** CloseDuplicate (ordinary) ≜

**refines** CloseDuplicate

any

i

where

grd_self: i ∉ CloseDuplicate
grd_seq Duplicate: i ∈ IsDuplicate

then

act: CloseDuplicate := CloseDuplicate ∪ {i}

end

**Event** NotDuplicate (ordinary) ≜

**refines** NotDuplicate

any

i

where

grd_self: i ∉ NotDuplicate
grd1_xor: i ∉ IsDuplicate
grd_seq NonDuplicate: i ∈ GatherInfo

then

act: NotDuplicate := NotDuplicate ∪ {i}

end

**Event** SetInitialAP (ordinary) ≜

**refines** SetInitialAP

any

i

lrs

where

grd_self: i ∉ SetInitialAP
grd1: lrs ⊆ LR
grd2: lrs ≠ ∅
grd_seq NonDuplicate: i ∈ NotDuplicate

then

act: SetInitialAP := SetInitialAP ∪ {i}
act1: ap := ap ∪ ({i} × lrs)

manually

end

**Event** SendStopMsg (ordinary) ≜

**refines** UpdateActionPlan

any

i
Appendix A Event-B Model of the Fire Dispatch Workflow

\begin{verbatim}
where

\text{grd\_self:} \quad i \notin \text{SendStopMsg}
\text{grd1:} \quad \text{lrs} \subseteq \text{ap}[\{i\}]
    //added manually
\text{grd\_seq:} \quad i \in \text{SetInitialAP}

\text{then}

\text{act:} \quad \text{SendStopMsg} := \text{SendStopMsg} \cup \{i\}
\text{act1:} \quad \text{notRequired\_ap} := \text{notRequired\_ap} \cup (\{i\} \times \text{lrs})
    \quad \text{added manually}
\end{verbatim}

\textbf{Event} IncidentManaged (\text{ordinary}) \triangleq 
\textbf{refines} \quad \text{ManageIncident}

\begin{verbatim}
any
\text{i}

\text{where}

\text{grd\_self:} \quad i \notin \text{IncidentManaged}
\text{grd\_seq:} \quad i \in \text{SetInitialAP}

\text{then}

\text{act:} \quad \text{IncidentManaged} := \text{IncidentManaged} \cup \{i\}
\end{verbatim}

\textbf{Event} CloseIncident (\text{ordinary}) \triangleq 
\textbf{refines} \quad \text{CloseIncident}

\begin{verbatim}
any
\text{i}

\text{where}

\text{grd\_seq:} \quad (i \in \text{SendStopMsg} \land i \in \text{IncidentManaged})
\text{grd\_self:} \quad i \notin \text{CloseIncident}

\text{then}

\text{act:} \quad \text{CloseIncident} := \text{CloseIncident} \cup \{i\}
\end{verbatim}

END

A.5 Fourth Refinement

A.5.1 Context C2

\textbf{CONTEXT} \quad C2
\textbf{EXTENDS} \quad C1
\textbf{SETS}
INCIDENT\_TYPE
LOCATION
LOCATION\_TYPE

CONSTANTS

PDA set the action plan according to the type and location

AXIOMS

axm1: \( PDA \in (INCIDENT\_TYPE \times LOCATION\_TYPE) \rightarrow \mathcal{P}(LR) \)

axm2: \( \forall it, loc : it \in INCIDENT\_TYPE \land loc \in LOCATION\_TYPE \Rightarrow PDA(it \mapsto loc) \neq \emptyset \)

END

A.5.2 ERS Diagram: M4
Figure A.5: ERS Diagram at Fourth Refinement: M4
A.5.3 Machine M4

**MACHINE** M4  
**REFINES** M3  
**SEES** C2

**VARIABLES**
- CreateIncident
- IsDuplicate
- CloseDuplicate
- NotDuplicate
- SetInitialAP
- SelectType
- SelectLocation
- InfoComp
- RequestAdditionalRes
- SendStopMsg
- HandleIncident
- IncidentManaged
- CloseIncident
- iType added manually
- ap added manually
- iLocation added manually
- iLocType added manually
- notRequired_ap added manually

**INVARINTS**

\[
\begin{align*}
\text{inv\_CreateIncident\_type} & : \text{CreateIncident} \subseteq \text{INCIDENT} \\
\text{inv\_RequestAdditionalRes\_type} & : \text{RequestAdditionalRes} \subseteq \text{INCIDENT} \times \text{LR} \\
\text{inv\_HandleIncident\_type} & : \text{HandleIncident} \subseteq \text{INCIDENT} \times \text{LR} \\
\text{inv\_SelectType\_seq} & : \text{SelectType} \subseteq \text{CreateIncident} \\
\text{inv\_SelectLocation\_seq} & : \text{SelectLocation} \subseteq \text{CreateIncident} \\
\text{inv\_InfoComp\_seq} & : \text{InfoComp} \subseteq (\text{SelectType} \cap \text{SelectLocation}) \\
\text{inv\_InfoComp\_glu} & : \text{InfoComp} = \text{GatherInfo} \\
\text{inv\_IsDuplicate\_Duplicate\_seq} & : \text{IsDuplicate} \subseteq \text{InfoComp} \\
\text{inv\_CloseDuplicate\_Duplicate\_seq} & : \text{CloseDuplicate} \subseteq \text{IsDuplicate} \\
\text{inv\_NotDuplicate\_NonDuplicate\_seq} & : \text{NotDuplicate} \subseteq \text{InfoComp} \\
\text{inv\_SetInitialAP\_NonDuplicate\_seq} & : \text{SetInitialAP} \subseteq \text{NotDuplicate} \\
\text{inv\_RequestAdditionalRes\_seq} & : \text{dom(} \text{RequestAdditionalRes}\text{)} \subseteq \text{SetInitialAP} \\
\text{inv\_SendStopMsg\_seq} & : \text{SendStopMsg} \subseteq \text{SetInitialAP}
\end{align*}
\]
inv\_HandleIncident\_seq: dom(HandleIncident) \subseteq SetInitialAP

inv\_IncidentManaged\_seq: IncidentManaged \subseteq SetInitialAP

inv\_CloseIncident\_seq: CloseIncident \subseteq (SendStopMsg \cap IncidentManaged)

inv\_xorfd0: partition((IsDuplicate \cup NotDuplicate), IsDuplicate, NotDuplicate)

inv1: \ iType \in SelectType \rightarrow INCIDENT\_TYPE

added manually

inv2: \ iLocation \in SelectLocation \rightarrow LOCATION

added manually

inv\_loc\_type: \ iLocType \in SelectLocation \rightarrow LOCATION\_TYPE

data variable

inv3: \ \forall i \in IsDuplicate \Rightarrow (\exists i0 \in NotDuplicate \land iLocation(i) = iLocation(i0))

added manually

in\_Handle\_Rep: \ \forall i \in INCIDENT \Rightarrow HandleIncident[i] \subseteq ap[i]

inv\_RequestAdditionalRes\_rep: \ \forall i \in INCIDENT \Rightarrow RequestAdditionalRes[i] \subseteq LR

EVENTS

Initialisation

begin

act\_CreateIncident: CreateIncident := \emptyset

act\_SelectType: SelectType := \emptyset

act\_SelectLocation: SelectLocation := \emptyset

act\_InfoComp: InfoComp := \emptyset

act\_IsDuplicate: IsDuplicate := \emptyset

act\_CloseDuplicate: CloseDuplicate := \emptyset

act\_NotDuplicate: NotDuplicate := \emptyset

act\_SetInitialAP: SetInitialAP := \emptyset

act\_RequestAdditionalRes: RequestAdditionalRes := \emptyset

act\_SendStopMsg: SendStopMsg := \emptyset

act\_HandleIncident: HandleIncident := \emptyset

act\_IncidentManaged: IncidentManaged := \emptyset

act\_CloseIncident: CloseIncident := \emptyset

act1: ap := \emptyset

act2: iLocation := \emptyset

act\_iLocType: iLocType := \emptyset

act3: iType := \emptyset

act\_notRequired\_ap: notRequired\_ap := \emptyset

end

Event SelectType \langle ordinary \rangle \equiv

any

i
where
  grd_seq: i ∈ CreateIncident
  grd_self: i /∈ SelectType
  grd1: t ∈ INCIDENT_TYPE
then
  act: SelectType := SelectType ∪ \{i\}
  act1: iType(i) := t
end

Event SelectLocation (\textit{ordinary}) \triangleq

\begin{align*}
\text{any} & \\
& i \\
& l \\
& lt
\end{align*}

where
  grd_seq: i ∈ CreateIncident
  grd_self: i /∈ SelectLocation
  grd1: l ∈ LOCATION
  grd2: lt ∈ LOCATION_TYPE
then
  act: SelectLocation := SelectLocation ∪ \{i\}
  act1: iLocation(i) := l
  act2: iLocType(i) := lt
end

Event CreateIncident (\textit{ordinary}) \triangleq
refines CreateIncident

\begin{align*}
\text{any} & \\
& i
\end{align*}

where
  grd_self: i /∈ CreateIncident
then
  act: CreateIncident := CreateIncident ∪ \{i\}
end

Event InfoComp (\textit{ordinary}) \triangleq
refines GatherInfo

\begin{align*}
\text{any} & \\
& i
\end{align*}

where
  grd_seq: (i ∈ SelectType ∧ i ∈ SelectLocation)
  grd_self: i /∈ InfoComp
then
\textbf{act:} \( \text{InfoComp} := \text{InfoComp} \cup \{i\} \)

\textbf{end}

\textbf{Event} \ IsDuplicate \ (\text{ordinary}) \ \Rightarrow

\textbf{refines} \ IsDuplicate

\textbf{any}

\textbf{i}

\textbf{i0}

\textbf{where}

\textbf{grd\_seq\_Duplicate:} \ i \in \text{InfoComp}

\textbf{grd\_self:} \ i \notin \text{IsDuplicate}

\textbf{grd1\_xor:} \ i \notin \text{NotDuplicate}

\textbf{grd\_i0:} \ i0 \in \text{NotDuplicate} \setminus \text{CloseIncident}

\text{added manually}

\textbf{grd\_i0i:} \ i\text{Location}(i) = i\text{Location}(i0)

\textbf{then}

\textbf{act:} \ IsDuplicate := \text{IsDuplicate} \cup \{i\}

\textbf{end}

\textbf{Event} \ CloseDuplicate \ (\text{ordinary}) \ \Rightarrow

\textbf{refines} \ CloseDuplicate

\textbf{any}

\textbf{i}

\textbf{where}

\textbf{grd\_seq\_Duplicate:} \ i \in \text{IsDuplicate}

\textbf{grd\_self:} \ i \notin \text{CloseDuplicate}

\textbf{then}

\textbf{act:} \ CloseDuplicate := \text{CloseDuplicate} \cup \{i\}

\textbf{end}

\textbf{Event} \ NotDuplicate \ (\text{ordinary}) \ \Rightarrow

\textbf{refines} \ NotDuplicate

\textbf{any}

\textbf{i}

\textbf{where}

\textbf{grd\_seq\_NonDuplicate:} \ i \in \text{InfoComp}

\textbf{grd\_self:} \ i \notin \text{NotDuplicate}

\textbf{grd1\_xor:} \ i \notin \text{IsDuplicate}

\textbf{then}

\textbf{act:} \ NotDuplicate := \text{NotDuplicate} \cup \{i\}

\textbf{end}

\textbf{Event} \ SetInitialAP \ (\text{ordinary}) \ \Rightarrow

\textbf{refines} \ SetInitialAP
any
lrs
i
l
t

where

\text{grd\_seq\_NonDuplicate}: \ i \in \text{NotDuplicate}
\text{grd\_self}: \ i \notin \text{SetInitialAP}
\text{grd1}: \ t = \text{iType}(i)
\text{grd2}: \ l = \text{iLocType}(i)
\text{grd3}: \ lrs = \text{PDA}(t \mapsto l)

\then
\text{act}: \ \text{SetInitialAP} := \text{SetInitialAP} \cup \{i\}
\text{act1}: \ \text{ap} := \text{ap} \cup \{i \times lrs\}

\text{manually}

\end

\text{Event RequestAdditionalRes (ordinary) \refines RequestAdditionalRes}

any
i
lr

where

\text{grd\_seq}: \ i \in \text{SetInitialAP}
\text{grd\_self}: \ i \mapsto lr \notin \text{RequestAdditionalRes}
\text{grd\_InputExpression1}: \ lr \in (\text{LR} \setminus \text{ap}[\{i\}])
\text{grd0\_par}: \ i \notin \text{SendStopMsg}

\then
\text{act}: \ \text{RequestAdditionalRes} := \text{RequestAdditionalRes} \cup \{i \mapsto lr\}
\text{act1}: \ \text{ap} := \text{ap} \cup \{i \mapsto lr\}

//\text{manually added}

\end

\text{Event SendStopMsg (ordinary) \refines SendStopMsg}

any
i
lrs

where

\text{grd1}: \ lrs \subseteq \text{ap}[\{i\}]

//\text{added manually}
\text{grd\_seq}: \ i \in \text{SetInitialAP}
\text{grd\_self}: \ i \notin \text{SendStopMsg}
then
  act:  \( \text{SendStopMsg} := \text{SendStopMsg} \cup \{i\} \)
  act1:  \( \text{notRequired}_{\text{ap}} := \text{notRequired}_{\text{ap}} \cup (\{i\} \times \text{lrs}) \)
        added manually
end

Event  HandleIncident  \( \langle \text{ordinary} \rangle \)  \( \triangleq \)
refines  HandleIncident
  any
    i
    lr
  where
    grd_{0\_par}:  i \notin \text{IncidentManaged}
    grd_{seq} :  i \in \text{SetInitialAP}
    grd_{self} :  i \mapsto lr \notin \text{HandleIncident}
    grd_{InputExpression1} :  lr \in \text{ap}[\{i\}]
  then
    act:  \text{HandleIncident} := \text{HandleIncident} \cup \{i \mapsto lr\}
end

Event  IncidentManaged  \( \langle \text{ordinary} \rangle \)  \( \triangleq \)
refines  IncidentManaged
  any
    i
  where
    grd_{self} :  i \notin \text{IncidentManaged}
    grd_{seq} :  i \in \text{SetInitialAP}
  then
    act:  \text{IncidentManaged} := \text{IncidentManaged} \cup \{i\}
end

Event  CloseIncident  \( \langle \text{ordinary} \rangle \)  \( \triangleq \)
refines  CloseIncident
  any
    i
  where
    grd_{seq} :  (i \in \text{SendStopMsg} \wedge i \in \text{IncidentManaged})
    grd_{self} :  i \notin \text{CloseIncident}
  then
    act:  \text{CloseIncident} := \text{CloseIncident} \cup \{i\}
end

END
A.6  Fifth Refinement

A.6.1  ERS Diagram: M5
Figure A.6: ERS Diagram at Fifth Refinement Level: M5
A.6.2 Machine M5

**MACHINE** M5
**REFINES** M4
**SEES** C2

**VARIABLES**
- CreateIncident
- IsDuplicate
- CloseDuplicate
- NotDuplicate
- SetInitialAP
- SelectType
- SelectLocation
- InfoComp
- RequestAdditionalRes
- IncidentManaged
- CloseIncident
- SendStopMsg
- AllocateRes
- ResAttend
- DeallocateRes
- iType added manually
- ap added manually
- iLocation added manually
- iLocType added manually
- StopMsgReceived
- notRequired_ap

**INVARIANTS**

- \( \text{inv}_{\text{CreateIncident}} \): \( \text{CreateIncident} \subseteq \text{INCIDENT} \)
- \( \text{inv}_{\text{RequestAdditionalRes}} \): \( \text{RequestAdditionalRes} \subseteq \text{INCIDENT} \times LR \)
- \( \text{inv}_{\text{SelectType}} \): \( \text{SelectType} \subseteq \text{CreateIncident} \)
- \( \text{inv}_{\text{SelectLocation}} \): \( \text{SelectLocation} \subseteq \text{CreateIncident} \)
- \( \text{inv}_{\text{InfoComp}} \): \( \text{InfoComp} \subseteq (\text{SelectType} \cap \text{SelectLocation}) \)
- \( \text{inv}_{\text{IsDuplicate}} \): \( \text{IsDuplicate} \subseteq \text{InfoComp} \)
- \( \text{inv}_{\text{CloseDuplicate}} \): \( \text{CloseDuplicate} \subseteq \text{IsDuplicate} \)
- \( \text{inv}_{\text{NotDuplicate}} \): \( \text{NotDuplicate} \subseteq \text{InfoComp} \)
- \( \text{inv}_{\text{SetInitialAP}} \): \( \text{SetInitialAP} \subseteq \text{NotDuplicate} \)
- \( \text{inv}_{\text{RequestAdditionalRes}} \): \( \text{dom}(\text{RequestAdditionalRes}) \subseteq \text{SetInitialAP} \)
\textbf{inv\_SendStopMsg\_seq}: \ SendStopMsg \subseteq \text{SetInitialAP}\\
\textbf{inv\_Allocate\_Res\_type}: \ AllocateRes \subseteq \text{INCIDENT} \times LR\\
\textbf{inv\_Allocate\_Res\_seq}: \ \text{dom}(AllocateRes) \subseteq \text{SetInitialAP}\\
\textbf{inv\_Res\_Attend\_seq}: \ ResAttend \subseteq AllocateRes\\
\textbf{inv\_Stop\_Msg\_Received\_seq}: \ StopMsgReceived \subseteq AllocateRes\\
\textbf{inv2\_xorFD}: \ \text{partition}((\text{ResAttend} \cup \text{StopMsgReceived}), \text{ResAttend}, \text{StopMsgReceived})\\
\textbf{inv\_Deallocate\_Res\_seq}: \ DeallocateRes \subseteq (\text{ResAttend} \cup \text{StopMsgReceived})\\
\textbf{inv\_Incident\_Managed\_seq}: \ IncidentManaged \subseteq \text{SetInitialAP}\\
\textbf{inv\_Close\_Incident\_seq}: \ CloseIncident \subseteq (\text{SendStopMsg} \cap \text{IncidentManaged})\\
\textbf{inv1\_xorfd0}: \ \text{partition}((\text{IsDuplicate} \cup \text{NotDuplicate}), \text{IsDuplicate}, \text{NotDuplicate})\\
\textbf{inv\_par\_ref}: \ \forall i \in \text{IncidentManaged} \Rightarrow \text{AllocateRes}[[i]] = \text{DeallocationRes}[[i]]\\
\textbf{inv\_alloc\_ap}: \ \forall i \in \text{INCIDENT} \Rightarrow \text{AllocateRes}[[i]] \subseteq \text{ap}[[i]]\\
\textbf{inv\_managed}: \ \forall i \in \text{CloseIncident} \Rightarrow \text{ap}[[i]] \not\text{notRequired\_ap}[[i]] \subseteq \text{DeallocationRes}[[i]]\\

manually added

**EVENTS**

**Initialisation**

begin

act\_Create\_Incident: \ CreateIncident := \emptyset \\
act\_Select\_Type: \ SelectType := \emptyset \\
act\_Select\_Location: \ SelectLocation := \emptyset \\
act\_Info\_Comp: \ InfoComp := \emptyset \\
act\_Is\_Duplicate: \ IsDuplicate := \emptyset \\
act\_Close\_Duplicate: \ CloseDuplicate := \emptyset \\
act\_Not\_Duplicate: \ NotDuplicate := \emptyset \\
act\_Set\_Initial\_AP: \ Set\_Initial\_AP := \emptyset \\
act\_Request\_Additional\_Res: \ Request\_Additional\_Res := \emptyset \\
act\_Send\_Stop\_Msg: \ Send\_Stop\_Msg := \emptyset \\
act\_Allocate\_Res: \ Allocate\_Res := \emptyset \\
act\_Res\_Attend: \ Res\_Attend := \emptyset \\
act\_Deallocate\_Res: \ Deallocate\_Res := \emptyset \\
act\_Incident\_Managed: \ Incident\_Managed := \emptyset \\
act\_Close\_Incident: \ Close\_Incident := \emptyset \\

act1: \ ap := \emptyset \\
act2: \ i\_Location := \emptyset \\
act\_i\_Loc\_Type: \ i\_Loc\_Type := \emptyset \\
act3: \ i\_Type := \emptyset \\
act4: \ not\_Required\_ap := \emptyset
Appendix A Event-B Model of the Fire Dispatch Workflow

\begin{verbatim}
act_StopMsgReceived: StopMsgReceived := Ø

Event AllocateRes (ordinary) ≜
  any
  i
  lr
  where
    grd_seq.: i ∈ SetInitialAP
    grd_self: i ↦ lr \notin AllocateRes
    grd_InputExpression1: lr ∈ ap[\{i\}]
    grd0_par: i \notin IncidentManaged
    grd1: i ↦ lr \notin notRequired_ap
  then
    act: AllocateRes := AllocateRes ∪ \{i ↦ lr\}
  end

Event UpdateResLocation (ordinary) ≜
  any
  i
  lr
  where
    grd_seq.: i ↦ lr ∈ AllocateRes
    grd0_loop: i ↦ lr \notin (ResAttend ∪ StopMsgReceived)
  then
    skip
  end

Event ReallocateQuickerRes (ordinary) ≜
  any
  i
  lr
  where
    grd_seq.: i ↦ lr ∈ AllocateRes
    grd0_loop: i ↦ lr \notin (ResAttend ∪ StopMsgReceived)
    grd1: i ↦ lr \notin notRequired_ap
  then
    skip
  end

Event ResAttend (ordinary) ≜
  any
  i
  lr
  where
\end{verbatim}
\textbf{Event} StopMsgReceived (ordinary) \(=\)
\begin{align*}
\text{any} & \\
\text{lr} & \\
i & \\
\text{where} & \\
grd\_seq: & i \mapsto lr \in \text{AllocateRes} \\
grd\_self: & i \mapsto lr \notin \text{ResAttend} \\
grd\_xor: & i \mapsto lr \notin \text{StopMsgReceived} \\
@\text{grd1} & i \mapsto lr \notin \text{notRequired\_ap} \text{ not necessary because it could be added but haven’t received msg yet} \\
\text{then} & \\
\text{act: } & \text{ResAttend} := \text{ResAttend} \cup \{ i \mapsto lr \} \\
\text{end} \\
\end{align*}

\textbf{Event} CreateIncident (ordinary) \(=\)
\begin{align*}
\text{refines } & \text{CreateIncident} \\
\text{any} & \\
i & \\
\text{where} & \\
grd\_self: & i \notin \text{CreateIncident} \\
\text{then} & \\
\text{act: } & \text{CreateIncident} := \text{CreateIncident} \cup \{ i \} \\
\text{end} \\
\end{align*}

\textbf{Event} SelectType (ordinary) \(=\)
\begin{align*}
\text{refines } & \text{SelectType} \\
\text{any} & \\
t & \\
i & \\
\text{where} & \\
grd\_seq: & i \in \text{CreateIncident} \\
grd\_self: & i \notin \text{SelectType} \\
grd1: & t \in \text{INCIDENT\_TYPE} \\
\text{then} & \\
\text{act: } & \text{SelectType} := \text{SelectType} \cup \{ i \} \\
\end{align*}
\textbf{act1}: \textit{iType}(i) := t

\textbf{end}

\textbf{Event} \ SelectLocation \ (\textit{ordinary}) \ \triangleright

\textbf{refines} \ SelectLocation

\hspace{1em} \textbf{any}

\hspace{2em} i

\hspace{2em} l

\hspace{2em} lt

\hspace{1.5em} \textbf{where}

\hspace{2.5em} \textit{grd\_seq}.: \ i \in \text{CreateIncident}

\hspace{2.5em} \textit{grd\_self}.: \ i \notin \text{SelectLocation}

\hspace{2.5em} \textit{grd1}.: \ l \in \textit{LOCATION}

\hspace{2.5em} \textit{grd2}.: \ lt \in \textit{LOCATION\_TYPE}

\hspace{1.5em} \textbf{then}

\hspace{2.5em} \textbf{act}: \ SelectLocation := \text{SelectLocation} \cup \{i\}

\hspace{2.5em} \textbf{act1}: \ iLocation(i) := l

\hspace{2.5em} \textbf{act2}: \ iLocType(i) := lt

\hspace{1em} \textbf{end}

\textbf{Event} \ InfoComp \ (\textit{ordinary}) \ \triangleright

\textbf{refines} \ InfoComp

\hspace{1em} \textbf{any}

\hspace{2em} i

\hspace{1.5em} \textbf{where}

\hspace{2.5em} \textit{grd\_seq}.: \ (i \in \text{SelectType} \land i \in \text{SelectLocation})

\hspace{2.5em} \textit{grd\_self}.: \ i \notin \text{InfoComp}

\hspace{1.5em} \textbf{then}

\hspace{2.5em} \textbf{act}: \ InfoComp := \text{InfoComp} \cup \{i\}

\hspace{1em} \textbf{end}

\textbf{Event} \ IsDuplicate \ (\textit{ordinary}) \ \triangleright

\textbf{refines} \ IsDuplicate

\hspace{1em} \textbf{any}

\hspace{2em} i

\hspace{2em} i0

\hspace{1.5em} \textbf{where}

\hspace{2.5em} \textit{grd\_seq\_Duplicate}: \ i \in \text{InfoComp}

\hspace{2.5em} \textit{grd\_self}: \ i \notin \text{IsDuplicate}

\hspace{2.5em} \textit{grd1\_xor}: \ i \notin \text{NotDuplicate}

\hspace{2.5em} \textit{grd\_i0\_i}: \ i0 \in \text{NotDuplicate} \setminus \text{CloseIncident}

\hspace{3em} \text{added manually}

\hspace{2.5em} \textit{grd\_i0i}: \ iLocation(i) = iLocation(i0)
then

act: IsDuplicate := IsDuplicate ∪ \{i\}

end

Event CloseDuplicate (ordinary) ≜

refines CloseDuplicate

any

i

where

grd_seq_Duplicate: \ i \in IsDuplicate
grd_self: \ i \notin CloseDuplicate

then

act: CloseDuplicate := CloseDuplicate ∪ \{i\}

end

Event NotDuplicate (ordinary) ≜

refines NotDuplicate

any

i

where

grd_seq_NonDuplicate: \ i \in InfoComp
grd1_xor: \ i \notin IsDuplicate
grd_self: \ i \notin NotDuplicate

then

act: NotDuplicate := NotDuplicate ∪ \{i\}

end

Event SetInitialAP (ordinary) ≜

refines SetInitialAP

any

i

lrs

l

t

where

grd3: \ lrs = PDA(t \mapsto l)
grd_seq_NonDuplicate: \ i \in NotDuplicate
grd_self: \ i \notin SetInitialAP
grd1: \ t = iType(i)
grd2: \ l = iLocType(i)

then

act: SetInitialAP := SetInitialAP ∪ \{i\}
act1: \ ap := ap ∪ (\{i\} \times lrs)

manually
end

**Event** RequestAdditionalRes *(ordinary)* \( \triangleq \)

reﬁnes RequestAdditionalRes

any

\[ i \]

\[ lr \]

where

\[ \text{grd}_\text{seq} : i \in \text{SetInitialAP} \]

\[ \text{grd}_\text{self} : i \mapsto lr \notin \text{RequestAdditionalRes} \]

\[ \text{grd}_\text{InputExpression} : lr \in (LR \setminus ap[{i}]) \]

\[ \text{grd}_\text{0_par} : i \notin \text{SendStopMsg} \]

then

\[ \text{act} : \text{RequestAdditionalRes} := \text{RequestAdditionalRes} \cup \{i \mapsto lr\} \]

\[ \text{act1} : ap := ap \cup \{i \mapsto lr\} \]

//manually added

end

**Event** SendStopMsg *(ordinary)* \( \triangleq \)

reﬁnes SendStopMsg

any

\[ i \]

\[ lrs \]

where

\[ \text{grd}_\text{seq,planF} : i \in \text{SetInitialAP} \]

\[ \text{grd}_\text{self} : i \notin \text{SendStopMsg} \]

\[ \text{grd}_\text{DuplicateInterrupt} : i \notin \text{IsDuplicate} \]

\[ \text{grd}_\text{lrs} : lrs = ap[{i}] \setminus \text{ResAttend}{[i]} \]

then

\[ \text{act} : \text{SendStopMsg} := \text{SendStopMsg} \cup \{i\} \]

\[ \text{act1} : \text{notRequired}_\text{ap} := \text{notRequired}_\text{ap} \cup (\{i\} \times lrs) \]

end

**Event** DeallocateRes *(ordinary)* \( \triangleq \)

reﬁnes HandleIncident

any

\[ i \]

\[ lr \]

where

\[ \text{grd}_\text{self} : i \mapsto lr \notin \text{DeallocateRes} \]

\[ \text{grd}_\text{seq} : i \mapsto lr \in (\text{ResAttend} \cup \text{StopMsgReceived}) \]

then

\[ \text{act} : \text{DeallocateRes} := \text{DeallocateRes} \cup \{i \mapsto lr\} \]
Appendix A Event-B Model of the Fire Dispatch Workflow

end

Event IncidentManaged (ordinary) ≜
refines IncidentManaged

any
i

where

grd_par: \( \text{AllocateRes}\{i\} = \text{DeallocateRes}\{i\} \)
grd_seq: \( i \in \text{SetInitialAP} \)
grd_self: \( i \notin \text{IncidentManaged} \)

then

act: \( \text{IncidentManaged} := \text{IncidentManaged} \cup \{i\} \)

end

Event CloseIncident (ordinary) ≜
refines CloseIncident

any
i

where

grd_seq: \( (i \in \text{SendStopMsg} \land i \in \text{IncidentManaged}) \)
grd_self: \( i \notin \text{CloseIncident} \)
gd1: \( \text{ap}\{i\} \setminus \text{notRequired}\text{ap}\{i\} \subseteq \text{DeallocateRes}\{i\} \)

to ensure all action plan items are satisfied

then

act: \( \text{CloseIncident} := \text{CloseIncident} \cup \{i\} \)

end

END

A.7 Sixth Refinement

A.7.1 Context C3

CONTEXT C3
EXTENDS C2
SETS

\( R_{\text{TYPE}} \) type of the resource

PR set of physical resources

CONSTANTS

\( \text{ltype} \)

\( \text{ptype} \)

AXIOMS
Appendix A Event-B Model of the Fire Dispatch Workflow

\[ axm1: \quad ltype \in LR \rightarrow R_{TYPE} \]
logical type of a resource

\[ axm2: \quad ptype \in PR \rightarrow R_{TYPE} \]
physical type of a resource

\[ axm3: \quad \forall lr \cdot lr \in LR \Rightarrow (\exists pr \cdot ptype(pr) = ltype(lr)) \]

END

A.7.2 Machine M6

MACHINE M6

REFINES M5

SEES C3

VARIABLES

CreateIncident
IsDuplicate
CloseDuplicate
NotDuplicate
SetInitialAP
SelectType
SelectLocation
InfoComp
RequestAdditionalRes
IncidentManaged
CloseIncident
SendStopMsg
AllocateRes
ResAttend
DeallocateRes
iType added manually
ap added manually
iLocation
iLocType added manually
alloc manually
alloc_LR manually
StopMsgReceived
notRequired_ap

INVARIANTS
inv_alloc_type: alloc ∈ PR → INCIDENT manually
inv_alloc_ran: ran(alloc) ⊆ SetInitialAP manually
inv_alloc_LR_type: alloc_LR ∈ PR → LR manually
inv_alloc_LR_PR_Typ: ∀ pr, lr · pr ∈ dom(alloc_LR) ∧ alloc_LR(pr) = lr ⇒ ptype(pr) = ltype(lr)
inv_alloc_alloc_LR: dom(alloc) = dom(alloc_LR)
inv_Close_Incident_alloc: ∀ i · i ∈ (CloseIncident ∪ CloseDuplicate) ⇒ i ∉ ran(alloc)

EVENTS
Initialisation

begin
act_CreateIncident: CreateIncident := ∅
act_SelectType: SelectType := ∅
act_SelectLocation: SelectLocation := ∅
act_InfoComp: InfoComp := ∅
act_IsDuplicate: IsDuplicate := ∅
act_CloseDuplicate: CloseDuplicate := ∅
act_NotDuplicate: NotDuplicate := ∅
act_SetInitialAP: SetInitialAP := ∅
act_RequestAdditionalRes: RequestAdditionalRes := ∅
act_SendStopMsg: SendStopMsg := ∅
act_AllocateRes: AllocateRes := ∅
act_ResAttend: ResAttend := ∅
act_DeallocateRes: DeallocateRes := ∅
act_IncidentManaged: IncidentManaged := ∅
act_CloseIncident: CloseIncident := ∅
act1: ap := ∅
act2: iLocation := ∅
act_iLocType: iLocType := ∅
act3: iType := ∅
act_alloc: alloc := ∅
act_alloc_LR: alloc_LR := ∅
act_notRequired: notRequired_ap := ∅
act_StopMsgReceived: StopMsgReceived := ∅
end
Event AllocateRes (ordinary)
refines AllocateRes

any
i
lr
pr

where

grd_seq: \( i \in \text{SetInitialAP} \)
grd_self: \( i \mapsto lr \notin \text{AllocateRes} \)
grd_InputExpression1: \( lr \in \text{ap}[i] \)
grd0_par: \( i \notin \text{IncidentManaged} \)
grd1: \( pr \notin \text{dom}(\text{alloc}) \)
  added manually
grd2: \( ptype(pr) = ltype(lr) \)
  added manually
grd11: \( i \mapsto lr \notin \text{notRequired}_\text{ap} \)

then

act: \( \text{AllocateRes} := \text{AllocateRes} \cup \{ i \mapsto lr \} \)
act1: \( \text{alloc}(pr) := i \)
  added manually
act2: \( \text{alloc}_\text{LR}(pr) := lr \)
  added manually

end

Event UpdateResLocation (ordinary) ≜

refines UpdateResLocation

any
i
lr

where

grd_seq: \( i \mapsto lr \in \text{AllocateRes} \)
grd0_loop: \( i \mapsto lr \notin (\text{ResAttend} \cup \text{StopMsgReceived}) \)

then

skip

end

Event ReallocateQuickerRes (ordinary) ≜

refines ReallocateQuickerRes

any
i
lr
pr
pr0

where
Appendix A Event-B Model of the Fire Dispatch Workflow

\textbf{grd\_seq:} \ i \mapsto lr \in \text{AllocateRes}

\textbf{grd0\_loop:} \ i \mapsto lr \notin (\text{ResAttend} \cup \text{StopMsgReceived})

\textbf{grd1:} \ pr0 \in (\text{alloc}^{-1}[\{i\}] \cap \text{alloc\_LR}^{-1}[\{lr\}])

\hspace{1cm} \text{added manually}

\textbf{grd2:} \ pr \notin \text{dom(alloc)}

\hspace{1cm} \text{manually}

\textbf{grd3:} \ \text{ptype}(pr) = \text{ltype}(lr)

\hspace{1cm} \text{manually}

\textbf{grd11:} \ i \mapsto lr \notin \text{notRequired\_ap}

\hspace{1cm} \text{manually}

\textbf{then}

\textbf{act1:} \ \text{alloc} := (\{pr0\} \leftarrow \text{alloc}) \cup \{pr \mapsto i\}

\textbf{act2:} \ \text{alloc\_LR} := (\{pr0\} \leftarrow \text{alloc\_LR}) \cup \{pr \mapsto lr\}

\textbf{end}

\textbf{Event} \ \text{ResAttend} \ (\text{ordinary}) \ \overset{\Delta}{=} \ \text{refines} \ \text{ResAttend}

\textbf{any}

\hspace{1cm} i

\hspace{1cm} lr

\textbf{where}

\textbf{grd\_self:} \ i \mapsto lr \notin \text{ResAttend}

\textbf{grd\_seq:} \ i \mapsto lr \in \text{AllocateRes}

\textbf{grd\_xor:} \ i \mapsto lr \notin \text{StopMsgReceived}

\hspace{1cm} \text{@grd1} \ i \mapsto lr \notin \text{notRequired\_ap} \text{ not necessary because it could be added but}

\hspace{1cm} \text{haven’t received msg yet}

\textbf{then}

\textbf{act:} \ \text{ResAttend} := \text{ResAttend} \cup \{i \mapsto lr\}

\textbf{end}

\textbf{Event} \ \text{CreateIncident} \ (\text{ordinary}) \ \overset{\Delta}{=} \ \text{refines} \ \text{CreateIncident}

\textbf{any}

\hspace{1cm} i

\textbf{where}

\textbf{grd\_self:} \ i \notin \text{CreateIncident}

\textbf{then}

\textbf{act:} \ \text{CreateIncident} := \text{CreateIncident} \cup \{i\}

\textbf{end}

\textbf{Event} \ \text{SelectType} \ (\text{ordinary}) \ \overset{\Delta}{=} \ \text{refines} \ \text{SelectType}

\textbf{any}
t
i

where
  grd_self:  i $\notin$ SelectType
  grd1:  t $\in$ INCIDENT_TYPE
  grd_seq:  i $\in$ CreateIncident

then
  act: SelectType := SelectType $\cup$ \{i\}
  act1: iType(i) := t

end

Event  SelectLocation (ordinary) $\sqsubseteq$
refines  SelectLocation

any
  i
  l
  lt

where
  grd_seq:  i $\in$ CreateIncident
  grd_self:  i $\notin$ SelectLocation
  grd1:  l $\in$ LOCATION
  grd2:  lt $\in$ LOCATION_TYPE

then
  act: SelectLocation := SelectLocation $\cup$ \{i\}
  act1: iLocation(i) := l
  act2: iLocType(i) := lt

end

Event  InfoComp (ordinary) $\sqsubseteq$
refines  InfoComp

any
  i

where
  grd_self:  i $\notin$ InfoComp
  grd_seq:  (i $\in$ SelectType $\land$ i $\in$ SelectLocation)

then
  act: InfoComp := InfoComp $\cup$ \{i\}

end

Event  IsDuplicate (ordinary) $\sqsubseteq$
refines  IsDuplicate

any
  i0
Appendix A Event-B Model of the Fire Dispatch Workflow

where

\( i \)

\( \text{where} \\ \\
\text{grd}\_\text{self}: \ i \notin Is\text{Duplicate} \\
\text{grd}\_\text{xor}: \ i \notin Not\text{Duplicate} \\
\text{grd}\_\text{i0}: \ i0 \in Not\text{Duplicate} \setminus Close\text{Incident} \\
\quad \text{manually} \\
\text{grd}\_\text{seq}\_\text{Duplicate}: \ i \in Info\text{Comp} \\
\text{grd}\_\text{i0i}: \ i\text{Location}(i) = i\text{Location}(i0) \\
\text{then} \\
\text{act}: \ Is\text{Duplicate} := Is\text{Duplicate} \cup \{i\} \\
\text{end} \\
\text{Event} \ \text{CloseDuplicate} \langle\text{ordinary}\rangle \equiv \\
\text{refines} \ \text{CloseDuplicate} \\
\quad \text{any} \\
\quad \ \ i \\
\text{where} \\
\text{grd}\_\text{self}: \ i \notin Close\text{Duplicate} \\
\text{grd}\_\text{seq}\_\text{Duplicate}: \ i \in Is\text{Duplicate} \\
\text{then} \\
\text{act}: \ Close\text{Duplicate} := Close\text{Duplicate} \cup \{i\} \\
\text{end} \\
\text{Event} \ \text{NotDuplicate} \langle\text{ordinary}\rangle \equiv \\
\text{refines} \ \text{NotDuplicate} \\
\quad \text{any} \\
\quad \ \ i \\
\text{where} \\
\text{grd}\_\text{xor}: \ i \notin Is\text{Duplicate} \\
\text{grd}\_\text{self}: \ i \notin Not\text{Duplicate} \\
\text{grd}\_\text{seq}\_\text{NonDuplicate}: \ i \in Info\text{Comp} \\
\text{then} \\
\text{act}: \ Not\text{Duplicate} := Not\text{Duplicate} \cup \{i\} \\
\text{end} \\
\text{Event} \ \text{SetInitialAP} \langle\text{ordinary}\rangle \equiv \\
\text{refines} \ \text{SetInitialAP} \\
\quad \text{any} \\
\quad \ \ i \\
\quad \ \ \text{lrs} \\
\quad \ \ \text{l} \\
\quad \ \ \text{t} \\
\text{where}
Appendix A Event-B Model of the Fire Dispatch Workflow

\[\begin{align*}
grd\_self: & \quad i \notin Set\_Initial\_AP \\
grd\_seq\_NonDuplicate: & \quad i \in Not\_Duplicate \\
grd1: & \quad t = iType(i) \\
grd2: & \quad l = iLocType(i) \\
grd3: & \quad lrs = PDA(t \mapsto l)
\end{align*}\]

\textbf{then}

\[\begin{align*}
act: & \quad Set\_Initial\_AP := Set\_Initial\_AP \cup \{i\} \\
act1: & \quad ap := ap \cup ([i] \times lrs) \\
\text{manually added}
\end{align*}\]

\textbf{end}

\textbf{Event} \ Request\_Additional\_Res \ (\textit{ordinary}) \ \overset{\text{refines}}{=} \ Request\_Additional\_Res

\textbf{any}

\[\begin{align*}
i \\
lr
\end{align*}\]

\textbf{where}

\[\begin{align*}
grd\_self: & \quad i \mapsto lr \notin Request\_Additional\_Res \\
grd\_Input\_Expression1: & \quad lr \in (LR \setminus ap([i])) \\
grd0\_par: & \quad i \notin Send\_Stop\_Msg \\
grd\_seq: & \quad i \in Set\_Initial\_AP
\end{align*}\]

\textbf{then}

\[\begin{align*}
act: & \quad Request\_Additional\_Res := Request\_Additional\_Res \cup \{i \mapsto lr\} \\
act1: & \quad ap := ap \cup \{i \mapsto lr\} \\
\text{manually added}
\end{align*}\]

\textbf{end}

\textbf{Event} \ Send\_Stop\_Msg \ (\textit{ordinary}) \ \overset{\text{refines}}{=} \ Send\_Stop\_Msg

\textbf{any}

\[\begin{align*}
i \\
lrs
\end{align*}\]

\textbf{where}

\[\begin{align*}
grd\_seq\_planF: & \quad i \in Set\_Initial\_AP \\
grd\_self: & \quad i \notin Send\_Stop\_Msg \\
grd\_Duplicate\_Interrupt: & \quad i \notin Is\_Duplicate \\
grd\_lrs: & \quad lrs = ap([i]) \setminus Res\_Attend([i])
\end{align*}\]

\textbf{then}

\[\begin{align*}
act: & \quad Send\_Stop\_Msg := Send\_Stop\_Msg \cup \{i\} \\
act1: & \quad not\_Required\_ap := not\_Required\_ap \cup ([i] \times lrs)
\end{align*}\]

\textbf{end}

\textbf{Event} \ Stop\_Msg\_Received \ (\textit{ordinary}) \ \overset{\text{refines}}{=} \ Stop\_Msg\_Received

\textbf{any}

\[\begin{align*}
i \\
lrs
\end{align*}\]

\textbf{where}

\[\begin{align*}
grd\_seq\_planF: & \quad i \in Set\_Initial\_AP \\
grd\_self: & \quad i \notin Send\_Stop\_Msg \\
grd\_Duplicate\_Interrupt: & \quad i \notin Is\_Duplicate \\
grd\_lrs: & \quad lrs = ap([i]) \setminus Res\_Attend([i])
\end{align*}\]

\textbf{then}

\[\begin{align*}
act: & \quad Send\_Stop\_Msg := Send\_Stop\_Msg \cup \{i\} \\
act1: & \quad not\_Required\_ap := not\_Required\_ap \cup ([i] \times lrs)
\end{align*}\]

\textbf{end}
any
lr
i

where

\texttt{grd\_seq}: i \mapsto lr \in \texttt{AllocateRes}
\texttt{grd\_self}: i \mapsto lr \notin \texttt{StopMsgReceived}
\texttt{grd\_xor}: i \mapsto lr \notin \texttt{ResAttend}
\texttt{grd1}: i \mapsto lr \in \texttt{notRequired}\_ap

in this case it is added because it can not be received unless not required

then

\texttt{act}: \texttt{StopMsgReceived} := \texttt{StopMsgReceived} \cup \{i \mapsto lr\}

end

\textbf{Event} \texttt{DeallocateRes} (ordinary) \refines

\textbf{refines} \texttt{DeallocateRes}

any
i
lr
pr

where

\texttt{grd\_seq}: i \mapsto lr \in (\texttt{ResAttend} \cup \texttt{StopMsgReceived})
\texttt{grd1}: pr \in (\texttt{alloc}^{-1}[\{i\}] \cap \texttt{alloc\_LR}^{-1}[\{lr\}])
\texttt{grd\_self}: i \mapsto lr \notin \texttt{DeallocateRes}

then

\texttt{act1}: \texttt{alloc} := \{pr\} \triangleleft \texttt{alloc}
\texttt{act2}: \texttt{alloc\_LR} := \{pr\} \triangleleft \texttt{alloc\_LR}
\texttt{act}: \texttt{DeallocateRes} := \texttt{DeallocateRes} \cup \{i \mapsto lr\}

end

\textbf{Event} \texttt{IncidentManaged} (ordinary) \refines

\textbf{refines} \texttt{IncidentManaged}

any
i

where

\texttt{grd\_seq}: i \in \texttt{SetInitialAP}
\texttt{grd\_self}: i \notin \texttt{IncidentManaged}
\texttt{grd\_par}: \texttt{AllocateRes}[\{i\}] = \texttt{DeallocateRes}[\{i\}]

then

\texttt{act}: \texttt{IncidentManaged} := \texttt{IncidentManaged} \cup \{i\}

end

\textbf{Event} \texttt{CloseIncident} (ordinary) \refines

\textbf{refines} \texttt{CloseIncident}
any
i

where
grd_self: i \notin CloseIncident
grd_seq.: (i \in SendStopMsg \land i \in IncidentManaged)
gd1: \quad ap\{i\} \setminus notRequired\_ap\{i\} \subseteq DeallocateRes\{i\}
to ensure all action plan items are satisfied
gd2: i \notin ran\(alloc\)

then
act: CloseIncident := CloseIncident \cup \{i\}

end

END

A.8 Seventh Refinement

A.8.1 Context C4

CONTEXT C4
EXTENDS C3
SETS
STATIONS
CONSTANTS
s\_location
rl0
duration0

AXIOMS
axm1: s\_location \in STATIONS \rightarrow LOCATION
axm2: rl0 \in PR \rightarrow ran(s\_location)
duration0: duration0 \in (LOCATION \times LOCATION) \rightarrow \mathbb{N}
axm3: \forall l \cdot (l \in LOCATION \Rightarrow duration0(l \mapsto l) = 0)
axm4: \forall l0, l1 \cdot (l0 \in LOCATION \setminus \{l1\} \Rightarrow duration0(l0 \mapsto l1) \neq 0 \land duration0(l1 \mapsto l0) \neq 0)

END

A.8.2 Machine M7

MACHINE M7
REFINES M6
Appendix A Event-B Model of the Fire Dispatch Workflow

SEES C4

VARIABLES

CreateIncident
IsDuplicate
CloseDuplicate
NotDuplicate
SetInitialAP
SelectType
SelectLocation
InfoComp
RequestAdditionalRes
IncidentManaged
CloseIncident
SendStopMsg
AllocateRes
ResAttend
DeallocateRes

iType added manually
ap added manually
iLocation
iLocType added manually
alloc manually
alloc,LR manually
rLocation location of a physical resource
duration duration from one location to another this is a variable because it depends on
time and not distance which can be affected by traffic
StopMsgReceived
notRequired_ap

INVARIANTS

inv_r_loc_type: rLocation ∈ PR → LOCATION

inv_duration_type: duration ∈ (LOCATION × LOCATION) → N

EVENTS

Initialisation {extended}

begin

act_CreateIncident: CreateIncident := ∅
act_SelectType: SelectType := ∅
act_SelectLocation: SelectLocation := ∅
act_InfoComp: InfoComp := ∅
act_IsDuplicate: \textit{IsDuplicate} := \emptyset
act_CloseDuplicate: \textit{CloseDuplicate} := \emptyset
act_NotDuplicate: \textit{NotDuplicate} := \emptyset
act_SetInitialAP: \textit{SetInitialAP} := \emptyset
act_RequestAdditionalRes: \textit{RequestAdditionalRes} := \emptyset
act_SendStopMsg: \textit{SendStopMsg} := \emptyset
act_AllocateRes: \textit{AllocateRes} := \emptyset
act_ResAttend: \textit{ResAttend} := \emptyset
act_DeallocateRes: \textit{DeallocateRes} := \emptyset
act_IncidentManaged: \textit{IncidentManaged} := \emptyset
act_CloseIncident: \textit{CloseIncident} := \emptyset

act1: ap := \emptyset
act2: iLocation := \emptyset
act_iLocType: iLocType := \emptyset
act3: iType := \emptyset
act_alloc: alloc := \emptyset
act_alloc_LR: alloc_{LR} := \emptyset
act_notRequired: notRequired\_ap := \emptyset
act_StopMsgReceived: StopMsgReceived := \emptyset
act_r_location: rLocation := rL0
act_duration: duration := duration0

end

\textbf{Event} AllocateRes (ordinary) \equiv

\textbf{extends} AllocateRes

\textit{any}

\begin{align*}
& i \\
& lr \\
& pr
\end{align*}

\textbf{where}

\begin{align*}
grd\_seq: & \quad i \in \text{SetInitialAP} \\
grd\_self: & \quad i \mapsto lr \notin \text{AllocateRes} \\
grd\_InputExpression1: & \quad lr \in ap\{i\} \\
grd0\_par: & \quad i \notin \text{IncidentManaged} \\
grd1: & \quad pr \notin \text{dom(alloc)} \\
& \quad \text{added manually} \\
grd2: & \quad \text{ptype(pr) = ltype(lr)} \\
& \quad \text{added manually} \\
grd11: & \quad i \mapsto lr \notin \text{notRequired\_ap}
\end{align*}

\textbf{then}

\begin{align*}
\text{act: } & \quad \text{AllocateRes} := \text{AllocateRes} \cup \{i \mapsto lr\}
\end{align*}
act1: alloc(pr) := i
added manually
act2: alloc_LR(pr) := lr
added manually

end

Event UpdateResLocation (ordinary) ≜
extends UpdateResLocation

any
i
lr
l
l0
pr

where

grd_seq: i ↦ lr ∈ AllocateRes
grd0_loop: i ↞ lr ∉ (ResAttend ∪ StopMsgReceived)
grd1: pr ∈ (alloc^{-1}[\{i\}] ∩ alloc_LR^{-1}[\{lr\}])
grd10: l0 = rLocation(pr)
grd11: l ∈ LOCATION \ {l0}
grd_dur: duration(l ↞ iLocation(i)) < duration(l0 ↞ iLocation(i))

then
act_loc: rLocation(pr) := l

end

Event ReallocateQuickerRes (ordinary) ≜
extends ReallocateQuickerRes

any
i
lr
pr
pr0

where

grd_seq: i ↦ lr ∈ AllocateRes
grd0_loop: i ↞ lr ∉ (ResAttend ∪ StopMsgReceived)
grd1: pr0 ∈ (alloc^{-1}[\{i\}] ∩ alloc_LR^{-1}[\{lr\}])
                added manually
grd2: pr ∉ dom(alloc)
                manually
grd3: ptype(pr) = ltype(lr)
                manually
grd11: i ↞ lr ∉ notRequired_ap
                manually
\textbf{Appendix A Event-B Model of the Fire Dispatch Workflow}

\begin{verbatim}
grd_quicker: duration(rLocation(pr) ↦→ iLocation(i)) < duration(rLocation(pr0) ↦→ iLocation(i))

then
act1: alloc := \{pr0\} ⊕ alloc \cup \{pr ↦→ i\}
act2: alloc_LR := \{pr0\} ⊕ alloc_LR \cup \{pr ↦→ lr\}
end

\textbf{Event} ResAttend \langle\text{ordinary}\rangle ≡
\textbf{extends} ResAttend

any
\quad i
\quad lr
\quad pr
where
\quad grd_self: i ↦→ lr \notin ResAttend
\quad grd_seq: i ↦→ lr \in AllocateRes
\quad grd_xor: i ↦→ lr \notin StopMsgReceived
\quad @grd1 i ↦→ lr \notin notRequired_ap not necessary because it could be added but haven’t received msg yet
\quad grd1: pr \in (alloc^{-1}([i]) \cap alloc_LR^{-1}([lr]))
\quad grd_loc: rLocation(pr) = iLocation(i)

then
act: ResAttend := ResAttend \cup \{i ↦→ lr\}
end

\textbf{Event} CreateIncident \langle\text{ordinary}\rangle ≡
\textbf{extends} CreateIncident

any
\quad i
where
\quad grd_self: i \notin CreateIncident
\quad grd1: t \in INCIDENT_TYPE

then
act: CreateIncident := CreateIncident \cup \{i\}
end

\textbf{Event} SelectType \langle\text{ordinary}\rangle ≡
\textbf{extends} SelectType

any
\quad t
\quad i
where
\quad grd_self: i \notin SelectType
\quad grd1: t \in INCIDENT_TYPE
\end{verbatim}
Appendix A Event-B Model of the Fire Dispatch Workflow

\[
\text{grd_seq: } \ i \in \text{CreateIncident}
\]
\[
\text{then}
\]
\[
\text{act: SelectType := SelectType } \cup \{i\}
\]
\[
\text{act1: } iType(i) := t
\]
\[
\text{end}
\]

\textbf{Event}\ SelectLocation \ (\text{ordinary}) \ \triangleq

\textbf{extends}\ SelectLocation

\text{any}

\[
i
\]
\[
l
\]
\[
lt
\]
\text{where}

\[
\text{grd_seq: } \ i \in \text{CreateIncident}
\]
\[
\text{grd_self: } \ i \notin \text{SelectLocation}
\]
\[
\text{grd1: } \ l \in \text{LOCATION}
\]
\[
\text{grd2: } \ lt \in \text{LOCATION\_TYPE}
\]
\[
\text{then}
\]
\[
\text{act: SelectLocation := SelectLocation } \cup \{i\}
\]
\[
\text{act1: } iLocation(i) := l
\]
\[
\text{act2: } iLocType(i) := lt
\]
\[
\text{end}
\]

\textbf{Event}\ InfoComp \ (\text{ordinary}) \ \triangleq

\textbf{extends}\ InfoComp

\text{any}

\[
i
\]
\text{where}

\[
\text{grd_self: } \ i \notin \text{InfoComp}
\]
\[
\text{grd_seq: } \ (i \in \text{SelectType } \land \ i \in \text{SelectLocation})
\]
\[
\text{then}
\]
\[
\text{act: InfoComp := InfoComp } \cup \{i\}
\]
\[
\text{end}
\]

\textbf{Event}\ IsDuplicate \ (\text{ordinary}) \ \triangleq

\textbf{extends}\ IsDuplicate

\text{any}

\[
i0
\]
\[
i
\]
\text{where}

\[
\text{grd_self: } \ i \notin \text{IsDuplicate}
\]
\[
\text{grd1\_xor: } \ i \notin \text{NotDuplicate}
\]
\[
\text{grd\_i0: } \ i0 \in \text{NotDuplicate } \setminus \text{CloseIncident}
\]
\text{manually}
Appendix A Event-B Model of the Fire Dispatch Workflow

\[\text{grd_seq.Duplicate}: \; i \in \text{InfoComp} \]
\[\text{grd_i01}: \; \text{Location}(i) = \text{Location}(i0) \]

\[\begin{align*}
\text{then} & \quad \text{act}: \; \text{IsDuplicate} := \text{IsDuplicate} \cup \{i\} \\
\text{end}
\end{align*} \]

\textbf{Event} \; \text{CloseDuplicate} \; \langle \text{ordinary} \rangle \; \equiv \textbf{extends} \; \text{CloseDuplicate}

\[\begin{align*}
\text{any} & \quad i \\
\text{where} & \\
\text{grd_self}: & \; i \notin \text{CloseDuplicate} \\
\text{grd_seq.Duplicate}: & \; i \in \text{IsDuplicate} \\
\text{then} & \quad \text{act}: \; \text{CloseDuplicate} := \text{CloseDuplicate} \cup \{i\} \\
\text{end}
\end{align*} \]

\textbf{Event} \; \text{NotDuplicate} \; \langle \text{ordinary} \rangle \; \equiv \textbf{extends} \; \text{NotDuplicate}

\[\begin{align*}
\text{any} & \quad i \\
\text{where} & \\
\text{grd1_xor}: & \; i \notin \text{IsDuplicate} \\
\text{grd_self}: & \; i \notin \text{NotDuplicate} \\
\text{grd_seq.NotDuplicate}: & \; i \in \text{InfoComp} \\
\text{then} & \quad \text{act}: \; \text{NotDuplicate} := \text{NotDuplicate} \cup \{i\} \\
\text{end}
\end{align*} \]

\textbf{Event} \; \text{SetInitialAP} \; \langle \text{ordinary} \rangle \; \equiv \textbf{extends} \; \text{SetInitialAP}

\[\begin{align*}
\text{any} & \quad i \\
\text{where} & \\
\text{grd_self}: & \; i \notin \text{SetInitialAP} \\
\text{grd_seq.NotDuplicate}: & \; i \in \text{NotDuplicate} \\
\text{grd1}: & \; t = \text{iType}(i) \\
\text{grd2}: & \; l = \text{iLocType}(i) \\
\text{grd3}: & \; lrs = \text{PDA}(t \rightarrow l) \\
\text{then}
\end{align*} \]
Appendix A Event-B Model of the Fire Dispatch Workflow

\textbf{act:} SetInitialAP := SetInitialAP \cup \{i\}
\textbf{act1:} ap := ap \cup (\{i\} \times lrs)
\hspace{1cm} \text{manually added}
\textbf{Event RequestAdditionalRes (ordinary) \equiv}
\textbf{extends} RequestAdditionalRes
\hspace{1cm} \textbf{any}
\hspace{1.5cm} i
\hspace{1.5cm} lr
\hspace{1.5cm} \textbf{where}
\hspace{2cm} \textbf{grd\_self:} \ i \rightarrow lr \notin \text{RequestAdditionalRes}
\hspace{2cm} \textbf{grd\_InputExpression1:} \ lrs \in (LR \setminus ap[\{i\}])
\hspace{2cm} \textbf{grd\_par:} \ i \notin \text{SendStopMsg}
\hspace{2cm} \textbf{grd\_seq:} \ i \in \text{SetInitialAP}
\hspace{1cm} \textbf{then}
\hspace{2cm} \textbf{act:} RequestAdditionalRes := RequestAdditionalRes \cup \{i \mapsto lr\}
\hspace{2cm} \textbf{act1:} ap := ap \cup \{i \mapsto lr\}
\hspace{1cm} \text{manually added}
\textbf{end}
\textbf{Event SendStopMsg (ordinary) \equiv}
\textbf{extends} SendStopMsg
\hspace{1cm} \textbf{any}
\hspace{1.5cm} i
\hspace{1.5cm} lr
\hspace{1.5cm} \textbf{where}
\hspace{2cm} \textbf{grd\_seq\_planF:} \ i \in \text{SetInitialAP}
\hspace{2cm} \textbf{grd\_self:} \ i \notin \text{SendStopMsg}
\hspace{2cm} \textbf{grd\_DuplicateInterrupt:} \ i \notin \text{IsDuplicate}
\hspace{2cm} \textbf{grd\_lrs:} \ lrs = ap[\{i\}] \setminus \text{ResAttend}[\{i\}]
\hspace{1cm} \textbf{then}
\hspace{2cm} \textbf{act:} SendStopMsg := SendStopMsg \cup \{i\}
\hspace{2cm} \textbf{act1:} notRequired\_ap := notRequired\_ap \cup (\{i\} \times lrs)
\textbf{end}
\textbf{Event StopMsgReceived (ordinary) \equiv}
\textbf{extends} StopMsgReceived
\hspace{1cm} \textbf{any}
\hspace{1.5cm} lr
\hspace{1.5cm} i
\hspace{1.5cm} \textbf{where}
\hspace{2cm} \textbf{grd\_seq:} \ i \mapsto lr \in \text{AllocateRes}
\begin{align*}
\text{grd}\_\text{self}: & \quad i \mapsto lr \notin \text{StopMsgReceived} \\
\text{grd}\_\text{xor}: & \quad i \mapsto lr \notin \text{ResAttend} \\
\text{grd1}: & \quad i \mapsto lr \in \text{notRequired}\_\text{ap} \\
    & \quad \text{in this case it is added because it cannot be received unless not required}
\end{align*}

\textbf{then}

\begin{align*}
\text{act}: & \quad \text{StopMsgReceived} := \text{StopMsgReceived} \cup \{i \mapsto lr\}
\end{align*}

\textbf{end}

\textbf{Event} \ DeallocateRes \ \langle\text{ordinary}\rangle \ 
hat

\textbf{extends} \ DeallocateRes

\textbf{any}

\begin{align*}
i \\
lr \\
pr
\end{align*}

\textbf{where}

\begin{align*}
\text{grd}\_\text{seq}: & \quad i \mapsto lr \in (\text{ResAttend} \cup \text{StopMsgReceived}) \\
\text{grd1}: & \quad pr \in (\text{alloc}\_1[i] \cap \text{alloc\_LR}\_1[\{lr\}]) \\
\text{grd}\_\text{self}: & \quad i \mapsto lr \notin \text{DeallocateRes}
\end{align*}

\textbf{then}

\begin{align*}
\text{act1}: & \quad \text{alloc} := \{pr\} \leftarrow \text{alloc} \\
\text{act2}: & \quad \text{alloc\_LR} := \{pr\} \leftarrow \text{alloc\_LR} \\
\text{act}: & \quad \text{DeallocateRes} := \text{DeallocateRes} \cup \{i \mapsto lr\}
\end{align*}

\textbf{end}

\textbf{Event} \ IncidentManaged \ \langle\text{ordinary}\rangle \ 
hat

\textbf{extends} \ IncidentManaged

\textbf{any}

\begin{align*}
i
\end{align*}

\textbf{where}

\begin{align*}
\text{grd}\_\text{seq}: & \quad i \in \text{SetInitialAP} \\
\text{grd}\_\text{self}: & \quad i \notin \text{IncidentManaged} \\
\text{grd}\_\text{par}: & \quad \text{AllocateRes}[i] = \text{DeallocateRes}[i]
\end{align*}

\textbf{then}

\begin{align*}
\text{act}: & \quad \text{IncidentManaged} := \text{IncidentManaged} \cup \{i\}
\end{align*}

\textbf{end}

\textbf{Event} \ CloseIncident \ \langle\text{ordinary}\rangle \ 
hat

\textbf{extends} \ CloseIncident

\textbf{any}

\begin{align*}
i
\end{align*}

\textbf{where}

\begin{align*}
\text{grd}\_\text{self}: & \quad i \notin \text{CloseIncident} \\
\text{grd}\_\text{seq}: & \quad (i \in \text{SendStopMsg} \land i \in \text{IncidentManaged})
\end{align*}
Appendix A Event-B Model of the Fire Dispatch Workflow

\[ \text{gd1: } \{i\} \setminus \text{notRequired}_\{i\} \subseteq \text{DeallocateRes}_\{i\} \]

to ensure all action plan items are satisfied

\[ \text{grd2: } i \notin \text{ran}(\text{alloc}) \]

\[ \text{then} \]

\[ \text{act: } \text{CloseIncident} := \text{CloseIncident} \cup \{i\} \]

\[ \text{end} \]

END
Appendix B

Fire Dispatch Model with Exception Handling

In this chapter we present the complete version of the fire dispatch workflow after introducing the exception handling combinators.

B.1 Static Part of the Model: Contexts

B.1.1 Context: C0

\textsc{context} C0

\textsc{sets}

\text{incident}

\text{end}

B.1.2 Context: C1

\textsc{context} C1

\text{extends} C0

\text{sets}

\text{lr}

\text{end}

B.1.3 Context: C2

\textsc{context} C2

\text{extends} C1
SETS
    INCIDENT_TYPE
    LOCATION
    LOCATION_TYPE

CONSTANTS
    PDA set the action plan according to the type and location

AXIOMS
    axm1: \( PDA : INCIDENT_TYPE \times LOCATION_TYPE \rightarrow P(LR) \)
    axm2: \( \forall \text{it}, \text{loc}: \text{it} \in INCIDENT_TYPE \land \text{loc} \in LOCATION_TYPE \Rightarrow PDA(it \mapsto loc) \neq \emptyset \)

END

B.1.4 Context: C3

CONTEXT C3
EXTENDS C2
SETS
    R_TYPE type of the resource

CONSTANTS
    ltype
    default_priority

AXIOMS
    axm1: default_priority \in INCIDENT_TYPE \rightarrow 1..3
    axm2: ltype \in LR \rightarrow R_TYPE

logical type of a resource

END

B.1.5 Context: C4

CONTEXT C4
EXTENDS C3
SETS
    PR set of physical resources

CONSTANTS
    ptype

AXIOMS
    axm2: ptype \in PR \rightarrow R_TYPE

physical type of a resource
\[ axm3: \forall lr \cdot lr \in LR \Rightarrow (\exists pr \cdot ptype(pr) = ltype(lr)) \]

**END**

### B.1.6 Context: C5

**CONTEXT** C5  
**EXTENDS** C4  
**SETS**  
STATIONS  
**CONSTANTS**  
s_location  
rl0  
duration0  

**AXIOMS**  
\[ axm1: s_{location} \in STATIONS \rightarrow LOCATION \]  
\[ axm2: rl0 \in PR \rightarrow ran(s_{location}) \]  
\[ duration0: duration0 \in (LOCATION \times LOCATION) \rightarrow N \]  
\[ axm3: \forall l \cdot (l \in LOCATION \Rightarrow duration0(l \mapsto l) = 0) \]  
\[ axm4: \forall l0, l1 \cdot (l0 \in LOCATION \setminus \{l1\} \Rightarrow duration0(l0 \mapsto l1) \neq 0 \land duration0(l1 \mapsto l0) \neq 0) \]

**END**

### B.1.7 Context: M9_implicitContext

This context is generated by the iUML-B statemachine.

**CONTEXT** M9_implicitContext  
**SETS**  
pr_status_STATES  
pr_available_sm_STATES  
pr_allocated_sm_STATES  
pr_free_sm_STATES  
**CONSTANTS**  
Available  
pr_available_sm_NULL  
pr_allocated  
pr_allocated_sm_NULL
Mobilised
InAttendance
pr_free
pr_free_sm_NULL
Returning
StationAvailable
Unavailable

**AXIOMS**

\[
\begin{align*}
typeof\ text{pr\ available\ sm\ NULL} & : \ pr\ available\ sm\ NULL \in \ pr\ available\ sm\ STATES \\
typeof\ text{pr\ allocated\ sm\ NULL} & : \ pr\ allocated\ sm\ NULL \in \ pr\ allocated\ sm\ STATES \\
typeof\ text{pr\ free\ sm\ NULL} & : \ pr\ free\ sm\ NULL \in \ pr\ free\ sm\ STATES \\
typeof\ text{Available} & : \ Available \in \ pr\ status\ STATES \\
typeof\ text{pr\ allocated} & : \ pr\ allocated \in \ pr\ available\ sm\ STATES \\
typeof\ text{Mobilised} & : \ Mobilised \in \ pr\ allocated\ sm\ STATES \\
typeof\ text{InAttendance} & : \ InAttendance \in \ pr\ allocated\ sm\ STATES \\
typeof\ text{pr\ free} & : \ pr\ free \in \ pr\ available\ sm\ STATES \\
typeof\ text{Returning} & : \ Returning \in \ pr\ free\ sm\ STATES \\
typeof\ text{StationAvailable} & : \ StationAvailable \in \ pr\ free\ sm\ STATES \\
typeof\ text{Unavailable} & : \ Unavailable \in \ pr\ status\ STATES \\
distinct\ states\ in\ pr\ status\ STATES: & \ partition(pr\ status\ STATES, \{Available\}, \{Unavailable\}) \\
distinct\ states\ in\ pr\ available\ sm\ STATES: & \ partition(pr\ available\ sm\ STATES, \{pr\ allocated\}, \{pr\ free\}, \{pr\ available\ sm\ NULL\}) \\
distinct\ states\ in\ pr\ allocated\ sm\ STATES: & \ partition(pr\ allocated\ sm\ STATES, \{Mobilised\}, \{InAttendance\}, \{pr\ allocated\ sm\ NULL\}) \\
distinct\ states\ in\ pr\ free\ sm\ STATES: & \ partition(pr\ free\ sm\ STATES, \{Returning\}, \{StationAvailable\}, \{pr\ free\ sm\ NULL\}) \\
\end{align*}
\]

**END**

### B.2 Model Refinements

#### B.2.1 Abstract Level: M0

**MACHINE** M0

**SEES** C0

**VARIABLES**

CreateIncident
NotDuplicate
IsDuplicate

INvariants

\[ inv_{CreateIncident\_type}: \text{CreateIncident} \subseteq \text{INCIDENT} \]
\[ inv_{NotDuplicate\_seq}: \text{NotDuplicate} \subseteq \text{CreateIncident} \]
\[ inv_{IsDuplicate\_DuplicateInterrupt\_seq}: \text{IsDuplicate} \subseteq \text{CreateIncident} \]
\[ inv_{Interrupt}: \text{partition}((\text{NotDuplicate} \cup \text{IsDuplicate}), \text{NotDuplicate}, \text{IsDuplicate}) \]

EVENTS

Initialisation

\begin{verbatim}
begin
  act_{CreateIncident}: CreateIncident := \emptyset
  act_{NotDuplicate}: NotDuplicate := \emptyset
  act_{IsDuplicate}: IsDuplicate := \emptyset
end
\end{verbatim}

Event CreateIncident \(\text{(ordinary)}\) \(\Downarrow\)

\begin{verbatim}
  any
  i
  where
    grd_self: i \notin \text{CreateIncident}
  then
    act: CreateIncident := CreateIncident \cup \{i\}
\end{verbatim}

Event NotDuplicate \(\text{(ordinary)}\) \(\Downarrow\)

\begin{verbatim}
  any
  i
  where
    grd_DuplicateInterrupt: i \notin \text{IsDuplicate}
    grd_seq: i \in \text{CreateIncident}
    grd_self: i \notin \text{NotDuplicate}
  then
    act: NotDuplicate := NotDuplicate \cup \{i\}
\end{verbatim}

Event IsDuplicate \(\text{(ordinary)}\) \(\Downarrow\)

\begin{verbatim}
  any
  i
  where
    grd_seq_DuplicateInterrupt: i \in \text{CreateIncident}
    grd_self: i \notin \text{IsDuplicate}
    grd Interrupt: i \notin \text{NotDuplicate}
\end{verbatim}
then
 act: $IsDuplicate := IsDuplicate \cup \{i\}$
end

END

\textbf{B.2.2 First Refinement: M1}

\textbf{MACHINE} M1

\textbf{REFINES} M0

\textbf{SEES} C0

\textbf{VARIABLES}

CreateIncident
GatherInfo
SetInitialAP
ManageIncident
UpdateAP
CloseIncident
IsDuplicate

\textbf{INvariants}

\begin{itemize}
  \item \texttt{inv\_CreateIncident\_type}: $CreateIncident \subseteq INCIDENT$
  \item \texttt{inv\_GatherInfo\_seq}: $GatherInfo \subseteq CreateIncident$
  \item \texttt{inv\_SetInitialAP\_seq}: $SetInitialAP \subseteq GatherInfo$
  \item \texttt{inv\_ManageIncident\_seq}: $ManageIncident \subseteq SetInitialAP$
  \item \texttt{inv\_UpdateAP\_seq}: $UpdateAP \subseteq SetInitialAP$
  \item \texttt{inv\_CloseIncident\_FD\_seq}: $CloseIncident \subseteq (UpdateAP \cap ManageIncident)$
  \item \texttt{inv\_IsDuplicate\_DuplicateInterrupt\_seq}: $IsDuplicate \subseteq CreateIncident$
  \item \texttt{inv\_CloseIncident\_glu}: $CloseIncident = \text{NotDuplicate}$
  \item \texttt{inv\_IsDuplicate\_Interrupt}: $\text{partition}((CloseIncident \cup IsDuplicate), CloseIncident, IsDuplicate)$
\end{itemize}

\textbf{EVENTS}

\textbf{Initialisation}

begin
  \begin{itemize}
    \item \texttt{act\_CreateIncident}: $CreateIncident := \emptyset$
    \item \texttt{act\_GatherInfo}: $GatherInfo := \emptyset$
    \item \texttt{act\_SetInitialAP}: $SetInitialAP := \emptyset$
    \item \texttt{act\_ManageIncident}: $ManageIncident := \emptyset$
    \item \texttt{act\_CloseIncident}: $CloseIncident := \emptyset$
    \item \texttt{act\_IsDuplicate}: $IsDuplicate := \emptyset$
  \end{itemize}
\texttt{act\_UpdateAP}: UpdateAP := \emptyset
\end{verbatim}

\textbf{Event} CreateIncident \langle ordinary \rangle \trianglelefteq
\textbf{refines} CreateIncident
\textbf{any}
i
\textbf{where}
grd\_self: \ i \notin CreateIncident
\textbf{then}
\textbf{act}: \ CreateIncident := CreateIncident \cup \{i\}
\end{verbatim}

\textbf{Event} GatherInfo \langle ordinary \rangle \trianglelefteq
\textbf{any}
i
\textbf{where}
grd\_DuplicateInterrupt: \ i \notin IsDuplicate
grd\_seq: \ i \in CreateIncident
grd\_self: \ i \notin GatherInfo
\textbf{then}
\textbf{act}: \ GatherInfo := GatherInfo \cup \{i\}
\end{verbatim}

\textbf{Event} SetInitialAP \langle ordinary \rangle \trianglelefteq
\textbf{any}
i
\textbf{where}
grd\_seq: \ i \in GatherInfo
grd\_self: \ i \notin SetInitialAP
grd\_DuplicateInterrupt: \ i \notin IsDuplicate
\textbf{then}
\textbf{act}: \ SetInitialAP := SetInitialAP \cup \{i\}
\end{verbatim}

\textbf{Event} UpdateAP \langle ordinary \rangle \trianglelefteq
\textbf{any}
i
\textbf{where}
grd\_seq: \ i \in SetInitialAP
grd\_self: \ i \notin UpdateAP
grd\_DuplicateInterrupt: \ i \notin IsDuplicate
\textbf{then}
\textbf{act}: \ UpdateAP := UpdateAP \cup \{i\}
\end{verbatim}

**Event** ManageIncident (ordinary) \(\cong\)

```
any
  i
where
  grd_seq:  \(i \in \text{Set}\text{Initial}\text{AP}\)
  grd_self:  \(i \notin \text{ManageIncident}\)
  grd_DuplicateInterrupt:  \(i \notin \text{IsDuplicate}\)
then
  act:  ManageIncident := ManageIncident \cup \{i\}
end
```

**Event** CloseIncident (ordinary) \(\cong\)

refines NotDuplicate

```
any
  i
where
  grd_seq_FD:  \((i \in \text{Update}\text{AP} \land i \in \text{ManageIncident})\)
  grd_self:  \(i \notin \text{CloseIncident}\)
  grd_DuplicateInterrupt:  \(i \notin \text{IsDuplicate}\)
then
  act:  CloseIncident := CloseIncident \cup \{i\}
end
```

**Event** IsDuplicate (ordinary) \(\cong\)

refines IsDuplicate

```
any
  i
where
  grd_seq:  \(i \in \text{CreateIncident}\)
  grd_self:  \(i \notin \text{IsDuplicate}\)
  grd_DuplicateInterrupt:  \(i \notin \text{CloseIncident}\)
then
  act:  IsDuplicate := IsDuplicate \cup \{i\}
end
```

END

### B.2.3 Second Refinement: M2

**MACHINE** M2

**REFINES** M1

**SEES** C1

**VARIABLES**
CreateIncident
GatherInfo
SetInitialAP
ManageIncident
RequestAdditionalRes
SendStopMsg
CloseIncident
IsDuplicate
ap
notRequired_ap

INVARINTS

\text{inv}_\text{CreateIncident}_\text{type}: \text{CreateIncident} \subseteq \text{INCIDENT}
\text{inv}_\text{SetActionPlan}_\text{type}: \text{SetInitialAP} \subseteq \text{INCIDENT}
\text{inv}_\text{RequestAdditionalResources}_\text{type}: \text{RequestAdditionalRes} \subseteq \text{INCIDENT} \times \text{LR}
\text{inv}_\text{GatherInfo}_\text{seq}: \text{GatherInfo} \subseteq \text{CreateIncident}
\text{inv}_\text{SetActionPlan}_\text{glu}: \text{SetInitialAP} = \text{SetInitialAP}
\text{inv}_\text{SetActionPlan}_\text{inSeq}: \text{SetInitialAP} \subseteq \text{GatherInfo}
\text{inv}_\text{RequestAdditionalResources}_\text{planF}_\text{seq}: \text{dom(} \text{RequestAdditionalRes}) \subseteq \text{SetInitialAP}
\text{inv}_\text{RequestAdditionalRes}_\text{Rep}: \forall i \cdot i \in \text{INCIDENT} \Rightarrow \text{RequestAdditionalRes}[i] \subseteq \text{LR}
\text{Replicator invariant}
\text{inv}_\text{SendStopMsg}_\text{seq}: \text{SendStopMsg} \subseteq \text{SetInitialAP}
\text{inv}_\text{SendStopMsg}_\text{glu}: \text{SendStopMsg} = \text{UpdateAP}
\text{inv}_\text{ManageIncident}_\text{seq}: \text{ManageIncident} \subseteq \text{SetInitialAP}
\text{inv}_\text{CloseIncident}_\text{FD}_\text{seq}: \text{CloseIncident} \subseteq \text{ManageIncident} \cap \text{SendStopMsg}
\text{changed weak seq to and refinement}
\text{inv}_\text{IsDuplicate}_\text{DuplicateInterrupt}_\text{seq}: \text{IsDuplicate} \subseteq \text{CreateIncident}
\text{inv}_\text{ap}_\text{type}: \text{ap} \in \text{INCIDENT} \leftrightarrow \text{LR}
\text{inv}_\text{ap}_\text{seq}: \text{dom(} \text{ap}) = \text{SetInitialAP}
\text{inv}_\text{notRequired}_\text{ap}_\text{type}: \text{notRequired}_\text{ap} \subseteq \text{ap}
\text{manually added data variable to describe the actions plan items that are not required after stop message is sent}
\text{inv}_\text{Interrupt}: \text{partition(} \text{CloseIncident} \cup \text{IsDuplicate}, \text{CloseIncident}, \text{IsDuplicate})

EVENTS

Initialisation
begin
  act_CreateIncident: CreateIncident := ∅
  act_GatherInfo: GatherInfo := ∅
  act_SetInitialAP: SetInitialAP := ∅
  act_ManageIncident: ManageIncident := ∅
  act_CloseIncident: CloseIncident := ∅
  act_IsDuplicate: IsDuplicate := ∅
  act_SendStopMsg: SendStopMsg := ∅
  act_ap: ap := ∅
  act_notRequired_ap: notRequired_ap := ∅
  act_RequestAdditionalRes: RequestAdditionalRes := ∅
end

Event CreateIncident (ordinary) \(\triangleq\)
refines CreateIncident
  any
  i
  where
  grd_self: i \notin CreateIncident
  then
  act: CreateIncident := CreateIncident \cup \{i\}
end

Event GatherInfo (ordinary) \(\triangleq\)
refines GatherInfo
  any
  i
  where
  grd_self: i \notin GatherInfo
  grd_DuplicateInterrupt: i \notin IsDuplicate
  grd_seq.: i \in CreateIncident
  then
  act: GatherInfo := GatherInfo \cup \{i\}
end

Event SetInitialAP (ordinary) \(\triangleq\)
refines SetInitialAP
  any
  i
  lrs
  where
  grd_DuplicateInterrupt: i \notin IsDuplicate
  grd_seq.: i \in GatherInfo
Appendix B  Fire Dispatch Model with Exception Handling

\[
\begin{align*}
grd\_self: & \quad i \notin SetInitialAP \\
grd1: & \quad lrs \subseteq LR \\
grd2: & \quad lrs \neq \emptyset \\
\text{then} & \\
\text{act}: & \quad SetInitialAP := SetInitialAP \cup \{i\} \\
\text{act1}: & \quad ap := ap \cup \{i \times lrs\} \\
\text{end} \\
\text{Event RequestAdditionalRes (ordinary)} & \triangleq \\
\text{any} & \\
i & \\
lr & \\
\text{where} & \\
grd\_seq.planF: & \quad i \in SetInitialAP \\
grd\_self: & \quad i \mapsto lr \notin RequestAdditionalRes \\
grd\_InputExpression1: & \quad lr \in LR \setminus ap[i] \\
grd0\_par: & \quad i \notin SendStopMsg \\
grd\_DuplicateInterrupt: & \quad i \notin IsDuplicate \\
grd1: & \quad lr \notin ap[i] \\
\text{added manually to replace lr \in LR \setminus ap[i]} & \\
\text{then} & \\
\text{act}: & \quad RequestAdditionalRes := RequestAdditionalRes \cup \{i \mapsto lr\} \\
\text{act1}: & \quad ap := ap \cup \{i \mapsto lr\} \\
\text{end} \\
\text{Event SendStopMsg (ordinary)} & \triangleq \\
\text{refines} & \quad UpdateAP \\
\text{any} & \\
i & \\
lrs & \\
\text{where} & \\
grd\_DuplicateInterrupt: & \quad i \notin IsDuplicate \\
grd\_seq.: & \quad i \in SetInitialAP \\
grd\_self: & \quad i \notin SendStopMsg \\
grd\_lrs: & \quad lrs \subseteq ap[i] \\
\text{then} & \\
\text{act}: & \quad SendStopMsg := SendStopMsg \cup \{i\} \\
\text{act1}: & \quad notRequired\_ap := notRequired\_ap \cup \{i \times lrs\} \\
\text{end} \\
\text{Event ManageIncident (ordinary)} & \triangleq \\
\text{refines} & \quad ManageIncident \\
\text{any} & \\
i & \\
\]
where
\[ \text{grd\_DuplicateInterrupt}: \ i \notin \text{IsDuplicate} \]
\[ \text{grd\_seq}: \ i \in \text{SetInitialAP} \]
\[ \text{grd\_self}: \ i \notin \text{ManageIncident} \]
then
\[ \text{act: } \text{ManageIncident} := \text{ManageIncident} \cup \{i\} \]
end

Event CloseIncident (ordinary) \(\cong\)
refines CloseIncident
any
\[ i \]
where
\[ \text{grd\_seq\_FD}: \ (i \in \text{SendStopMsg} \land i \in \text{ManageIncident}) \]
\[ \text{grd\_self}: \ i \notin \text{CloseIncident} \]
\[ \text{grd\_DuplicateInterrupt}: \ i \notin \text{IsDuplicate} \]
then
\[ \text{act: } \text{CloseIncident} := \text{CloseIncident} \cup \{i\} \]
end

Event IsDuplicate (ordinary) \(\cong\)
refines IsDuplicate
any
\[ i \]
where
\[ \text{grd\_seq\_DuplicateInterrupt}: \ i \in \text{CreateIncident} \]
\[ \text{grd\_self}: \ i \notin \text{IsDuplicate} \]
\[ \text{grd1}: \ i \notin \text{CloseIncident} \]
then
\[ \text{act: } \text{IsDuplicate} := \text{IsDuplicate} \cup \{i\} \]
end

END

B.2.4 Third Refinement: M3

MACHINE M3
REFINES M2
SEES C2
VARIABLES
CreateIncident
SelectLocation
SelectType
Appendix B Fire Dispatch Model with Exception Handling

CompInfo
SetInitialAP
ManageIncident
RequestAdditionalRes
SendStopMsg
CloseIncident
IsDuplicate
ap
notRequired_ap
iLocation
iType
iLocType

INVARIANTS

inv_CreateIncident_type: CreateIncident \(\subseteq\) INCIDENT
inv_SetActionPlan_type: SetInitialAP \(\subseteq\) INCIDENT
inv_RequestAdditionalResources_type: RequestAdditionalRes \(\subseteq\) INCIDENT \(\times\) LR

inv_SelectType_seq: SelectType \(\subseteq\) CreateIncident
inv_CompInfo_seq: CompInfo \(\subseteq\) (SelectLocation \(\cap\) SelectType)
inv_CompInfo_glu: CompInfo = GatherInfo
inv_SetActionPlan_glu: SetInitialAP = SetInitialAP
inv_SetActionPlan_inSeq: SetInitialAP \(\subseteq\) CompInfo
inv_RequestAdditionalResources_planF_seq: \(\text{dom}(\text{RequestAdditionalRes}) \subseteq\) SetInitialAP

inv_RequestAdditionalRes_Rep: \(\forall i \cdot\ i \in\ \text{INCIDENT} \Rightarrow \text{RequestAdditionalRes}[[i]] \subseteq LR\)
Replicator invariant

inv_SendStopMsg_seq: SendStopMsg \(\subseteq\) SetInitialAP
inv_ManageStopMsg_seq: ManageIncident \(\subseteq\) SetInitialAP
inv_CloseIncident_FD_seq: CloseIncident \(\subseteq\) ManageIncident \(\cap\) SendStopMsg
inv_IsDuplicate_DuplicateInterrupt_seq: IsDuplicate \(\subseteq\) CreateIncident
inv_Interrupt: partition((CloseIncident\(\cup\)IsDuplicate), CloseIncident, IsDuplicate)

inv_iLocation: iLocation \(\in\) SelectLocation \(\rightarrow\) LOCATION
data variable

inv_loc_type: iLocType \(\in\) SelectLocation \(\rightarrow\) LOCATION_TYPE
data variable

inv_iType: iType \(\in\) SelectType \(\rightarrow\) INCIDENT_TYPE
data variable
EVENTS
Initialisation
begin
  act_CreateIncident: CreateIncident := ∅
  act_SetInitialAP: SetInitialAP := ∅
  act_ManageIncident: ManageIncident := ∅
  act_CloseIncident: CloseIncident := ∅
  act_IsDuplicate: IsDuplicate := ∅
  act_SendStopMsg: SendStopMsg := ∅
  act_ap: ap := ∅
  act_notRequired_ap: notRequired_ap := ∅
  act_RequestAdditionalRes: RequestAdditionalRes := ∅
  act2: iLocation := ∅
  act3: iType := ∅
  act_loc_type: iLocType := ∅
  act_SelectLocation: SelectLocation := ∅
  act_SelectType: SelectType := ∅
  act_CompInfo: CompInfo := ∅
end
Event CreateIncident (ordinary) ≜
refines CreateIncident
  any
  i
  where
    grd_self: i /∈ CreateIncident
  then
    act: CreateIncident := CreateIncident ∪ {i}
end
Event SelectLocation (ordinary) ≜
  any
  i
  l
  lt
  where
    grd_seq: i ∈ CreateIncident
    grd_self: i /∈ SelectLocation
    grd_DuplicateInterrupt: i /∈ IsDuplicate
    grd1: l ∈ LOCATION
    grd2: lt ∈ LOCATION_TYPE
  then
    act: SelectLocation := SelectLocation ∪ {i}
\[\begin{align*}
act_1: \quad & iLocation(i) := l \\
act_2: \quad & iLocType(i) := lt \\
end
\]

**Event** SelectType (ordinary) \[\cong\]

any

\[\begin{align*}
i \\
t
\end{align*}\]

where

\[\begin{align*}
g_{\text{seq.}}: \quad & i \in \text{CreateIncident} \\
g_{\text{self}}: \quad & i \notin \text{SelectType} \\
g_1: \quad & t \in \text{INCIDENT\_TYPE} \\
g_{\text{DuplicateInterrupt}}: \quad & i \notin \text{IsDuplicate}
\end{align*}\]

then

\[\begin{align*}
act: \quad & \text{SelectType} := \text{SelectType} \cup \{i\} \\
act_1: \quad & iType(i) := t
\end{align*}\]

end

**Event** CompInfo (ordinary) \[\cong\]

refines GatherInfo

any

\[\begin{align*}
i \\
\end{align*}\]

where

\[\begin{align*}
g_{\text{DuplicateInterrupt}}: \quad & i \notin \text{IsDuplicate} \\
g_{\text{seq.}}: \quad & (i \in \text{SelectLocation} \land i \in \text{SelectType}) \\
g_{\text{self}}: \quad & i \notin \text{CompInfo}
\end{align*}\]

then

\[\begin{align*}
act: \quad & \text{CompInfo} := \text{CompInfo} \cup \{i\}
\end{align*}\]

end

**Event** SetInitialAP (ordinary) \[\cong\]

refines SetInitialAP

any

\[\begin{align*}
i \\
t \\
l \\
lrs
\end{align*}\]

where

\[\begin{align*}
g_{\text{seq}}: \quad & i \in \text{CompInfo} \\
g_{\text{self}}: \quad & i \notin \text{SetInitialAP} \\
g_1: \quad & t = iType(i) \\
g_2: \quad & l = iLocType(i) \\
g_3: \quad & lrs = \text{PDA}(t \rightarrow l) \\
g_{\text{DuplicateInterrupt}}: \quad & i \notin \text{IsDuplicate}
\end{align*}\]
then
  \texttt{act}: \texttt{SetInitialAP} := \texttt{SetInitialAP} \cup \{i\}
  \texttt{act1}: \texttt{ap} := \texttt{ap} \cup (\{i\} \times \texttt{lrs})
end

\textbf{Event} RequestAdditionalRes \langle \textit{ordinary} \rangle \cong
\textbf{refines} RequestAdditionalRes

any
  lr
  i

\textbf{where}
  \texttt{grd\_seq\_planF}: i \in \texttt{SetInitialAP}
  \texttt{grd\_self}: i \mapsto lr \notin \texttt{RequestAdditionalRes}
  \texttt{grd\_InputExpression1}: lr \in \texttt{LR}
  \texttt{grd0\_par}: i \notin \texttt{SendStopMsg}
  \texttt{grd\_DuplicateInterrupt}: i \notin \texttt{IsDuplicate}
  \texttt{grd1}: lr \notin \texttt{ap}\{i\}
then
  \texttt{act1}: \texttt{ap} := \texttt{ap} \cup \{i \mapsto lr\}
  \texttt{act}: RequestAdditionalRes := RequestAdditionalRes \cup \{i \mapsto lr\}
end

\textbf{Event} SendStopMsg \langle \textit{ordinary} \rangle \cong
\textbf{refines} SendStopMsg

any
  i
  lrs

\textbf{where}
  \texttt{grd\_DuplicateInterrupt}: i \notin \texttt{IsDuplicate}
  \texttt{grd\_seq}: i \in \texttt{SetInitialAP}
  \texttt{grd\_self}: i \notin \texttt{SendStopMsg}
  \texttt{grd\_lrs}: lrs \subseteq \texttt{ap}\{i\}
then
  \texttt{act}: \texttt{SendStopMsg} := \texttt{SendStopMsg} \cup \{i\}
  \texttt{act1}: \texttt{notRequired\_ap} := \texttt{notRequired\_ap} \cup (\{i\} \times \texttt{lrs})
end

\textbf{Event} ManageIncident \langle \textit{ordinary} \rangle \cong
\textbf{refines} ManageIncident

any
  i

\textbf{where}
  \texttt{grd\_DuplicateInterrupt}: i \notin \texttt{IsDuplicate}
\begin{verbatim}
grd_seq: \ i \in \text{SetInitialAP}  
grd_self: \ i \notin \text{ManageIncident}  
\text{then}  
\quad \text{act: } \text{ManageIncident} := \text{ManageIncident} \cup \{i\}  
\text{end}  
\textbf{Event} \text{ CloseIncident} (\text{ordinary}) \triangleq  
\textbf{refines} \text{ CloseIncident}  
\quad \text{any}  
\quad i  
\quad \text{where}  
\quad \text{grd_seq.FD: } (i \in \text{SendStopMsg} \land i \in \text{ManageIncident})  
\quad \text{grd_self: } i \notin \text{CloseIncident}  
\quad \text{grd_duplicateInterrupt: } i \notin \text{IsDuplicate}  
\quad \text{then}  
\quad \quad \text{act: } \text{CloseIncident} := \text{CloseIncident} \cup \{i\}  
\text{end}  
\textbf{Event} \text{ IsDuplicate} (\text{ordinary}) \triangleq  
\textbf{refines} \text{ IsDuplicate}  
\quad \text{any}  
\quad i  
\quad \text{where}  
\quad \text{grd_seq_duplicateInterrupt: } i \in \text{CreateIncident}  
\quad \text{grd_self: } i \notin \text{IsDuplicate}  
\quad \text{grd1: } i \notin \text{CloseIncident}  
\quad \text{then}  
\quad \quad \text{act: } \text{IsDuplicate} := \text{IsDuplicate} \cup \{i\}  
\text{end}  
\end{verbatim}

\textbf{B.2.5 Fourth Refinement: M4}  

\textbf{MACHINE} \text{ M4}  
\textbf{REFINES} \text{ M3}  
\textbf{SEES} \text{ C3}  
\textbf{VARIABLES}  
\text{CreateIncident}  
\text{SelectLocation}  
\text{SelectType}  
\text{CompInfo}
Appendix B Fire Dispatch Model with Exception Handling

SetInitialAP
ManageIncident
RequestAdditionalRes
SendStopMsg
CloseIncident
IsDuplicate
ap
notRequired_ap
iLocation
iType
DisablePriorityChange
priority
iLocType

INTEGRATIONS

inv_CreateIncident_type:  CreateIncident ⊆ INCIDENT
inv_SetActionPlan_type:  SetInitialAP ⊆ INCIDENT
inv_RequestAdditionalResources_type:  RequestAdditionalRes ⊆ INCIDENT × LR
inv_SelectType_seq:  SelectType ⊆ CreateIncident
inv_DisablePriorityChange_priority_seq:  DisablePriorityChange ⊆ SelectType
inv_CompInfo_seq:  CompInfo ⊆ (SelectLocation ∩ SelectType)
weak sequencing is used
inv_SetActionPlan_glu:  SetInitialAP = SetInitialAP
inv_SetActionPlan_inSeq:  SetInitialAP ⊆ CompInfo
inv_RequestAdditionalResources_planF_seq:  dom(RequestAdditionalRes) ⊆ SetInitialAP

inv_RequestAdditionalRes_Rep:  ∀i · i ∈ INCIDENT ⇒ RequestAdditionalRes[i] ⊆ LR
Replicator invariant
inv_SendStopMsg_seq:  SendStopMsg ⊆ SetInitialAP
inv_ManageIncident_seq:  ManageIncident ⊆ SetInitialAP
inv_CloseIncident_FD_seq:  CloseIncident ⊆ ManageIncident ∩ SendStopMsg
inv_IsDuplicate_DuplicateInterrupt_seq:  IsDuplicate ⊆ CreateIncident
inv_Interrupt:  partition((CloseIncident∪IsDuplicate), CloseIncident, IsDuplicate)

inv_priority:  priority ∈ SelectType → 1..3

EVENTS

Initialisation
begin
  act_CreateIncident: CreateIncident := ∅
  act_SetInitialAP: SetInitialAP := ∅
  act_ManageIncident: ManageIncident := ∅
  act_CloseIncident: CloseIncident := ∅
  act_IsDuplicate: IsDuplicate := ∅
  act_SendStopMsg: SendStopMsg := ∅
  act_ap: ap := ∅
  act_notRequired_ap: notRequired_ap := ∅
  act_RequestAdditionalRes: RequestAdditionalRes := ∅
  act2: iLocation := ∅
  act3: iType := ∅
  act_loc_type: iLocType := ∅
  act_SelectLocation: SelectLocation := ∅
  act_SelectType: SelectType := ∅
  act_CompInfo: CompInfo := ∅
  act_DisablePriorityChange: DisablePriorityChange := ∅
  act4: priority := ∅
end

Event CreateIncident (ordinary) \(\cong\)
refines CreateIncident

any

where

  grd_self: i \notin CreateIncident

then
  act: CreateIncident := CreateIncident \cup \{i\}

end

Event SelectLocation (ordinary) \(\cong\)
refines SelectLocation

any

i

l

lt

where

  grd_seq.: i \in CreateIncident
  grd_self: i \notin SelectLocation
  grd_DuplicateInterrupt: i \notin IsDuplicate
  grd1: l \in LOCATION
  grd2: lt \in LOCATION\_TYPE

then
\textbf{act}: \ SelectLocation := \ SelectLocation \cup \{i\} \\
\textbf{act1}: \ iLocation(i) := l \\
\textbf{act2}: \ iLocType(i) := lt

\textbf{end}

\textbf{Event} \ SelectType \ (\textsl{ordinary}) \equiv

\textbf{refines} \ SelectType

\textbf{any}

\textbf{i}

\textbf{t}

\textbf{p}

\textbf{where}

\textbf{grd\_seq}: \ i \in CreateIncident \\
\textbf{grd\_self}: \ i \notin SelectType \\
\textbf{grd1}: \ t \in INCIDENT\_TYPE \\
\textbf{grd\_DuplicateInterrupt}: \ i \notin IsDuplicate \\
\textbf{grd2}: \ p = \text{default\_priority}(t)

\textbf{then}

\textbf{act}: \ SelectType := \ SelectType \cup \{i\} \\
\textbf{act1}: \ iType(i) := t \\
\textbf{act2}: \ priority(i) := p

\textbf{end}

\textbf{Event} \ ChangePriority \ (\textsl{ordinary}) \equiv

\textbf{any}

\textbf{i}

\textbf{p}

\textbf{where}

\textbf{grd\_seq\_priority}: \ i \in SelectType \\
\textbf{grd0\_loop}: \ i \notin DisablePriorityChange \\
\textbf{grd1}: \ p \in 1..3 \\
\textbf{grd2}: \ p \neq priority(i) \\
\textbf{grd3}: \ i \notin CloseIncident \\
\quad \text{we can also set DisablePriorityChange in CloseIncident} \\
\textbf{grd\_DuplicateInterrupt}: \ i \notin IsDuplicate

\textbf{then}

\textbf{act1}: \ priority(i) := p

\textbf{end}

\textbf{Event} \ DisablePriorityChange \ (\textsl{ordinary}) \equiv

\textbf{any}

\textbf{i}

\textbf{where}

\textbf{grd\_DuplicateInterrupt}: \ i \notin IsDuplicate
Appendix B Fire Dispatch Model with Exception Handling

grd_seq.priority: \( i \in \text{SelectType} \)
grd_self: \( i \notin \text{DisablePriorityChange} \)
grd1: \( i \notin \text{CloseIncident} \)

we can also set DisablePriorityChange in CloseIncident

then

act: \( \text{DisablePriorityChange} := \text{DisablePriorityChange} \cup \{i\} \)

end

Event CompInfo (ordinary) \( \sqsupseteq \)
refines CompInfo

any

i

where

grd_DuplicateInterrupt: \( i \notin \text{IsDuplicate} \)
grd_seq: \( (i \in \text{SelectLocation} \land i \in \text{SelectType}) \)
grd_self: \( i \notin \text{CompInfo} \)

then

act: \( \text{CompInfo} := \text{CompInfo} \cup \{i\} \)

end

Event SetInitialAP (ordinary) \( \sqsupseteq \)
refines SetInitialAP

any

lrs

i

t

l

where

grd_seq: \( i \in \text{CompInfo} \)
grd_self: \( i \notin \text{SetInitialAP} \)
grd1: \( t = \text{iType}(i) \)
grd2: \( l = \text{iLocType}(i) \)
grd3: \( lrs = \text{PDA}(t \rightarrow l) \)
grd_DuplicateInterrupt: \( i \notin \text{IsDuplicate} \)

then

act: \( \text{SetInitialAP} := \text{SetInitialAP} \cup \{i\} \)
act1: \( \text{ap} := \text{ap} \cup \{i\} \times lrs \)

end

Event RequestAdditionalRes (ordinary) \( \sqsupseteq \)
refines RequestAdditionalRes

any

i
\[ \text{lr} \]

**where**

- \( \text{grd\_seq\_planF} : i \in \text{SetInitialAP} \)
- \( \text{grd\_self} : i \mapsto \text{lr} \notin \text{RequestAdditionalRes} \)
- \( \text{grd\_InputExpression1} : \text{lr} \in LR \)
- \( \text{grd0\_par} : i \notin \text{SendStopMsg} \)
- \( \text{grd\_DuplicateInterrupt} : i \notin \text{IsDuplicate} \)
- \( \text{grd1} : \text{lr} \notin \text{ap}[\{i\}] \)

**then**

- \( \text{act1} : \text{ap} := \text{ap} \cup \{i \mapsto \text{lr}\} \)
- \( \text{act} : \text{RequestAdditionalRes} := \text{RequestAdditionalRes} \cup \{i \mapsto \text{lr}\} \)

**end**

**Event** SendStopMsg (ordinary) \( \Rightarrow \)**

**refines** SendStopMsg

**any**

- \( i \)
- \( \text{lrs} \)

**where**

- \( \text{grd\_lrs} : \text{lrs} \subseteq \text{ap}[\{i\}] \)
- \( \text{grd\_DuplicateInterrupt} : i \notin \text{IsDuplicate} \)
- \( \text{grd\_seq} : i \in \text{SetInitialAP} \)
- \( \text{grd\_self} : i \notin \text{SendStopMsg} \)

**then**

- \( \text{act1} : \text{notRequired\_ap} := \text{notRequired\_ap} \cup (\{i\} \times \text{lrs}) \)
- \( \text{act} : \text{SendStopMsg} := \text{SendStopMsg} \cup \{i\} \)

**end**

**Event** ManageIncident (ordinary) \( \Rightarrow \)**

**refines** ManageIncident

**any**

- \( i \)

**where**

- \( \text{grd\_DuplicateInterrupt} : i \notin \text{IsDuplicate} \)
- \( \text{grd\_seq} : i \in \text{SetInitialAP} \)
- \( \text{grd\_self} : i \notin \text{ManageIncident} \)

**then**

- \( \text{act} : \text{ManageIncident} := \text{ManageIncident} \cup \{i\} \)

**end**

**Event** CloseIncident (ordinary) \( \Rightarrow \)**

**refines** CloseIncident

**any**
\begin{verbatim}

where

\text{grd}\_\text{seq\_FD}: \ (i \in \text{SendStopMsg} \land i \in \text{ManageIncident})
\text{grd}\_\text{self}: \ i \notin \text{CloseIncident}
\text{grd}\_\text{DuplicateInterrupt}: \ i \notin \text{IsDuplicate}

then

\text{act}: \ \text{CloseIncident} := \text{CloseIncident} \cup \{i\}

\end

\textbf{Event} \ \text{IsDuplicate (ordinary)} \ 
\textbf{refines} \ \text{IsDuplicate}

\begin{verbatim}

any

\text{i}

where

\text{grd}\_\text{seq}\_\text{DuplicateInterrupt}: \ i \in \text{CreateIncident}
\text{grd}\_\text{self}: \ i \notin \text{IsDuplicate}
\text{grd1}: \ i \notin \text{CloseIncident}

then

\text{act}: \ \text{IsDuplicate} := \text{IsDuplicate} \cup \{i\}

\end

\end{verbatim}

\textbf{END}

\end{verbatim}

\textbf{B.2.6 Fifth Refinement: M5}

\textbf{MACHINE} \ M5

\textbf{REFINES} \ M4

\textbf{SEES} \ C3

\textbf{VARIABLES}

\begin{verbatim}
CreateIncident
SelectLocation
SelectType
CompInfo
SetInitialAP
ResHandleIncident
IncidentManaged
RequestAdditionalRes
SendStopMsg
CloseIncident
IsDuplicate
ap
\end{verbatim}
notRequired\_ap
iLocation
iType
DisablePriorityChange
priority
iLocType

**INVARIANTS**

\[
\begin{align*}
\text{inv\_CreateIncident\_type: } & \text{CreateIncident} \subseteq INCIDENT \\
\text{inv\_SetActionPlan\_type: } & \text{SetInitialAP} \subseteq INCIDENT \\
\text{inv\_RequestAdditionalResources\_type: } & \text{RequestAdditionalRes} \subseteq INCIDENT \times LR \\
\text{inv\_SelectType\_seq: } & \text{SelectType} \subseteq \text{CreateIncident} \\
\text{inv\_DisablePriorityChange\_priority\_seq: } & \text{DisablePriorityChange} \subseteq \text{SelectType} \\
\text{inv\_CompInfo\_seq: } & \text{CompInfo} \subseteq (\text{SelectLocation} \cap \text{SelectType}) \\
\text{inv\_SetActionPlan\_glu: } & \text{SetInitialAP} = \text{SetInitialAP} \\
\text{inv\_SetActionPlan\_inSeq: } & \text{SetInitialAP} \subseteq \text{CompInfo} \\
\text{inv\_RequestAdditionalResources\_planF\_seq: } & \text{dom}(\text{RequestAdditionalRes}) \subseteq \text{SetInitialAP} \\
\end{align*}
\]

\[
\begin{align*}
\text{inv\_RequestAdditionalRes\_Rep: } & \forall i \in INCIDENT \Rightarrow \text{RequestAdditionalRes}[\{i\}] \subseteq LR \\
\text{Replicator invariant} \\
\text{inv\_SendStopMsg\_seq: } & \text{SendStopMsg} \subseteq \text{SetInitialAP} \\
\text{inv\_HandleIncident\_type: } & \text{ResHandleIncident} \subseteq INCIDENT \times LR \\
\text{inv\_HandleIncident\_seq: } & \text{dom}(\text{ResHandleIncident}) \subseteq \text{SetInitialAP} \\
\text{inv\_IncidentManaged\_seq: } & \text{IncidentManaged} \subseteq \text{SetInitialAP} \\
\text{inv\_IncidentManaged\_glu: } & \text{IncidentManaged} = \text{ManageIncident} \\
\text{inv\_CloseIncident\_FD\_seq: } & \text{CloseIncident} \subseteq \text{IncidentManaged} \cap \text{SendStopMsg} \\
\text{inv\_IsDuplicate\_DuplicateInterrupt\_seq: } & \text{IsDuplicate} \subseteq \text{CreateIncident} \\
\text{inv\_Interrupt: } & \text{partition}((\text{CloseIncident} \cup \text{IsDuplicate}), \text{CloseIncident}, \text{IsDuplicate}) \\
\end{align*}
\]

\[
\begin{align*}
\text{inv\_par\_rep: } & \forall i \in INCIDENT \Rightarrow \text{ResHandleIncident}[\{i\}] \subseteq \text{ap}[\{i\}] \\
\end{align*}
\]

**EVENTS**

**Initialisation**

\[
\begin{align*}
\text{act\_CreateIncident: } & \text{CreateIncident} := \emptyset \\
\text{act\_SetInitialAP: } & \text{SetInitialAP} := \emptyset \\
\text{act\_CloseIncident: } & \text{CloseIncident} := \emptyset \\
\text{act\_IsDuplicate: } & \text{IsDuplicate} := \emptyset \\
\end{align*}
\]
act\_SendStopMsg: SendStopMsg := \emptyset
act\_ap: ap := \emptyset
act\_notRequired\_ap: notRequired\_ap := \emptyset
act\_RequestAdditionalRes: RequestAdditionalRes := \emptyset
act2: iLocation := \emptyset
act\_loc\_type: iLocType := \emptyset
act3: iTyp := \emptyset
act\_SelectLocation: SelectLocation := \emptyset
act\_SelectType: SelectType := \emptyset
act\_CompInfo: CompInfo := \emptyset
act\_DisablePriorityChange: DisablePriorityChange := \emptyset
act4: priority := \emptyset
act\_HandleIncident: ResHandleIncident := \emptyset
act\_IncidentManaged: IncidentManaged := \emptyset

decl
Event CreateIncident (ordinary) ≡
refines CreateIncident
any
i
where
  grd\_self: i \notin CreateIncident
then
  act: CreateIncident := CreateIncident \cup \{i\}
end
Event SelectLocation (ordinary) ≡
refines SelectLocation
any
i
l
where
  grd\_seq.: i \in CreateIncident
  grd\_self: i \notin SelectLocation
  grd\_DuplicateInterrupt: i \notin IsDuplicate
  grd1: l \in LOCATION
  grd2: lt \in LOCATION\_TYPE
then
  act: SelectLocation := SelectLocation \cup \{i\}
  act1: iLocation(i) := l
  act2: iLocType(i) := lt
end
Event SelectType (ordinary) ≡
refines SelectType
   any
   i
t
p
where
grd_seq: i ∈ CreateIncident
grd_self: i ∉ SelectType
grd1: t ∈ INCIDENT_TYPE
grd_DuplicateInterrupt: i ∉ IsDuplicate
grd2: p = default_priority(t)
then
act: SelectType := SelectType ∪ {i}
act1: iType(i) := t
act2: priority(i) := p
end

Event ChangePriority (ordinary) ≡
refines ChangePriority
   any
   i
p
where
grd_seq_priority: i ∈ SelectType
grd0_loop: i ∉ DisablePriorityChange
grd1: p ∈ 1..3
grd2: p ≠ priority(i)
grd3: i ∉ CloseIncident
we can also set DisablePriorityChange in CloseIncident
grd_DuplicateInterrupt: i ∉ IsDuplicate
then
act1: priority(i) := p
end

Event DisablePriorityChange (ordinary) ≡
refines DisablePriorityChange
   any
   i
where
grd_DuplicateInterrupt: i ∉ IsDuplicate
grd_seq_priority: i ∈ SelectType
grd_self: i ∉ DisablePriorityChange
Appendix B Fire Dispatch Model with Exception Handling

\[
\text{grd1: } i \notin \text{CloseIncident}
\]
we can also set DisablePriorityChange in CloseIncident

\[
\text{then}
\]
\[
\text{act: } \text{DisablePriorityChange} := \text{DisablePriorityChange} \cup \{i\}
\]

\end

\textbf{Event} \ CompInfo \langle \text{ordinary} \rangle \triangleq 
\textbf{refines} \ CompInfo
any
i
where
\[
\text{grd\_DuplicateInterrupt: } i \notin \text{IsDuplicate}
\]
\[
\text{grd\_seq: } (i \in \text{SelectLocation} \land i \in \text{SelectType})
\]
\[
\text{grd\_self: } i \notin \text{CompInfo}
\]

\[
\text{then}
\]
\[
\text{act: } \text{CompInfo} := \text{CompInfo} \cup \{i\}
\]

\end

\textbf{Event} \ SetInitialAP \langle \text{ordinary} \rangle \triangleq 
\textbf{refines} \ SetInitialAP
any
lrs
i
t
l
where
\[
\text{grd\_seq: } i \in \text{CompInfo}
\]
\[
\text{grd\_self: } i \notin \text{SetInitialAP}
\]
\[
\text{grd1: } t = \text{iType}(i)
\]
\[
\text{grd2: } l = \text{iLocType}(i)
\]
\[
\text{grd3: } lrs = \text{PDA}(t \mapsto l)
\]
\[
\text{grd\_DuplicateInterrupt: } i \notin \text{IsDuplicate}
\]

\[
\text{then}
\]
\[
\text{act: } \text{SetInitialAP} := \text{SetInitialAP} \cup \{i\}
\]
\[
\text{act1: } \text{ap} := \text{ap} \cup (\{i\} \times lrs)
\]

\end

\textbf{Event} \ RequestAdditionalRes \langle \text{ordinary} \rangle \triangleq 
\textbf{refines} \ RequestAdditionalRes
any
i
lr
where
Appendix B Fire Dispatch Model with Exception Handling

\[
\begin{align*}
\text{grd}_\text{seq_planF}: & \; i \in \text{SetInitialAP} \\
\text{grd}_\text{self}: & \; i \mapsto lr \notin \text{RequestAdditionalRes} \\
\text{grd}_\text{InputExpression1}: & \; lr \in LR \\
\text{grd}_\text{0_par}: & \; i \notin \text{SendStopMsg} \\
\text{grd}_\text{DuplicateInterrupt}: & \; i \notin \text{IsDuplicate} \\
\text{grd1}: & \; lr \notin ap[i] \\
\end{align*}
\]
then
\[
\begin{align*}
\text{act1}: & \; ap := ap \cup \{i \mapsto lr\} \\
\text{act}: & \; \text{RequestAdditionalRes} := \text{RequestAdditionalRes} \cup \{i \mapsto lr\} \\
\end{align*}
\]
end

\[
\text{Event} \; \text{SendStopMsg} \; \langle \text{ordinary} \rangle \;
\]
\[
\text{refines} \; \text{SendStopMsg} \\
\text{any} \\
\begin{align*}
\text{i} \\
\text{lrs}
\end{align*}
where
\[
\begin{align*}
\text{grd}_\text{self}: & \; i \notin \text{SendStopMsg} \\
\text{grd}_\text{lrs}: & \; lrs \subseteq ap[i] \\
\text{grd}_\text{DuplicateInterrupt}: & \; i \notin \text{IsDuplicate} \\
\text{grd}_\text{seq}: & \; i \in \text{SetInitialAP} \\
\end{align*}
\]
then
\[
\begin{align*}
\text{act1}: & \; \text{notRequired}_\text{ap} := \text{notRequired}_\text{ap} \cup (\{i\} \times lrs) \\
\text{act}: & \; \text{SendStopMsg} := \text{SendStopMsg} \cup \{i\} \\
\end{align*}
\]
end

\[
\text{Event} \; \text{ResHandleIncident} \; \langle \text{ordinary} \rangle \;
\]
\[
\text{any} \\
\begin{align*}
\text{lr} \\
\text{i}
\end{align*}
where
\[
\begin{align*}
\text{grd}_\text{0_par}: & \; i \notin \text{IncidentManaged} \\
\text{grd}_\text{DuplicateInterrupt}: & \; i \notin \text{IsDuplicate} \\
\text{grd}_\text{seq}: & \; i \in \text{SetInitialAP} \\
\text{grd}_\text{self}: & \; i \mapsto lr \notin \text{ResHandleIncident} \\
\text{grd}_\text{InputExpression1}: & \; lr \in ap[i] \\
\end{align*}
\]
then
\[
\begin{align*}
\text{act}: & \; \text{ResHandleIncident} := \text{ResHandleIncident} \cup \{i \mapsto lr\} \\
\end{align*}
\]
end

\[
\text{Event} \; \text{IncidentManaged} \; \langle \text{ordinary} \rangle \;
\]
\[
\text{refines} \; \text{ManageIncident} \\
\text{any} \\
\begin{align*}
\text{i}
\end{align*}
where
\begin{align*}
\text{grd} \_ \text{seq} & : \quad i \in \text{SetInitialAP} \\
\text{grd} \_ \text{self} & : \quad i \notin \text{IncidentManaged} \\
\text{grd} \_ \text{DuplicateInterrupt} & : \quad i \notin \text{IsDuplicate}
\end{align*}
then
\begin{align*}
\text{act} & : \quad \text{IncidentManaged} := \text{IncidentManaged} \cup \{i\}
\end{align*}
end

Event CloseIncident (ordinary) \overset{\cong}{=} refines CloseIncident

any
\begin{align*}
i
\end{align*}
where
\begin{align*}
\text{grd} \_ \text{DuplicateInterrupt} & : \quad i \notin \text{IsDuplicate} \\
\text{grd} \_ \text{seq} \_ \text{FD} & : \quad (i \in \text{SendStopMsg} \land i \in \text{IncidentManaged}) \\
\text{grd} \_ \text{self} & : \quad i \notin \text{CloseIncident}
\end{align*}
then
\begin{align*}
\text{act} & : \quad \text{CloseIncident} := \text{CloseIncident} \cup \{i\}
\end{align*}
end

Event IsDuplicate (ordinary) \overset{\cong}{=} refines IsDuplicate

any
\begin{align*}
i
\end{align*}
where
\begin{align*}
\text{grd} & : \quad i \notin \text{CloseIncident} \\
\text{grd} \_ \text{seq} \_ \text{DuplicateInterrupt} & : \quad i \in \text{CreateIncident} \\
\text{grd} \_ \text{self} & : \quad i \notin \text{IsDuplicate}
\end{align*}
then
\begin{align*}
\text{act} & : \quad \text{IsDuplicate} := \text{IsDuplicate} \cup \{i\}
\end{align*}
end

END

B.2.7 Sixth Refinement: M6

MACHINE M6
REFINES M5
SEES C3
VARIABLES
CreateIncident
SelectLocation
SelectType
CompInfo
SetInitialAP
HandleIncident
IncidentManaged
RequestAdditionalRes
SendStopMsg
CloseIncident
IsDuplicate
ap
notRequired_ap
iLocation
iType
DisablePriorityChange
priority
ResLost
ResCompTask
iLocType

INVIARANTS

\textit{inv\_CreateIncident\_type}: \textit{CreateIncident} \subseteq INCIDENT
\textit{inv\_SetActionPlan\_type}: \textit{SetInitialAP} \subseteq INCIDENT
\textit{inv\_RequestAdditionalResources\_type}: \textit{RequestAdditionalRes} \subseteq INCIDENT \times LR
\textit{inv\_SelectType\_seq}: \textit{SelectType} \subseteq \textit{CreateIncident}
\textit{inv\_DisablePriorityChange\_priority\_seq}: \textit{DisablePriorityChange} \subseteq \textit{SelectType}
\textit{inv\_CompInfo\_seq}: \textit{CompInfo} \subseteq (\textit{SelectLocation} \cap \textit{SelectType})
\textit{inv\_SetActionPlan\_glu}: \textit{SetInitialAP} = \textit{SetInitialAP}
\textit{inv\_SetActionPlan\_inSeq}: \textit{SetInitialAP} \subseteq \textit{CompInfo}
\textit{inv\_RequestAdditionalResources\_planF\_seq}: \textit{dom}(\textit{RequestAdditionalRes}) \subseteq \textit{SetInitialAP}

\textit{inv\_RequestAdditionalRes\_Rep}: \forall i \cdot i \in INCIDENT \Rightarrow \textit{RequestAdditionalRes}[i] \subseteq LR

Replicator invariant
\textit{inv\_SendStopMsg\_seq}: \textit{SendStopMsg} \subseteq \textit{SetInitialAP}
\textit{inv\_HandleIncident\_type}: \textit{HandleIncident} \subseteq INCIDENT \times LR
\textit{inv\_HandleIncident\_seq}: \textit{dom}(\textit{HandleIncident}) \subseteq \textit{SetInitialAP}
\textit{inv\_CompTask\_glu}: \textit{ResCompTask} = \textit{ResHandleIncident}
\textit{inv\_CompTask\_seq}: \textit{ResCompTask} \subseteq \textit{HandleIncident}
\textit{inv\_IncidentManaged\_seq}: \textit{IncidentManaged} \subseteq \textit{SetInitialAP}
\textbf{inv\_CloseIncident\_FD\_seq:} \quad \text{CloseIncident} \subseteq \text{IncidentManaged} \cap \text{SendStopMsg}

\textbf{inv\_IsDuplicate\_DuplicateInterrupt\_seq:} \quad \text{IsDuplicate} \subseteq \text{CreateIncident}

\textbf{inv\_ResLost\_type:} \quad \text{ResLost} \subseteq \text{INCIDENT} \times \text{LR}

\textbf{inv\_ResLost:} \quad \text{dom(ResLost)} \subseteq \text{SetInitialAP}

\textbf{inv\_ResLost\_retry:} \quad \text{partition}((\text{HandleIncident} \cup \text{ResLost}), \text{HandleIncident}, \text{ResLost})

\textbf{inv\_IncidentManaged\_par:} \quad \forall i \cdot i \in \text{IncidentManaged} \Rightarrow \text{HandleIncident}[i] = \text{ResCompTask}[i]

\textbf{inv\_par\_rep:} \quad \forall i \cdot i \in \text{INCIDENT} \Rightarrow \text{HandleIncident}[i] \subseteq \text{ap}[i]

\textbf{EVENTS}

\textbf{Initialisation}

\textbf{begin}

\quad \text{act\_CreateIncident:} \quad \text{CreateIncident} := \emptyset

\quad \text{act\_SetInitialAP:} \quad \text{SetInitialAP} := \emptyset

\quad \text{act\_CloseIncident:} \quad \text{CloseIncident} := \emptyset

\quad \text{act\_IsDuplicate:} \quad \text{IsDuplicate} := \emptyset

\quad \text{act\_SendStopMsg:} \quad \text{SendStopMsg} := \emptyset

\quad \text{act\_ap:} \quad \text{ap} := \emptyset

\quad \text{act\_notRequired\_ap:} \quad \text{notRequired\_ap} := \emptyset

\quad \text{act\_RequestAdditionalRes:} \quad \text{RequestAdditionalRes} := \emptyset

\quad \text{act2:} \quad \text{iLocation} := \emptyset

\quad \text{act\_loc\_type:} \quad \text{iLocType} := \emptyset

\quad \text{act3:} \quad \text{iType} := \emptyset

\quad \text{act\_SelectLocation:} \quad \text{SelectLocation} := \emptyset

\quad \text{act\_SelectType:} \quad \text{SelectType} := \emptyset

\quad \text{act\_CompInfo:} \quad \text{CompInfo} := \emptyset

\quad \text{act\_DisablePriorityChange:} \quad \text{DisablePriorityChange} := \emptyset

\quad \text{act4:} \quad \text{priority} := \emptyset

\quad \text{act\_ResLost:} \quad \text{ResLost} := \emptyset

\quad \text{act\_HandleIncident:} \quad \text{HandleIncident} := \emptyset

\quad \text{act\_IncidentManaged:} \quad \text{IncidentManaged} := \emptyset

\quad \text{act\_ResCompTask:} \quad \text{ResCompTask} := \emptyset

\textbf{end}

\textbf{Event} \quad \text{CreateIncident \hspace{1em} (ordinary) } \triangleq

\textbf{refines} \quad \text{CreateIncident}

\textbf{any}

\quad i

\textbf{where}

\quad \text{grd\_self:} \quad i \notin \text{CreateIncident}

\textbf{then}
\textbf{Event} SelectLocation (ordinary) \supseteq \textbf{refines} SelectLocation

\framebox{\textbf{any}}

\framebox{i}
\framebox{l}
\framebox{lt}

\textbf{where}

\framebox{grd\_seq.: } i \in \text{CreateIncident}
\framebox{grd\_self: } i \notin \text{SelectLocation}
\framebox{grd\_DuplicateInterrupt: } i \notin \text{IsDuplicate}
\framebox{grd1: } l \in \text{LOCATION}
\framebox{grd2: } lt \in \text{LOCATION\_TYPE}

\textbf{then}

\framebox{act: } \text{SelectLocation} := \text{SelectLocation} \cup \{i\}
\framebox{act1: } i\text{Location}(i) := l
\framebox{act2: } i\text{LocType}(i) := lt

\textbf{end}

\textbf{Event} SelectType (ordinary) \supseteq \textbf{refines} SelectType

\framebox{\textbf{any}}

\framebox{i}
\framebox{t}
\framebox{p}

\textbf{where}

\framebox{grd\_seq.: } i \in \text{CreateIncident}
\framebox{grd\_self: } i \notin \text{SelectType}
\framebox{grd1: } t \in \text{INCIDENT\_TYPE}
\framebox{grd\_DuplicateInterrupt: } i \notin \text{IsDuplicate}
\framebox{grd2: } p = \text{default\_priority}(t)

\textbf{then}

\framebox{act: } \text{SelectType} := \text{SelectType} \cup \{i\}
\framebox{act1: } i\text{Type}(i) := t
\framebox{act2: } \text{priority}(i) := p

\textbf{end}

\textbf{Event} ChangePriority (ordinary) \supseteq \textbf{refines} ChangePriority

\framebox{\textbf{any}}

\framebox{i}
p

where

\text{grd\_seq\_priority}: \quad i \in \text{SelectType}
\text{grd0\_loop}: \quad i \notin \text{DisablePriorityChange}
\text{grd1}: \quad p \in 1..3
\text{grd2}: \quad p \neq \text{priority}(i)
\text{grd3}: \quad i \notin \text{CloseIncident}

we can also set DisablePriorityChange in CloseIncident

\text{grd\_DuplicateInterrupt}: \quad i \notin \text{IsDuplicate}

then

act1: \quad \text{priority}(i) := p

end

\text{Event DisablePriorityChange \{ordinary\} \triangleq}
\text{refines DisablePriorityChange}

\text{any}

i

where

\text{grd\_DuplicateInterrupt}: \quad i \notin \text{IsDuplicate}
\text{grd\_seq\_priority}: \quad i \in \text{SelectType}
\text{grd\_self}: \quad i \notin \text{DisablePriorityChange}
\text{grd1}: \quad i \notin \text{CloseIncident}

we can also set DisablePriorityChange in CloseIncident

then

act: \quad \text{DisablePriorityChange} := \text{DisablePriorityChange} \cup \{i\}

end

\text{Event CompInfo \{ordinary\} \triangleq}
\text{refines CompInfo}

\text{any}

i

where

\text{grd\_DuplicateInterrupt}: \quad i \notin \text{IsDuplicate}
\text{grd\_seq}: \quad (i \in \text{SelectLocation} \land i \in \text{SelectType})
\text{grd\_self}: \quad i \notin \text{CompInfo}

then

act: \quad \text{CompInfo} := \text{CompInfo} \cup \{i\}

end

\text{Event SetInitialAP \{ordinary\} \triangleq}
\text{refines SetInitialAP}

\text{any}

lrs
\[
\begin{align*}
\text{where} & \quad \text{grd\_seq} : \; i \in \text{CompInfo} \\
\text{grd\_self} : \; i \notin \text{SetInitialAP} \\
\text{grd1} : \; t = \text{tType}(i) \\
\text{grd2} : \; l = \text{iLocType}(i) \\
\text{grd3} : \; \text{lr} = \text{PDA}(t \mapsto l) \\
\text{DuplicateInterrupt} : \; i \notin \text{IsDuplicate} \\
\text{then} & \quad \text{act} : \; \text{SetInitialAP} := \text{SetInitialAP} \cup \{i\} \\
\text{act1} : \; \text{ap} := \text{ap} \cup (\{i\} \times \text{lr}) \\
\text{end}
\end{align*}
\]

**Event** RequestAdditionalRes \(\text{(ordinary)}\) \(\bowtie\)

**refines** RequestAdditionalRes

any
i
\text{lr}

\text{where}
\text{grd\_seq\_planF} : \; i \in \text{SetInitialAP} \\
\text{grd\_self} : \; i \mapsto \text{lr} \notin \text{RequestAdditionalRes} \\
\text{grd\_InputExpression1} : \; \text{lr} \in \text{LR} \\
\text{grd0\_par} : \; i \notin \text{SendStopMsg} \\
\text{DuplicateInterrupt} : \; i \notin \text{IsDuplicate} \\
\text{grd1} : \; \text{lr} \notin \text{ap}(\{i\}) \\

\text{then} & \quad \text{act1} : \; \text{ap} := \text{ap} \cup (\{i\} \mapsto \text{lr}) \\
\text{act} : \; \text{RequestAdditionalRes} := \text{RequestAdditionalRes} \cup (\{i\mapsto \text{lr}\}) \\
\text{end}

**Event** SendStopMsg \(\text{(ordinary)}\) \(\bowtie\)

**refines** SendStopMsg

any
i
\text{ls}

\text{where}
\text{grd\_self} : \; i \notin \text{SendStopMsg} \\
\text{grd\_ls} : \; \text{lr} \subseteq \text{ap}(\{i\}) \\
\text{DuplicateInterrupt} : \; i \notin \text{IsDuplicate} \\
\text{grd\_seq} : \; i \in \text{SetInitialAP} \\

\text{then}
\texttt{act1: notRequired\_ap := notRequired\_ap \cup \{i\} \times lrs}
\texttt{act: SendStopMsg := SendStopMsg \cup \{i\}}
\texttt{end}

\textbf{Event} HandleIncident (\textit{ordinary}) \triangleq
\texttt{any}
\texttt{lr}
\texttt{i}
\texttt{where}
\texttt{grd\_ResLostInterrupt: i \mapsto lr \notin ResLost}
\texttt{grd\_DuplicateInterrupt: i \notin IsDuplicate}
\texttt{grd\_seq.: i \in SetInitialAP}
\texttt{grd\_self: i \mapsto lr \notin HandleIncident}
\texttt{grd\_InputExpression1: lr \in ap\{i\}}
\texttt{grd0\_par: i \notin IncidentManaged}
\texttt{then}
\texttt{act: HandleIncident := HandleIncident \cup \{i \mapsto lr\}}
\texttt{end}

\textbf{Event} ResCompTask (\textit{ordinary}) \triangleq
\textbf{refines} ResHandleIncident
\texttt{any}
\texttt{lr}
\texttt{i}
\texttt{where}
\texttt{grd\_self: i \mapsto lr \notin ResCompTask}
\texttt{grd\_DuplicateInterrupt: i \notin IsDuplicate}
\texttt{grd\_seq.: i \mapsto lr \in HandleIncident}
\texttt{then}
\texttt{act: ResCompTask := ResCompTask \cup \{i \mapsto lr\}}
\texttt{end}

\textbf{Event} ResLost (\textit{ordinary}) \triangleq
\texttt{any}
\texttt{i}
\texttt{lr}
\texttt{where}
\texttt{grd\_retry: i \mapsto lr \notin HandleIncident}
\texttt{grd\_self: i \mapsto lr \notin ResLost}
\texttt{grd\_seq: i \in SetInitialAP}
\texttt{grd\_par: i \notin IncidentManaged}
\texttt{grd\_DuplicateInterrupt: i \notin IsDuplicate}
\texttt{then}
\texttt{act: ResLost := ResLost \cup \{i \mapsto lr\}
\[ \textbf{act1}: \text{HandleIncident} := \text{HandleIncident} \setminus \{i \mapsto lr\} \]

\textbf{Event} Reset_ResLost \langle \text{ordinary} \rangle \triangleq

\begin{verbatim}
any
\hspace{1em} i
\hspace{1em} lr
\textbf{where}
\hspace{1em} \text{grd}_\text{seq}: \ i \mapsto lr \in \text{ResLost}
\textbf{then}
\hspace{1em} \textbf{act}: \text{ResLost} := \text{ResLost} \setminus \{i \mapsto lr\} 
\end{verbatim}

\textbf{end}

\textbf{Event} IncidentManaged \langle \text{ordinary} \rangle \triangleq

\textbf{refines} IncidentManaged

\begin{verbatim}
any
\hspace{1em} i
\textbf{where}
\hspace{1em} \text{grd}_\text{par}: \text{HandleIncident}[[i]] = \text{ResCompTask}[[i]]
\hspace{1em} \text{grd}_\text{self}: \ i \notin \text{IncidentManaged}
\hspace{1em} \text{grd}_\text{DuplicateInterrupt}: \ i \notin \text{IsDuplicate}
\hspace{1em} \text{grd}_\text{seq}: \ i \in \text{SetInitialAP}
\textbf{then}
\hspace{1em} \textbf{act}: \text{IncidentManaged} := \text{IncidentManaged} \cup \{i\}
\end{verbatim}

\textbf{end}

\textbf{Event} CloseIncident \langle \text{ordinary} \rangle \triangleq

\textbf{refines} CloseIncident

\begin{verbatim}
any
\hspace{1em} i
\textbf{where}
\hspace{1em} \text{grd}_\text{DuplicateInterrupt}: \ i \notin \text{IsDuplicate}
\hspace{1em} \text{grd}_\text{seq}_\text{FD}: \ (i \in \text{SendStopMsg} \land i \in \text{IncidentManaged})
\hspace{1em} \text{grd}_\text{self}: \ i \notin \text{CloseIncident}
\textbf{then}
\hspace{1em} \textbf{act}: \text{CloseIncident} := \text{CloseIncident} \cup \{i\}
\end{verbatim}

\textbf{end}

\textbf{Event} IsDuplicate \langle \text{ordinary} \rangle \triangleq

\textbf{refines} IsDuplicate

\begin{verbatim}
any
\hspace{1em} i
\textbf{where}
\hspace{1em} \text{grd1}: \ i \notin \text{CloseIncident}
\end{verbatim}
Appendix B Fire Dispatch Model with Exception Handling

$\text{grd_seq.DuplicateInterrupt: } i \in \text{CreateIncident}$
$\text{grd_self: } i \notin \text{IsDuplicate}$

$\text{then}$
$\text{act: } \text{IsDuplicate} := \text{IsDuplicate} \cup \{i\}$

$\text{end}$

END

B.2.8 Seventh Refinement: M7

MACHINE M7
REFINES M6
SEES C3
VARIABLES
CreateIncident
SelectLocation
SelectType
CompInfo
SetInitialAP
AllocateRes
ResAttend
DeallocateRes
StopMsgReceived
IncidentManaged
RequestAdditionalRes
SendStopMsg
CloseIncident
IsDuplicate
ap
notRequired_ap
iLocation
iType
DisablePriorityChange
priority
RemoveAlloc
RemoveBrokenRes
ResCompTask
iLocType

INVARIANTS
inv\_CreateIncident\_type: \( \text{CreateIncident} \subseteq \text{INCIDENT} \)

inv\_SetActionPlan\_type: \( \text{SetInitialAP} \subseteq \text{INCIDENT} \)

inv\_RequestAdditionalResources\_type: \( \text{RequestAdditionalRes} \subseteq \text{INCIDENT} \times \text{LR} \)

inv\_SelectType\_seq: \( \text{SelectType} \subseteq \text{CreateIncident} \)

inv\_DisablePriorityChange\_priority\_seq: \( \text{DisablePriorityChange} \subseteq \text{SelectType} \)

inv\_CompInfo\_seq: \( \text{CompInfo} \subseteq (\text{SelectLocation} \cap \text{SelectType}) \)

inv\_SetActionPlan\_glu: \( \text{SetInitialAP} = \text{SetInitialAP} \)

inv\_SetActionPlan\_inSeq: \( \text{SetInitialAP} \subseteq \text{CompInfo} \)

inv\_RequestAdditionalResources\_planF\_seq: \( \text{dom}(\text{RequestAdditionalRes}) \subseteq \text{SetInitialAP} \)

inv\_RequestAdditionalRes\_Rep: \( \forall i \in \text{INCIDENT} \Rightarrow \text{RequestAdditionalRes}[\{i\}] \subseteq \text{LR} \)

Replicator invariant

inv\_SendStopMsg\_seq: \( \text{SendStopMsg} \subseteq \text{SetInitialAP} \)

inv\_AllocateRes\_type: \( \text{AllocateRes} \subseteq \text{INCIDENT} \times \text{LR} \)

inv\_AllocateRes\_allocation\_seq: \( \text{dom}(\text{AllocateRes}) \subseteq \text{SetInitialAP} \)

inv\_Attend\_allocation\_seq: \( \text{ResAttend} \subseteq \text{AllocateRes} \)

inv\_StopMsgReceived\_seq: \( \text{StopMsgReceived} \subseteq \text{AllocateRes} \)

inv\_DeallocateRes\_seq: \( \text{DeallocateRes} \subseteq (\text{ResAttend} \cup \text{StopMsgReceived}) \)

inv\_DeallocateRes\_glu: \( \text{DeallocateRes} = \text{HandleIncident} \)

inv\_xorFD: \( \text{partition}((\text{ResAttend} \cup \text{StopMsgReceived}), \text{ResAttend}, \text{StopMsgReceived}) \)

inv\_CompTask\_seq: \( \text{ResCompTask} \subseteq \text{DeallocateRes} \)

inv\_IncidentManaged\_seq: \( \text{IncidentManaged} \subseteq \text{SetInitialAP} \)

inv\_CloseIncident\_FD\_seq: \( \text{CloseIncident} \subseteq \text{IncidentManaged} \cap \text{SendStopMsg} \)

inv\_IsDuplicate\_DuplicateInterrupt\_seq: \( \text{IsDuplicate} \subseteq \text{CreateIncident} \)

inv\_ResLost\_gluing: \( \text{partition}(\text{ResLost}, \text{RemoveAlloc}, \text{RemoveBrokenRes}) \)

inv\_RemoveAlloc\_seq: \( \text{dom}(\text{RemoveAlloc}) \subseteq \text{SetInitialAP} \)

inv\_RemoveBrokenRes\_seq: \( \text{dom}(\text{RemoveBrokenRes}) \subseteq \text{SetInitialAP} \)

inv\_ResLost\_retry: \( \text{partition}((\text{DeallocateRes} \cup (\text{RemoveAlloc} \cup \text{RemoveBrokenRes})), \text{DeallocateRes}, (\text{RemoveAlloc} \cup \text{RemoveBrokenRes})) \)

in\_rep: \( \forall i \in \text{INCIDENT} \Rightarrow \text{AllocateRes}[\{i\}] \subseteq \text{ap}[\{i\}] \)

in\_par: \( \forall i \in \text{IncidentManaged} \Rightarrow \text{AllocateRes}[\{i\}] = \text{ResCompTask}[\{i\}] \)

in\_managed: \( \forall i \in \text{CloseIncident} \Rightarrow \text{ap}[\{i\}] \setminus \text{notRequired}_\text{ap}[\{i\}] \subseteq \text{DeallocateRes}[\{i\}] \)

manually added

EVENTS
Initialisation

begin
  act_CreateIncident: CreateIncident := ∅
  act_SetInitialAP: SetInitialAP := ∅
  act_CloseIncident: CloseIncident := ∅
  act_IsDuplicate: IsDuplicate := ∅
  act_SendStopMsg: SendStopMsg := ∅
  act_ap: ap := ∅
  act_notRequired_ap: notRequiredAp := ∅
  act_RequestAdditionalRes: RequestAdditionalRes := ∅
  act2: iLocation := ∅
  act_loc_type: iLocType := ∅
  act3: iType := ∅
  act_SelectLocation: SelectLocation := ∅
  act_SelectType: SelectType := ∅
  act_ComplInfo: CompInfo := ∅
  act_DisablePriorityChange: DisablePriorityChange := ∅
  act4: priority := ∅
  act_StopMsgReceived: StopMsgReceived := ∅
  act_AllocateRes: AllocateRes := ∅
  act_Attend: ResAttend := ∅
  act_DeallocateRes: DeallocateRes := ∅
  act_IncidentManaged: IncidentManaged := ∅
  act_RemoveAlloc: RemoveAlloc := ∅
  act_RemoveBrokenRes: RemoveBrokenRes := ∅
  act_ResCompTask: ResCompTask := ∅

end

Event CreateIncident (ordinary) ≡
refines CreateIncident

any
  i
where
grd_self: i /∈ CreateIncident
then
  act: CreateIncident := CreateIncident ∪ {i}
end

Event SelectLocation (ordinary) ≡
refines SelectLocation

any
  i

lt

where

\( \text{grd\_seq} : i \in \text{CreateIncident} \)
\( \text{grd\_self} : i \notin \text{SelectLocation} \)
\( \text{grd\_DuplicateInterrupt} : i \notin \text{IsDuplicate} \)
\( \text{grd1} : l \in \text{LOCATION} \)
\( \text{grd2} : lt \in \text{LOCATION\_TYPE} \)

then

\( \text{act} : \text{SelectLocation} := \text{SelectLocation} \cup \{i\} \)
\( \text{act1} : \text{Location}(i) := l \)
\( \text{act2} : \text{LocType}(i) := lt \)

end

**Event** \( \text{SelectType} \) (\text{ordinary}) \( \models \)

**refines** \( \text{SelectType} \)

\( \text{any} \)

\( i \)
\( t \)
\( p \)

where

\( \text{grd\_seq} : i \in \text{CreateIncident} \)
\( \text{grd\_self} : i \notin \text{SelectType} \)
\( \text{grd1} : t \in \text{INCIDENT\_TYPE} \)
\( \text{grd\_DuplicateInterrupt} : i \notin \text{IsDuplicate} \)
\( \text{grd2} : p = \text{default\_priority}(t) \)

then

\( \text{act} : \text{SelectType} := \text{SelectType} \cup \{i\} \)
\( \text{act1} : \text{Type}(i) := t \)
\( \text{act2} : \text{priority}(i) := p \)

end

**Event** \( \text{ChangePriority} \) (\text{ordinary}) \( \models \)

**refines** \( \text{ChangePriority} \)

\( \text{any} \)

\( i \)
\( p \)

where

\( \text{grd\_seq\_priority} : i \in \text{SelectType} \)
\( \text{grd\_loop} : i \notin \text{DisablePriorityChange} \)
\( \text{grd1} : p \in 1..3 \)
\( \text{grd2} : p \neq \text{priority}(i) \)
\( \text{grd3} : i \notin \text{CloseIncident} \)

we can also set \( \text{DisablePriorityChange} \) in \( \text{CloseIncident} \)
\[ \text{grd\_DuplicateInterrupt: } i \notin \text{IsDuplicate} \]

then

\[ \text{act}_1: \text{priority}(i) := p \]

end

Event DisablePriorityChange \langle\text{ordinary}\rangle \triangleq

refines DisablePriorityChange

any 

i

where

\[ \text{grd\_DuplicateInterrupt: } i \notin \text{IsDuplicate} \]

\[ \text{grd\_seq\_priority: } i \in \text{SelectType} \]

\[ \text{grd\_self: } i \notin \text{DisablePriorityChange} \]

\[ \text{grd}_1: i \notin \text{CloseIncident} \]

we can also set DisablePriorityChange in CloseIncident

then

\[ \text{act: } \text{DisablePriorityChange} := \text{DisablePriorityChange} \cup \{i\} \]

end

Event CompInfo \langle\text{ordinary}\rangle \triangleq

refines CompInfo

any 

i

where

\[ \text{grd\_DuplicateInterrupt: } i \notin \text{IsDuplicate} \]

\[ \text{grd\_seq\_priority: } (i \in \text{SelectLocation} \land i \in \text{SelectType}) \]

\[ \text{grd\_self: } i \notin \text{CompInfo} \]

then

\[ \text{act: } \text{CompInfo} := \text{CompInfo} \cup \{i\} \]

end

Event SetInitialAP \langle\text{ordinary}\rangle \triangleq

refines SetInitialAP

any

\[ \text{lrs} \]

\[ i \]

\[ t \]

\[ l \]

where

\[ \text{grd\_seq: } i \in \text{CompInfo} \]

\[ \text{grd\_self: } i \notin \text{SetInitialAP} \]

\[ \text{grd}_1: t = i\text{Type}(i) \]

\[ \text{grd}_2: l = i\text{LocType}(i) \]
Appendix B Fire Dispatch Model with Exception Handling

\[ \text{grd3: } lrs = PDA(t \mapsto l) \]

\[ \text{grd\_DuplicateInterrupt: } i \notin IsDuplicate } \]

then

\[ \text{act: } SetInitialAP := SetInitialAP \cup \{i\} \]

\[ \text{act1: } ap := ap \cup \{(i) \times lrs\} \]

end

**Event** RequestAdditionalRes (ordinary) \(\cong\)

**refines** RequestAdditionalRes

any

\[ \text{i} \]

\[ \text{lr} \]

where

\[ \text{grd\_seq\_planF: } i \in SetInitialAP \]

\[ \text{grd\_self: } i \mapsto lr \notin RequestAdditionalRes \]

\[ \text{grd\_InputExpression1: } lr \in LR \setminus ap[i] \]

\[ \text{grd\_par: } i \notin SendStopMsg \]

\[ \text{grd\_DuplicateInterrupt: } i \notin IsDuplicate \]

\[ \text{grd1: } lr \notin ap[i] \]

added manually

then

\[ \text{act1: } ap := ap \cup \{i \mapsto lr\} \]

\[ \text{act: } RequestAdditionalRes := RequestAdditionalRes \cup \{i \mapsto lr\} \]

end

**Event** SendStopMsg (ordinary) \(\cong\)

**refines** SendStopMsg

any

\[ \text{i} \]

\[ \text{lrs} \]

where

\[ \text{grd\_self: } i \notin SendStopMsg \]

\[ \text{grd\_lrs: } lrs = ap[i] \setminus ResAttend[i] \]

\[ \text{grd\_DuplicateInterrupt: } i \notin IsDuplicate \]

\[ \text{grd\_seq: } i \in SetInitialAP \]

then

\[ \text{act1: } notRequired\_ap := notRequired\_ap \cup \{(i) \times lrs\} \]

\[ \text{act: } SendStopMsg := SendStopMsg \cup \{i\} \]

end

**Event** AllocateRes (ordinary) \(\cong\)

any

\[ \text{i} \]

\[ \text{lr} \]
Appendix B Fire Dispatch Model with Exception Handling

where

\[ \text{grdResLostRetry: } i \mapsto lr \notin (\text{RemoveAlloc} \cup \text{RemoveBrokenRes}) \]
\[ \text{grd_seq_allocation: } i \in \text{SetInitialAP} \]
\[ \text{grd_self: } i \mapsto lr \notin \text{AllocateRes} \]
\[ \text{grd_InputExpression1: } lr \in \text{ap}[i] \]
\[ \text{grd0_par: } i \notin \text{IncidentManaged} \]
\[ \text{grd_DuplicateInterrupt: } i \notin \text{IsDuplicate} \]
\[ \text{grd1: } i \mapsto lr \notin \text{notRequired_ap} \]

then

\[ \text{act: } \text{AllocateRes} := \text{AllocateRes} \cup \{i \mapsto lr\} \]

end

Event UpdateResLocation (ordinary) \( \triangleq \)

any

\[ i \]
\[ lr \]

where

\[ \text{grdResLostRetry: } i \mapsto lr \notin (\text{RemoveAlloc} \cup \text{RemoveBrokenRes}) \]
\[ \text{grd_seq_allocation: } i \mapsto lr \in \text{AllocateRes} \]
\[ \text{grd0_loop: } i \mapsto lr \notin (\text{ResAttend} \cup \text{StopMsgReceived}) \]
\[ \text{@grdResLostRetry } i \mapsto lr \notin \text{ResLost} \]
\[ \text{grd_DuplicateInterrupt: } i \notin \text{IsDuplicate} \]
\[ \text{grd1: } i \mapsto lr \notin \text{notRequired_ap} \]

then

\[ \text{skip} \]

end

Event ReallocateQuickerRes (ordinary) \( \triangleq \)

any

\[ i \]
\[ lr \]

where

\[ \text{grdResLostRetry: } i \mapsto lr \notin (\text{RemoveAlloc} \cup \text{RemoveBrokenRes}) \]
\[ \text{grd_seq_allocation: } i \mapsto lr \in \text{AllocateRes} \]
\[ \text{grd0_loop: } i \mapsto lr \notin (\text{ResAttend} \cup \text{StopMsgReceived}) \]
\[ \text{@grdResLostRetry } i \mapsto lr \notin \text{ResLost} \]
\[ \text{grd_DuplicateInterrupt: } i \notin \text{IsDuplicate} \]
\[ \text{grd1: } i \mapsto lr \notin \text{notRequired_ap} \]

then

\[ \text{skip} \]

end

Event StopMsgReceived (ordinary) \( \triangleq \)

any
lr
i

where
grd\_InputExpression1: \( lr \in ap\{i\} \)
\[ grd\_seq\_allocation: \quad i \mapsto lr \in AllocateRes \]
\[ grd\_self: \quad i \mapsto lr \notin StopMsgReceived \]
\[ @grdResLostRetry i \mapsto lr \notin ResLost \]
\[ grd\_DuplicateInterrupt: \quad i \notin IsDuplicate \]
\[ grd\_xor: \quad i \mapsto lr \notin ResAttend \]
\[ grd1: \quad i \mapsto lr \notin notRequired\_ap \]
\[ grd\_ResLostRetry: \quad i \mapsto lr \notin (RemoveAlloc \cup RemoveBrokenRes) \]
\[ grd\_seq:\quad i \in SetInitialAP \]
then
\[ act: \quad StopMsgReceived := StopMsgReceived \cup \{i \mapsto lr\} \]
end

\textbf{Event} ResAttend (ordinary) \( \triangleq \)

any
i
lr

where
\[ grd\_ResLostRetry: \quad i \mapsto lr \notin (RemoveAlloc \cup RemoveBrokenRes) \]
\[ grd\_InputExpression1: \quad lr \in ap\{i\} \]
\[ grd\_seq\_allocation: \quad i \mapsto lr \in AllocateRes \]
\[ grd\_self: \quad i \mapsto lr \notin ResAttend \]
\[ @grdResLostRetry i \mapsto lr \notin ResLost \]
\[ grd\_DuplicateInterrupt: \quad i \notin IsDuplicate \]
\[ grd\_xor: \quad i \mapsto lr \notin StopMsgReceived \]
\[ grd\_seq:\quad i \in SetInitialAP \]
\[ grd1: \quad i \mapsto lr \notin notRequired\_ap \]
then
\[ act: \quad ResAttend := ResAttend \cup \{i \mapsto lr\} \]
end

\textbf{Event} DeallocateRes (ordinary) \( \triangleq \)

refines HandleIncident

any
lr
i

where
\[ grd\_ResLostRetry: \quad i \mapsto lr \notin (RemoveAlloc \cup RemoveBrokenRes) \]
\[ grd\_seq:\quad i \mapsto lr \in (ResAttend \cup StopMsgReceived) \]
grd\_self: i \mapsto lr \notin \text{DeallocateRes}

@grdResLostRetry i \mapsto lr \notin \text{ResLost}

grd\_DuplicateInterrupt: i \notin \text{IsDuplicate}

@grd1 lr \in \text{ap}\{[i]\} // \text{GRD PO} // @grdResLostRetry i \mapsto lr \notin \text{ResLost}

\text{then}

\text{act: } \text{DeallocateRes} := \text{DeallocateRes} \cup \{i \mapsto lr\}

\text{end}

\text{Event: IncidentManaged (ordinary) } \triangleq \text{refines IncidentManaged}

\text{any}

i

\text{where}

\text{grd\_par: } \text{AllocateRes}\{[i]\} = \text{ResCompTask}\{[i]\}

\text{grd\_self: } i \notin \text{IncidentManaged}

\text{grd\_DuplicateInterrupt: } i \notin \text{IsDuplicate}

\text{grd\_seq: } i \in \text{SetInitialAP}

\text{then}

\text{act: } \text{IncidentManaged} := \text{IncidentManaged} \cup \{i\}

\text{end}

\text{Event: RemoveAlloc (ordinary) } \triangleq \text{refines ResLost}

\text{any}

i

lr

\text{where}

\text{grd\_DuplicateInterrupt: } i \notin \text{IsDuplicate}

\text{grd\_self: } i \mapsto lr \notin \text{RemoveAlloc}

\text{grd\_seq: } i \in \text{SetInitialAP}

\text{grd\_par: } i \notin \text{IncidentManaged}

\text{grd\_xor: } i \mapsto lr \notin \text{RemoveBrokenRes}

\text{grd1: } i \mapsto lr \in \text{AllocateRes}

\text{grd2: } i \mapsto lr \notin \text{DeallocateRes}

\text{then}

\text{act: } \text{RemoveAlloc} := \text{RemoveAlloc} \cup \{i \mapsto lr\}

\text{act1: } \text{AllocateRes} := \text{AllocateRes} \setminus \{i \mapsto lr\}

\text{act2: } \text{ResAttend} := \text{ResAttend} \setminus \{i \mapsto lr\}

\text{act3: } \text{StopMsgReceived} := \text{StopMsgReceived} \setminus \{i \mapsto lr\}

\text{act4: } \text{DeallocateRes} := \text{DeallocateRes} \setminus \{i \mapsto lr\}

\text{end}

\text{Event: RemoveBrokenRes (ordinary) } \triangleq \text{refines ResLost}
any
  i
  lr
where
  grd_DuplicateInterrupt: i \notin IsDuplicate
  grd_self: i \mapsto lr \notin RemoveBrokenRes
  grd_seq: i \in SetInitialAP
  grd_par: i \notin IncidentManaged
  grd_xor: i \mapsto lr \notin RemoveAlloc
  grd1: i \mapsto lr \in AllocateRes
  grd2: i \mapsto lr \notin DeallocateRes
then
  act: RemoveBrokenRes := RemoveBrokenRes \cup \{i \mapsto lr\}
  act1: AllocateRes := AllocateRes \setminus \{i \mapsto lr\}
  act2: ResAttend := ResAttend \setminus \{i \mapsto lr\}
  act3: StopMsgReceived := StopMsgReceived \setminus \{i \mapsto lr\}
  act4: DeallocateRes := DeallocateRes \setminus \{i \mapsto lr\}
end

Event Reset_ResLost (ordinary) ∼
refines Reset_ResLost
any
  i
  lr
where
  grd_seq: i \mapsto lr \in (RemoveBrokenRes \cup RemoveAlloc)
then
  act1: RemoveAlloc := RemoveAlloc \setminus \{i \mapsto lr\}
  act2: RemoveBrokenRes := RemoveBrokenRes \setminus \{i \mapsto lr\}
end
Event ResCompTask (ordinary) ∼
refines ResCompTask
any
  lr
  i
where
  grd_self: i \mapsto lr \notin ResCompTask
  grd_DuplicateInterrupt: i \notin IsDuplicate
  grd_seq: i \mapsto lr \in DeallocateRes
then
  act: ResCompTask := ResCompTask \cup \{i \mapsto lr\}
end
Event CloseIncident \langle \text{ordinary} \rangle \models

refines CloseIncident

\smallskip

any

\begin{align*}
\text{\textit{act}: } CloseIncident & := CloseIncident \cup \{i\}
\end{align*}

\end{align*}

\text{end}

Event IsDuplicate \langle \text{ordinary} \rangle \models

refines IsDuplicate

\smallskip

any

\begin{align*}
\text{\textit{act}: } IsDuplicate & := IsDuplicate \cup \{i\}
\end{align*}

\text{end}

\text{END}

\textbf{B.2.9 Eighth Refinement: M8}

\textbf{MACHINE } M8

\textbf{REFINES } M7

\textbf{SEES } C4

\textbf{VARIABLES}

\begin{itemize}
\item CreateIncident
\item SelectLocation
\item SelectType
\item CompInfo
\item SetInitialAP
\item AllocateRes
\item ResAttend
\end{itemize}
DeallocateRes
StopMsgReceived
IncidentManaged
RequestAdditionalRes
SendStopMsg
CloseIncident
IsDuplicate
ap
notRequired_ap
iLocation
iLocType
iType
DisablePriorityChange
priority
RemoveAlloc
RemoveBrokenRes
alloc allocate physical resource
alloc_LR relate lr to pr
HandleDupAlloc
CloseDuplicate
ResCompTask

**INVARIANTS**

inv_alloc_type: alloc ∈ PR \rightarrow INCIDENT

manually

inv_alloc_seq: ran(allo) ⊆ SetInitialAP

inv_alloc_LR_type: alloc_LR ∈ PR \rightarrow LR

manually

inv_alloc_LR_PR_TYP: \forall pr, lr. pr ∈ dom(allo LR) ∧ alloc_LR(pr) = lr \Rightarrow ptype(pr) = ptype(lr)

inv_alloc_alloc_LR: dom(allo) = dom(allo_LR)

inv_HandleAllocDup_type: HandleDupAlloc ⊆ INCIDENT \times LR

inv_HandleAllocDup_seq: dom(HandleDupAlloc) \subseteq IsDuplicate

inv_HandleAllocDup_rep: \forall i. i ∈ INCIDENT \Rightarrow HandleDupAlloc[{i}] \subseteq AllocateRes[{i}]

inv_CloseDuplicate_seq: \forall i. i ∈ CloseDuplicate \Rightarrow HandleDupAlloc[{i}] = AllocateRes[{i}]

inv_alloc_close: \forall i. i ∈ (CloseIncident \cup CloseDuplicate) \Rightarrow i \notin ran(allo)
Appendix B Fire Dispatch Model with Exception Handling

inv1: \( \text{CloseDuplicate} \subseteq \text{IsDuplicate} \)

Added manually to discharge some proofs

EVENTS

Initialisation

begin

\[
\begin{align*}
\text{act}_{\text{CreateIncident}} &: \text{CreateIncident} := \emptyset \\
\text{act}_{\text{SetInitialAP}} &: \text{SetInitialAP} := \emptyset \\
\text{act}_{\text{CloseIncident}} &: \text{CloseIncident} := \emptyset \\
\text{act}_{\text{IsDuplicate}} &: \text{IsDuplicate} := \emptyset \\
\text{act}_{\text{SendStopMsg}} &: \text{SendStopMsg} := \emptyset \\
\text{act}_{\text{ap}} &: \text{ap} := \emptyset \\
\text{act}_{\text{notRequired_ap}} &: \text{notRequired_ap} := \emptyset \\
\text{act}_{\text{RequestAdditionalRes}} &: \text{RequestAdditionalRes} := \emptyset \\
\text{act}_{2} &: \text{iLocation} := \emptyset \\
\text{act}_{3} &: \text{iType} := \emptyset \\
\text{act}_{\text{loc_type}} &: \text{iLocType} := \emptyset \\
\text{act}_{\text{SelectLocation}} &: \text{SelectLocation} := \emptyset \\
\text{act}_{\text{SelectType}} &: \text{SelectType} := \emptyset \\
\text{act}_{\text{CompInfo}} &: \text{CompInfo} := \emptyset \\
\text{act}_{\text{DisablePriorityChange}} &: \text{DisablePriorityChange} := \emptyset \\
\text{act}_{4} &: \text{priority} := \emptyset \\
\text{act}_{\text{StopMsgReceived}} &: \text{StopMsgReceived} := \emptyset \\
\text{act}_{\text{AllocateRes}} &: \text{AllocateRes} := \emptyset \\
\text{act}_{\text{Attend}} &: \text{ResAttend} := \emptyset \\
\text{act}_{\text{DeallocateRes}} &: \text{DeallocateRes} := \emptyset \\
\text{act}_{\text{IncidentManaged}} &: \text{IncidentManaged} := \emptyset \\
\text{act}_{\text{RemoveAlloc}} &: \text{RemoveAlloc} := \emptyset \\
\text{act}_{\text{RemoveBrokenRes}} &: \text{RemoveBrokenRes} := \emptyset \\
\text{act}_{\text{alloc}} &: \text{alloc} := \emptyset \\
\text{act}_{\text{alloc_LR}} &: \text{alloc_LR} := \emptyset \\
\text{act}_{\text{HandleDupAlloc}} &: \text{HandleDupAlloc} := \emptyset \\
\text{act}_{\text{CloseDuplicate}} &: \text{CloseDuplicate} := \emptyset \\
\text{act}_{\text{ResCompTask}} &: \text{ResCompTask} := \emptyset
\end{align*}
\]

end

Event CreateIncident (ordinary) \( \Downarrow \)

refines CreateIncident

any

i

where

\[
\text{grd_self: } i \notin \text{CreateIncident}
\]

then
act: CreateIncident := CreateIncident ∪ \{i\}

end

Event SelectLocation (ordinary) \equiv

refines SelectLocation

\begin{verbatim}
any
i
l
lt
\end{verbatim}

where
\begin{align*}
grd_{seq}: & \ i \in \text{CreateIncident} \\
grd_{self}: & \ i \notin \text{SelectLocation} \\
grd_{DuplicateInterrupt}: & \ i \notin \text{IsDuplicate} \\
grd_{1}: & \ l \in \text{LOCATION} \\
grd_{2}: & \ lt \in \text{LOCATION_TYPE}
\end{align*}

then
\begin{align*}
act: & \ SelectLocation := SelectLocation \cup \{i\} \\
act_{1}: & \ iLocation(i) := l \\
act_{2}: & \ iLocType(i) := lt
\end{align*}

end

Event SelectType (ordinary) \equiv

refines SelectType

\begin{verbatim}
any
i
t
p
\end{verbatim}

where
\begin{align*}
grd_{seq}: & \ i \in \text{CreateIncident} \\
grd_{self}: & \ i \notin \text{SelectType} \\
grd_{1}: & \ t \in \text{INCIDENT_TYPE} \\
grd_{DuplicateInterrupt}: & \ i \notin \text{IsDuplicate} \\
grd_{2}: & \ p = \text{default_priority}(t)
\end{align*}

then
\begin{align*}
act: & \ SelectType := SelectType \cup \{i\} \\
act_{1}: & \ iTType(i) := t \\
act_{2}: & \ priority(i) := p
\end{align*}

end

Event ChangePriority (ordinary) \equiv

refines ChangePriority

\begin{verbatim}
any
i
\end{verbatim}
where

grd\_seq.priority: i ∈ SelectType

grd0\_loop: i /∈ DisablePriorityChange

grd1: p ∈ 1..3

grd2: p ≠ priority(i)

grd3: i /∈ CloseIncident

we can also set DisablePriorityChange in CloseIncident

grd\_DuplicateInterrupt: i /∈ IsDuplicate
	hen

act1: priority(i) := p

end

Event DisablePriorityChange \langle\text{ordinary}\rangle ≡
refines DisablePriorityChange

any

i

where

grd\_DuplicateInterrupt: i /∈ IsDuplicate

grd\_seq.priority: i ∈ SelectType

grd\_self: i /∈ DisablePriorityChange

grd1: i /∈ CloseIncident

we can also set DisablePriorityChange in CloseIncident

then

act: DisablePriorityChange := DisablePriorityChange ∪ \{i\}

end

Event CompInfo \langle\text{ordinary}\rangle ≡
refines CompInfo

any

i

where

grd\_DuplicateInterrupt: i /∈ IsDuplicate

grd\_seq.: (i ∈ SelectLocation ∧ i ∈ SelectType)

grd\_self: i /∈ CompInfo

then

act: CompInfo := CompInfo ∪ \{i\}

end

Event SetInitialAP \langle\text{ordinary}\rangle ≡
refines SetInitialAP

any

lrs
Appendix B Fire Dispatch Model with Exception Handling

Where

\begin{align*}
\text{grd\_seq: } & i \in \text{CompInfo} \\
\text{grd\_self: } & i \notin \text{SetInitialAP} \\
\text{grd1: } & t = \text{ getType}(i) \\
\text{grd2: } & l = \text{ locType}(i) \\
\text{grd3: } & \text{lr} = \text{PDA}(t \mapsto l) \\
\text{DuplicateInterrupt: } & i \notin \text{IsDuplicate} \\
\end{align*}

\text{then}

\begin{align*}
\text{act: } & \text{SetInitialAP} := \text{SetInitialAP} \cup \{i\} \\
\text{act1: } & \text{ap} := \text{ap} \cup \{i\} \times \text{lr} \\
\end{align*}

\text{end}

\textbf{Event} RequestAdditionalRes \langle \text{ordinary} \rangle \cong \textbf{refines} RequestAdditionalRes

\text{any}

\begin{align*}
i \\
\text{lr} \\
\end{align*}

\text{where}

\begin{align*}
\text{grd\_seq\_planF: } & i \in \text{SetInitialAP} \\
\text{grd\_self: } & i \mapsto \text{lr} \notin \text{RequestAdditionalRes} \\
\text{grd\_InputExpression1: } & \text{lr} \in \text{LR} \\
\text{grd\_par: } & i \notin \text{SendStopMsg} \\
\text{DuplicateInterrupt: } & i \notin \text{IsDuplicate} \\
\text{grd1: } & \text{lr} \notin \text{ap}\{i\} \\
\end{align*}

\text{then}

\begin{align*}
\text{act1: } & \text{ap} := \text{ap} \cup \{i\} \mapsto \text{lr} \\
\text{act: } & \text{RequestAdditionalRes} := \text{RequestAdditionalRes} \cup \{i \mapsto \text{lr}\} \\
\end{align*}

\text{end}

\textbf{Event} SendStopMsg \langle \text{ordinary} \rangle \cong \textbf{refines} SendStopMsg

\text{any}

\begin{align*}
i \\
\text{lr} \\
\end{align*}

\text{where}

\begin{align*}
\text{grd\_lr: } & \text{lr} = \text{ap}\{i\} \setminus \text{ResAttend}\{i\} \\
\text{DuplicateInterrupt: } & i \notin \text{IsDuplicate} \\
\text{grd\_seq: } & i \in \text{SetInitialAP} \\
\text{grd\_self: } & i \notin \text{SendStopMsg} \\
\end{align*}

\text{then}
\textbf{Appendix B Fire Dispatch Model with Exception Handling}

\begin{align*}
\text{act1: } & \text{notRequired}_{ap} := \text{notRequired}_{ap} \cup \{i\} \times \text{lr} \\
\text{act: } & \text{SendStopMsg} := \text{SendStopMsg} \cup \{i\}
\end{align*}

\textbf{end}

\textbf{Event} AllocateRes (ordinary) $\triangleq$

\textbf{refines} AllocateRes

\textbf{any}

\begin{align*}
i \\
lr \\
pr
\end{align*}

\textbf{where}

\begin{align*}
grd\_seq\_allocation: & \quad i \in \text{SetInitialAP} \\
grd\_self: & \quad i \mapsto lr \notin \text{AllocateRes} \\
grd\_InputExpression1: & \quad lr \in \text{ap}[\{i\}] \\
grd0\_par: & \quad i \notin \text{IncidentManaged} \\
grd\_DuplicateInterrupt: & \quad i \notin \text{IsDuplicate} \\
grd1: & \quad i \mapsto lr \notin \text{notRequired}_{ap} \\
grd\_ResLostRetry: & \quad i \mapsto lr \notin (\text{RemoveAlloc} \cup \text{RemoveBrokenRes}) \\
grd2: & \quad p\text{type}(pr) = l\text{type}(lr) \\
grd3: & \quad pr \in \text{dom}(\text{alloc}) \Rightarrow \text{priority}(i) < \text{priority}(\text{alloc}(pr))
\end{align*}

\textbf{then}

\begin{align*}
\text{act: } & \text{AllocateRes} := \text{AllocateRes} \cup \{i \mapsto lr\} \\
\text{act1: } & \text{alloc}(pr) := i \\
\text{act2: } & \text{alloc\_LR}(pr) := lr \\
\text{end}
\end{align*}

\textbf{Event} UpdateResLocation (ordinary) $\triangleq$

\textbf{refines} UpdateResLocation

\textbf{any}

\begin{align*}
i \\
lr \\
pr
\end{align*}

\textbf{where}

\begin{align*}
grd\_seq\_allocation: & \quad i \mapsto lr \in \text{AllocateRes} \\
grd0\_loop: & \quad i \mapsto lr \notin (\text{ResAttend} \cup \text{StopMsgReceived}) \\
& \quad @\text{grd\_ResLostRetry} i \mapsto lr \notin \text{ResLost} \\
grd\_DuplicateInterrupt: & \quad i \notin \text{IsDuplicate} \\
grd1: & \quad i \mapsto lr \notin \text{notRequired}_{ap} \\
grd\_ResLostRetry: & \quad i \mapsto lr \notin (\text{RemoveAlloc} \cup \text{RemoveBrokenRes}) \\
grd2: & \quad pr \in (\text{alloc}^{-1}[\{i\}] \cap \text{alloc\_LR}^{-1}[\{lr\}])
\end{align*}

\textbf{then}

\begin{align*}
\text{skip}
\end{align*}

\textbf{end}
**Event** \textnormal{ReallocateQuickerRes} \ (\textit{ordinary}) \ \overset{\sim}{=} \\
\textbf{refines} \ \textnormal{ReallocateQuickerRes}\\
\quad \textbf{any} \\
\quad i \\
\quad lr \\
\quad pr \\
\quad pr\_quicker \\
\textbf{where} \\
\quad \texttt{grd}\_\texttt{seq} \ \texttt{allocation}: \ i \mapsto lr \in \textit{AllocateRes} \\
\quad \texttt{grd}\_\texttt{0}\_\texttt{loop}: \ i \mapsto lr \notin (\textit{ResAttend} \cup \textit{StopMsgReceived}) \\
\quad \quad \quad \@\texttt{grd}\_\texttt{ResLost}\_\texttt{Retry} \ i \mapsto lr \notin \textit{ResLost} \\
\quad \texttt{grd}\_\texttt{Duplicate}\_\texttt{Interrupt}: \ i \notin \textit{IsDuplicate} \\
\quad \texttt{grd1}: \ i \mapsto lr \notin \textit{notRequired}\_\texttt{ap} \\
\quad \texttt{grd}\_\texttt{ResLost}\_\texttt{Retry}: \ i \mapsto lr \notin (\texttt{RemoveAlloc} \cup \texttt{RemoveBrokenRes}) \\
\quad \texttt{grd2}: \ pr \in (\texttt{alloc}^{-1}[\{i\}] \cap \texttt{alloc}\_\texttt{LR}^{-1}[\{lr\}]) \\
\quad \texttt{grd3}: \ pr\_\texttt{quicker} \notin \texttt{dom(alloc)} \\
\quad \texttt{grd4}: \ \texttt{ptype}(pr\_\texttt{quicker}) = \texttt{ltype}(lr) \\
\quad \quad \quad \@\texttt{grd5} \ \texttt{ptype(pr)} = \texttt{ltype(lr)} \\
\textbf{then} \\
\quad \texttt{act1}: \ \texttt{alloc} := (\{pr\} \triangleq \texttt{alloc}) \cup \{pr\_\texttt{quicker} \mapsto i\} \\
\quad \texttt{act2}: \ \texttt{alloc}\_\texttt{LR} := (\{pr\} \triangleq \texttt{alloc}\_\texttt{LR}) \cup \{pr\_\texttt{quicker} \mapsto lr\} \\
\textbf{end} \\

**Event** \textnormal{StopMsgReceived} \ (\textit{ordinary}) \ \overset{\sim}{=} \\
\textbf{refines} \ \textnormal{StopMsgReceived}\\
\quad \textbf{any} \\
\quad lr \\
\quad i \\
\quad pr \\
\textbf{where} \\
\quad \texttt{grd}\_\texttt{seq}\_\texttt{allocation}: \ i \mapsto lr \in \textit{AllocateRes} \\
\quad \texttt{grd}\_\texttt{self}: \ i \mapsto lr \notin \textit{StopMsgReceived} \\
\quad \quad \quad \@\texttt{grd}\_\texttt{ResLost}\_\texttt{Retry} \ i \mapsto lr \notin \textit{ResLost} \\
\quad \texttt{grd}\_\texttt{Duplicate}\_\texttt{Interrupt}: \ i \notin \textit{IsDuplicate} \\
\quad \texttt{grd}\_\texttt{xor}: \ i \mapsto lr \notin \textit{ResAttend} \\
\quad \texttt{grd1}: \ i \mapsto lr \in \textit{notRequired}\_\texttt{ap} \\
\quad \texttt{grd}\_\texttt{ResLost}\_\texttt{Retry}: \ i \mapsto lr \notin (\texttt{RemoveAlloc} \cup \texttt{RemoveBrokenRes}) \\
\quad \texttt{grd}\_\texttt{seq}_\texttt{seq}: \ i \in \textit{SetInitialAP} \\
\quad \texttt{grd}\_\texttt{Input}\_\texttt{Expression1}: \ lr \in ap[\{i\}] \\
\quad \texttt{grd2}: \ pr \in (\texttt{alloc}^{-1}[\{i\}] \cap \texttt{alloc}\_\texttt{LR}^{-1}[\{lr\}]) \\
\textbf{then} \\
\quad \texttt{act}: \ \texttt{StopMsgReceived} := \texttt{StopMsgReceived} \cup \{i \mapsto lr\}
end

Event ResAttend (ordinary) \triangleq
refines ResAttend

any
  i
  lr
  pr

where

\text{grd\_InputExpression1}: lr \in ap\{i\}\n\text{grd\_seq\_allocation}: i \mapsto lr \in \text{AllocateRes}\n\text{grd\_self}: i \mapsto lr \notin \text{ResAttend}\n  @\text{grdResLostRetry} i \mapsto lr \notin \text{ResLost}\n\text{grd\_DuplicateInterrupt}: i \notin \text{IsDuplicate}\n\text{grd\_xor}: i \mapsto lr \notin \text{StopMsgReceived}\n\text{grd\_seq}: i \in \text{SetInitialAP}\n\text{grd1}: i \mapsto lr \notin \text{notRequired}\_ap\n\text{grdResLostRetry}: i \mapsto lr \notin (\text{RemoveAlloc} \cup \text{RemoveBrokenRes})\n\text{grd2}: pr \in (\text{alloc}^{-1}\{i\} \cap \text{alloc\_LR}^{-1}\{lr\})

\text{then}\n
  \text{act}: \text{ResAttend} := \text{ResAttend} \cup \{i \mapsto lr\}\n
end

Event DeallocateRes (ordinary) \triangleq
refines DeallocateRes

any
  lr
  i
  pr

where

\text{grd\_seq}: i \mapsto lr \in (\text{ResAttend} \cup \text{StopMsgReceived})\n\text{grd\_self}: i \mapsto lr \notin \text{DeallocateRes}\n  @\text{grdResLostRetry} i \mapsto lr \notin \text{ResLost}\n\text{grd\_DuplicateInterrupt}: i \notin \text{IsDuplicate}\n  @\text{grd1} lr \in \text{ap\{i\}} // \text{GRD PO}//@\text{grdResLostRetry} i \mapsto lr \notin \text{ResLost}\n\text{grdResLostRetry}: i \mapsto lr \notin (\text{RemoveAlloc} \cup \text{RemoveBrokenRes})\n\text{grd1}: pr \in (\text{alloc}^{-1}\{i\} \cap \text{alloc\_LR}^{-1}\{lr\}) \quad \text{manually}\n
\text{then}\n
  \text{act}: \text{DeallocateRes} := \text{DeallocateRes} \cup \{i \mapsto lr\}\n  \text{act1}: \text{alloc} := \{pr\} \leftarrow \text{alloc}\n    \text{manually}
\textbf{act2:} \(\text{alloc} \_\text{LR} := \{pr\} \leq \text{alloc} \_\text{LR}\)  

\begin{verbatim}
manually
end
\end{verbatim}

\textbf{Event} IncidentManaged (\textit{ordinary}) \(\hat{=}\) \textbf{refines} IncidentManaged
\begin{verbatim}
any
\textbf{where}
\begin{itemize}
  \item \textbf{grd} \_\textbf{self:} \(i \notin \text{IncidentManaged}\)
  \item \textbf{grd} \_\textbf{DuplicateInterrupt:} \(i \notin \text{IsDuplicate}\)
  \item \textbf{grd} \_\textbf{seq:} \(i \in \text{SetInitialAP}\)
  \item \textbf{grd} \_\textbf{par:} \(\text{AllocateRes}[\{i\}] = \text{ResCompTask}[\{i\}]\)
\end{itemize}
\begin{verbatim}
then
\textbf{act:} \(\text{IncidentManaged} := \text{IncidentManaged} \cup \{i\}\)
\end{verbatim}
\end{verbatim}

\textbf{Event} RemoveAlloc (\textit{ordinary}) \(\hat{=}\) \textbf{refines} RemoveAlloc
\begin{verbatim}
any
\textbf{where}
\begin{itemize}
  \item \textbf{grd} \_\textbf{2:} \(i \mapsto lr \notin \text{alloc}^{-1}; \text{alloc} \_\text{LR}\)
  \item \(\text{alloc} \_\text{LR}^{-1}[\{i\}] \cap \text{alloc} \_\text{LR}^{-1}[\{lr\}] = \emptyset\)
  \item \textbf{grd} \_\textbf{self:} \(i \mapsto lr \notin \text{RemoveAlloc}\)
  \item \textbf{grd} \_\textbf{seq:} \(i \in \text{SetInitialAP}\)
  \item \textbf{grd} \_\textbf{par:} \(i \notin \text{IncidentManaged}\)
  \item \textbf{grd} \_\textbf{xor:} \(i \mapsto lr \notin \text{RemoveBrokenRes}\)
  \item \textbf{grd} \_\textbf{DuplicateInterrupt:} \(i \notin \text{IsDuplicate}\)
  \item \textbf{grd} \_\textbf{Retry:} \(i \mapsto lr \notin \text{DeallocateRes}\)
  \item \textbf{grd} \_\textbf{1:} \(i \mapsto lr \in \text{AllocateRes}\)
\end{itemize}
\begin{verbatim}
then
\textbf{act:} \(\text{RemoveAlloc} := \text{RemoveAlloc} \cup \{i \mapsto lr\}\)
\textbf{act1:} \(\text{AllocateRes} := \text{AllocateRes} \setminus \{i \mapsto lr\}\)
\textbf{act2:} \(\text{ResAttend} := \text{ResAttend} \setminus \{i \mapsto lr\}\)
\textbf{act3:} \(\text{StopMsgReceived} := \text{StopMsgReceived} \setminus \{i \mapsto lr\}\)
\textbf{act4:} \(\text{DeallocateRes} := \text{DeallocateRes} \setminus \{i \mapsto lr\}\)
\end{verbatim}
\end{verbatim}

\textbf{Event} RemoveBrokenRes (\textit{ordinary}) \(\hat{=}\) \textbf{refines} RemoveBrokenRes
\begin{verbatim}
any
\end{verbatim}
\begin{align*}
i, \quad lr, \quad pr
\end{align*}

\textbf{where}

\begin{align*}
\text{grd\_DuplicateInterrupt: } i &\notin IsDuplicate \\
\text{grd\_self: } i &\mapsto lr \notin RemoveBrokenRes \\
\text{grd\_seq: } i &\in SetInitialAP \\
\text{grd\_par: } i &\notin IncidentManaged \\
\text{grd\_xor: } i &\mapsto lr \notin RemoveAlloc \\
\text{grd1: } i &\mapsto lr \in AllocateRes \\
\text{grd2: } i &\mapsto lr \notin DeallocateRes \\
\text{grd3: } pr &\in \{ alloc^{-1}(i) \cap alloc\_LR^{-1}(lr) \}
\end{align*}

\textbf{then}

\begin{align*}
\text{act: } Remove\_Broken\_Res &:= Remove\_Broken\_Res \cup \{ i \mapsto lr \} \\
\text{act1: } Allocate\_Res &:= Allocate\_Res \setminus \{ i \mapsto lr \} \\
\text{act2: } Res\_Attend &:= Res\_Attend \setminus \{ i \mapsto lr \} \\
\text{act3: } Stop\_Msg\_Received &:= Stop\_Msg\_Received \setminus \{ i \mapsto lr \} \\
\text{act4: } Deallocate\_Res &:= Deallocate\_Res \setminus \{ i \mapsto lr \} \\
\text{act5: } alloc &:= \{ pr \} \bowtie alloc \\
\hspace{1cm} \text{manually} \\
\text{act6: } alloc\_LR &:= \{ pr \} \bowtie alloc\_LR \\
\hspace{1cm} \text{manually}
\end{align*}

\textbf{end}

\textbf{Event} Reset\_ResLost \langle \text{ordinary} \rangle \triangleq

\textbf{refines} Reset\_ResLost

\begin{align*}
\text{any}
\end{align*}

\textbf{where}

\begin{align*}
\text{grd\_seq: } i &\mapsto lr \in (Remove\_Broken\_Res \cup Remove\_Alloc)
\end{align*}

\textbf{then}

\begin{align*}
\text{act1: } Remove\_Alloc &:= Remove\_Alloc \setminus \{ i \mapsto lr \} \\
\text{act2: } Remove\_Broken\_Res &:= Remove\_Broken\_Res \setminus \{ i \mapsto lr \}
\end{align*}

\textbf{end}

\textbf{Event} CloseIncident \langle \text{ordinary} \rangle \triangleq

\textbf{refines} CloseIncident

\begin{align*}
\text{any}
\end{align*}

\textbf{where}

\begin{align*}
\text{grd2: } i &\notin \text{ran}(alloc) \\
\text{grd\_seq\_FD: } (i \in Send\_Stop\_Msg \land i \in Incident\_Managed)
\end{align*}
\textbf{Appendix B Fire Dispatch Model with Exception Handling}

\texttt{grd\_self: i \notin \text{CloseIncident}}
\texttt{grd\_DuplicateInterrupt: i \notin \text{IsDuplicate}}
\texttt{grd1: ap[i] \setminus \text{notRequired\_ap[i]} \subseteq \text{DeallocateRes[i]}}

\texttt{then}
\texttt{act: CloseIncident := CloseIncident \cup \{i\}}
\texttt{end}

\textbf{Event} \texttt{IsDuplicate (ordinary) \models}
\texttt{refines IsDuplicate}

\texttt{any}
\texttt{i}
\texttt{where}
\texttt{grd\_seq\_DuplicateInterrupt: i \in \text{CreateIncident}}
\texttt{grd\_self: i \notin \text{IsDuplicate}}
\texttt{grd1: i \notin \text{CloseIncident}}

\texttt{then}
\texttt{act: IsDuplicate := IsDuplicate \cup \{i\}}
\texttt{end}

\textbf{Event} \texttt{HandleDupAlloc (ordinary) \models}

\texttt{any}
\texttt{i}
\texttt{lr}
\texttt{pr}
\texttt{where}
\texttt{grd\_seq: i \in \text{IsDuplicate}}
\texttt{grd\_self: i \mapsto lr \notin \text{HandleDupAlloc}}
\texttt{grd\_exp: lr \in \text{AllocateRes[i]}}
\texttt{grd1: pr \in (\text{alloc}^{-1}[\{i\}] \cap \text{alloc\_LR}^{-1}[\{lr\}])}

\texttt{then}
\texttt{act: HandleDupAlloc := HandleDupAlloc \cup \{i \mapsto lr\}}
\texttt{act1: alloc := \{pr\} \triangleleft alloc}
\texttt{act26: alloc\_LR := \{pr\} \triangleleft alloc\_LR}
\texttt{end}

\textbf{Event} \texttt{CloseDuplicate (ordinary) \models}

\texttt{any}
\texttt{i}
\texttt{where}
\texttt{grd\_seq: \text{HandleDupAlloc[i]} = \text{AllocateRes[i]}}
\texttt{grd\_self: i \notin \text{CloseDuplicate}}
\texttt{grd1: i \in \text{IsDuplicate}}
\texttt{grd2: i \notin \text{ran(alloc)}}
then

\textbf{act}: \quad \text{CloseDuplicate} := \text{CloseDuplicate} \cup \{i\}

end

\textbf{Event} \quad \text{ResCompTask (ordinary) } \rightleftharpoons

\textbf{refines} \quad \text{ResCompTask}

\quad any

\quad lr

\quad i

\textbf{where}

\quad \text{grd\_DuplicateInterrupt}: \quad i \notin \text{IsDuplicate}

\quad \text{grd\_seq}: \quad i \mapsto lr \in \text{DeallocateRes}

\quad \text{grd\_self}: \quad i \mapsto lr \notin \text{ResCompTask}

then

\textbf{act}: \quad \text{ResCompTask} := \text{ResCompTask} \cup \{i \mapsto lr\}

end

END

B.2.10 Ninth Refinement: M9

\textbf{MACHINE} \quad M9

\textbf{REFINES} \quad M8

\textbf{SEES} \quad \text{M9, implicitContext, C4}

\textbf{VARIABLES}

\begin{itemize}
  \item pr\_status
  \item pr\_available\_sm
  \item pr\_allocated\_sm
  \item pr\_free\_sm
  \item CreateIncident
  \item SelectLocation
  \item SelectType
  \item CompInfo
  \item SetInitialAP
  \item AllocateRes
  \item ResAttend
  \item DeallocateRes
  \item StopMsgReceived
  \item IncidentManaged
  \item RequestAdditionalRes
  \item SendStopMsg
\end{itemize}
Appendix B Fire Dispatch Model with Exception Handling

CloseIncident
IsDuplicate
ap
notRequired_ap
iLocation
iType
DisablePriorityChange
priority
RemoveAlloc
RemoveBrokenRes
alloc allocate physical resource
alloc_LR relate lr to pr
HandleDupAlloc
CloseDuplicate
ResCompTask
iLocType

INvariants

typeof_pr_status: pr_status ∈ PR → pr_status_STATES

typeof_pr_available_sm: pr_available_sm ∈ PR → pr_available_sm_STATES

typeof_pr_allocated_sm: pr_allocated_sm ∈ PR → pr_allocated_sm_STATES

typeof_pr_free_sm: pr_free_sm ∈ PR → pr_free_sm_STATES

superstateof_pr_available_sm: dom(pr_available_sm▷{pr_available_sm_NULL}) = dom(pr_status▷{Available})

superstateof_pr_allocated_sm: dom(pr_allocated_sm▷{pr_allocated_sm_NULL}) = dom(pr_available_sm▷{pr_allocated})

superstateof_pr_free_sm: dom(pr_free_sm▷{pr_free_sm_NULL}) = dom(pr_available_sm▷{pr_free})

inv1: ∀pr·pr_available_sm(pr) = pr_allocated ⇔ pr ∈ dom(alloc)

inv2: ∀pr·pr_available_sm(pr) ∈ {pr_free, pr_available_sm_NULL} ⇔ pr /∈ dom(alloc)

Events

Initialisation  (extended)

begin

act_CreateIncident: CreateIncident := ∅

act_SetInitialAP: SetInitialAP := ∅

act_CloseIncident: CloseIncident := ∅

act_IsDuplicate: IsDuplicate := ∅

act_SendStopMsg: SendStopMsg := ∅
act_{ap} : \text{ap} := \emptyset

act_{notRequired\_ap} : notRequired\_ap := \emptyset

act_{RequestAdditionalRes} : RequestAdditionalRes := \emptyset

act2 : iLocation := \emptyset

act3 : iType := \emptyset

act\_loc\_type : iLocType := \emptyset

act_{SelectLocation} : SelectLocation := \emptyset

act_{SelectType} : SelectType := \emptyset

act_{CompInfo} : CompInfo := \emptyset

act_{DisablePriorityChange} : DisablePriorityChange := \emptyset

act4 : priority := \emptyset

act_{StopMsgReceived} : StopMsgReceived := \emptyset

act_{AllocateRes} : AllocateRes := \emptyset

act_{Attend} : ResAttend := \emptyset

act_{Deallocation} : DeallocateRes := \emptyset

act_{IncidentManaged} : IncidentManaged := \emptyset

act_{RemoveAlloc} : RemoveAlloc := \emptyset

act_{RemoveBrokenRes} : RemoveBrokenRes := \emptyset

act\_alloc : alloc := \emptyset

act\_{alloc\_LR} : alloc\_LR := \emptyset

act_{HandleDupAlloc} : HandleDupAlloc := \emptyset

act_{CloseDuplicate} : CloseDuplicate := \emptyset

act_{ResCompTask} : ResCompTask := \emptyset

init_{pr\_free\_sm} : pr\_free\_sm := PR \times \{\text{StationAvailable}\}

init_{pr\_available\_sm} : pr\_available\_sm := PR \times \{pr\_free\}

init_{pr\_status} : pr\_status := PR \times \{\text{Available}\}

init_{pr\_allocated\_sm} : pr\_allocated\_sm := PR \times \{pr\_allocated\_sm\_NULL\}

\textbf{Event} CreateIncident \langle\text{ordinary}\rangle \triangleq
\textbf{extends} CreateIncident
\begin{align*}
\text{any} \\
\text{where} \\
\text{grd\_self}: \ i \not\in \text{CreateIncident} \\
\text{then} \\
\text{act}: \ CreateIncident := CreateIncident \cup \{i\}
\end{align*}
\textbf{end}

\textbf{Event} SelectLocation \langle\text{ordinary}\rangle \triangleq
\textbf{extends} SelectLocation
\begin{align*}
\text{any} \\
\text{where} \\
\end{align*}
Appendix B Fire Dispatch Model with Exception Handling

\[ l \]
\[ lt \]

**where**

- \( grd\_seq: \ i \in CreateIncident \)
- \( grd\_self: \ i \notin SelectLocation \)
- \( grd\_DuplicateInterrupt: \ i \notin IsDuplicate \)
- \( grd1: \ l \in LOCATION \)
- \( grd2: \ lt \in LOCATION\_TYPE \)

**then**

- \( act: \ SelectLocation := SelectLocation \cup \{i\} \)
- \( act1: \ iLocation(i) := l \)
- \( act2: \ iLocType(i) := lt \)

**end**

**Event** SelectType \((\text{ordinary}) \equiv \)**

**extends** SelectType

**any**

- \( i \)
- \( t \)
- \( p \)

**where**

- \( grd\_seq: \ i \in CreateIncident \)
- \( grd\_self: \ i \notin SelectType \)
- \( grd1: \ t \in INCIDENT\_TYPE \)
- \( grd\_DuplicateInterrupt: \ i \notin IsDuplicate \)
- \( grd2: \ p = default\_priority(t) \)

**then**

- \( act: \ SelectType := SelectType \cup \{i\} \)
- \( act1: \ iType(i) := t \)
- \( act2: \ priority(i) := p \)

**end**

**Event** ChangePriority \((\text{ordinary}) \equiv \)**

**extends** ChangePriority

**any**

- \( i \)
- \( p \)

**where**

- \( grd\_seq\_priority: \ i \in SelectType \)
- \( grd0\_loop: \ i \notin DisablePriorityChange \)
- \( grd1: \ p \in 1..3 \)
- \( grd2: \ p \neq priority(i) \)
Appendix B Fire Dispatch Model with Exception Handling

\[ \text{grd}3: \ i \notin \text{CloseIncident} \]

we can also set \text{DisablePriorityChange} in \text{CloseIncident}

\[ \text{grd}_{\text{DuplicateInterrupt}}: \ i \notin \text{IsDuplicate} \]

\[ \text{act1}: \ priority(i) := p \]

\textbf{Event} \ DisablePriorityChange \langle \text{ordinary} \rangle \triangleq \textbf{extends} \ DisablePriorityChange \]

\[ \text{any} \]

\[ \text{where} \]

\[ \text{grd}_{\text{DuplicateInterrupt}}: \ i \notin \text{IsDuplicate} \]

\[ \text{grd}_{\text{seq}.priority}: \ i \in \text{SelectType} \]

\[ \text{grd}_{\text{self}}: \ i \notin \text{DisablePriorityChange} \]

\[ \text{grd1}: \ i \notin \text{CloseIncident} \]

we can also set \text{DisablePriorityChange} in \text{CloseIncident}

\[ \text{then} \]

\[ \text{act}: \ DisablePriorityChange := DisablePriorityChange \cup \{i\} \]

\textbf{end}

\textbf{Event} \ CompInfo \langle \text{ordinary} \rangle \triangleq \textbf{extends} \ CompInfo \]

\[ \text{any} \]

\[ \text{where} \]

\[ \text{grd}_{\text{DuplicateInterrupt}}: \ i \notin \text{IsDuplicate} \]

\[ \text{grd}_{\text{seq}.}: \ (i \in \text{SelectLocation} \land i \in \text{SelectType}) \]

\[ \text{grd}_{\text{self}}: \ i \notin \text{CompInfo} \]

\[ \text{then} \]

\[ \text{act}: \ CompInfo := CompInfo \cup \{i\} \]

\textbf{end}

\textbf{Event} \ SetInitialAP \langle \text{ordinary} \rangle \triangleq \textbf{extends} \ SetInitialAP \]

\[ \text{any} \]

\[ \text{where} \]

\[ \text{grd}_{\text{seq}.}: \ i \in \text{CompInfo} \]

\[ \text{grd}_{\text{self}}: \ i \notin \text{SetInitialAP} \]

\[ lrs \]

\[ i \]

\[ t \]

\[ l \]
Appendix B Fire Dispatch Model with Exception Handling

grd1: \( t = iType(i) \)
grd2: \( l = iLocType(i) \)
grd3: \( lrs = PDA(t \mapsto l) \)
grd\_DuplicateInterrupt: \( i \notin IsDuplicate \)

then
act: \( SetInitialAP := SetInitialAP \cup \{i\} \)
act1: \( ap := ap \cup (\{i\} \times lrs) \)
end

Event RequestAdditionalRes (ordinary) \( \triangleq \)
extends RequestAdditionalRes

any
\( i \)
\( lr \)
where
grd\_seq\_planF: \( i \in SetInitialAP \)
grd\_self: \( i \mapsto lr \notin RequestAdditionalRes \)
grd\_InputExpression1: \( lr \in LR \)
grd0\_par: \( i \notin SendStopMsg \)
grd\_DuplicateInterrupt: \( i \notin IsDuplicate \)
grd1: \( lr \notin ap[i] \)
then
act1: \( ap := ap \cup \{i \mapsto lr\} \)
act: \( RequestAdditionalRes := RequestAdditionalRes \cup \{i \mapsto lr\} \)
end

Event SendStopMsg (ordinary) \( \triangleq \)
extends SendStopMsg

any
\( i \)
\( lrs \)
where
grd\_lrs: \( lrs = ap[i] \setminus ResAttend[i] \)
grd\_DuplicateInterrupt: \( i \notin IsDuplicate \)
grd\_seq: \( i \in SetInitialAP \)
grd\_self: \( i \notin SendStopMsg \)
then
act1: \( notRequired\_ap := notRequired\_ap \cup (\{i\} \times lrs) \)
act: \( SendStopMsg := SendStopMsg \cup \{i\} \)
end

Event AllocateRes (ordinary) \( \triangleq \)
extends AllocateRes
any
  i
  lr
  pr

where

grd_seq.allocation: \( i \in \text{SetInitialAP} \)
grd_self: \( i \mapsto lr \not\in \text{AllocateRes} \)
grd_InputExpression1: \( lr \in \text{ap}[i] \)
grd0_par: \( i \notin \text{IncidentManaged} \)
grd_DuplicateInterrupt: \( i \notin \text{IsDuplicate} \)
grd1: \( i \mapsto lr \notin \text{notRequired\_ap} \)
grdResLostRetry: \( i \mapsto lr \notin (\text{RemoveAlloc} \cup \text{RemoveBrokenRes}) \)
grd2: \( \text{ptype}(pr) = \text{ltype}(lr) \)
grd3: \( pr \in \text{dom}(\text{alloc}) \Rightarrow \text{priority}(i) < \text{priority}(\text{alloc}(pr)) \)
isin\_Mobilised\_or\_isin\_pr\_free: \( (pr\_\text{allocated\_sm}(pr) = \text{Mobilised} \lor pr\_\text{available\_sm}(pr) = pr\_\text{free}) \)

then

act: \( \text{AllocateRes} := \text{AllocateRes} \cup \{i \mapsto lr\} \)
act1: \( \text{alloc}(pr) := i \)
act2: \( \text{alloc\_LR}(pr) := lr \)
leave_pr\_free\_sm: \( pr\_\text{free\_sm}(pr) := pr\_\text{free\_sm\_NULL} \)
enter_Mobilised: \( pr\_\text{allocated\_sm}(pr) := \text{Mobilised} \)
enter\_pr\_allocated: \( pr\_\text{available\_sm}(pr) := pr\_\text{allocated} \)

end

Event UpdateResLocation \langle \text{ordinary} \rangle 

extends UpdateResLocation

any
  i
  lr
  pr

where

grd_seq.allocation: \( i \mapsto lr \in \text{AllocateRes} \)
grd0_loop: \( i \mapsto lr \notin (\text{ResAttend} \cup \text{StopMsgReceived}) \)
  \( \ominus \text{grdResLostRetry} i \mapsto lr \notin \text{ResLost} \)
grd_DuplicateInterrupt: \( i \notin \text{IsDuplicate} \)
grd1: \( i \mapsto lr \notin \text{notRequired\_ap} \)
grdResLostRetry: \( i \mapsto lr \notin (\text{RemoveAlloc} \cup \text{RemoveBrokenRes}) \)
grd2: \( pr \in (\text{alloc}^{-1}[i] \cap \text{alloc\_LR}^{-1}[lr]) \)
isin\_Mobilised: \( pr\_\text{allocated\_sm}(pr) = \text{Mobilised} \)

then

skip
end

**Event** ReallocateQuickerRes \(\langle\text{ordinary}\rangle\) \(\triangleq\) extends ReallocateQuickerRes

any

\(i\)

\(lr\)

\(pr\)

\(pr\_quicker\)

where

\(grd\_seq\_allocation\): \(i \mapsto lr \in \text{AllocateRes}\)

\(grd0\_loop\): \(i \mapsto lr \notin (\text{ResAttend} \cup \text{StopMsgReceived})\)

@grdResLostRetry \(i \mapsto lr \notin \text{ResLost}\)

\(grd\_Duplicate\_Interrupt\): \(i \notin \text{IsDuplicate}\)

\(grd1\): \(i \mapsto lr \notin \text{notRequired\_ap}\)

\(grd\_Res\_Lost\_Retry\): \(i \mapsto lr \notin (\text{RemoveAlloc} \cup \text{RemoveBrokenRes})\)

\(grd2\): \(pr \in (\text{alloc}^{-1}[\{i\}] \cap \text{alloc\_LR}^{-1}[\{lr\}])\)

\(grd3\): \(pr\_quicker \notin \text{dom}(\text{alloc})\)

\(grd4\): \(\text{ptype}(pr\_quicker) = \text{ltype}(lr)\)

@grd5 \(\text{ptype}(pr) = \text{ltype}(lr)\)

\(grd5\): \(\text{pr\_available\_sm}(pr\_quicker) = \text{pr\_free}\)

manually

\(grd6\): \(\text{pr\_allocated\_sm}(pr) = \text{Mobilised}\)

manually

then

\(act1\): \(\text{alloc} := (\{pr\} \leftarrow \text{alloc}) \cup \{pr\_quicker \mapsto i\}\)

\(act2\): \(\text{alloc\_LR} := ((\{pr\} \leftarrow \text{alloc\_LR}) \cup \{pr\_quicker \mapsto lr\}\)

\(act3\): \(\text{pr\_available\_sm} := ((\{pr, pr\_quicker\} \leftarrow \text{pr\_available\_sm}) \cup \{pr \mapsto \text{pr\_free}\}) \cup \{pr\_quicker \mapsto \text{pr\_allocated}\}\)

manually

\(act4\): \(\text{pr\_free\_sm} := ((\{pr, pr\_quicker\} \leftarrow \text{pr\_free\_sm}) \cup \{pr \mapsto \text{Returning}\}) \cup \{pr\_quicker \mapsto \text{pr\_free\_sm\_NULL}\}\)

manually

\(act5\): \(\text{pr\_allocated\_sm} := ((\{pr, pr\_quicker\} \leftarrow \text{pr\_allocated\_sm}) \cup \{pr \mapsto \text{pr\_allocated\_sm\_NULL}\}) \cup \{pr\_quicker \mapsto \text{Mobilised}\}\)

manually

end

**Event** StopMsgReceived \(\langle\text{ordinary}\rangle\) \(\triangleq\) extends StopMsgReceived

any

\(lr\)

\(i\)
pr

where

grd_seq_allocation: $i \mapsto lr \in \text{AllocateRes}$
grd_self: $i \mapsto lr \notin \text{StopMsgReceived}$
  @grdResLostRetry $i \mapsto lr \notin \text{ResLost}$
grd_DuplicateInterrupt: $i \notin \text{IsDuplicate}$
grd_xor: $i \mapsto lr \notin \text{ResAttend}$
grd1: $i \mapsto lr \notin \text{notRequired}_ap$
grdResLostRetry: $i \mapsto lr \notin (\text{RemoveAlloc} \cup \text{RemoveBrokenRes})$
grd_seq: $i \in \text{SetInitialAP}$
grd_InputExpression1: $lr \in \text{ap}[i]$\ngrd2: $pr \in (\text{alloc}^{-1}[[i]] \cap \text{alloc}_{LR}^{-1}[[lr]])$

then

act: $\text{StopMsgReceived} := \text{StopMsgReceived} \cup \{i \mapsto lr\}$

end

Event ResAttend (ordinary) $\triangleq$

extends ResAttend

any

  $i$
  $lr$
  $pr$

where

grd_InputExpression1: $lr \in \text{ap}[i]$
grd_seq_allocation: $i \mapsto lr \in \text{AllocateRes}$
grd_self: $i \mapsto lr \notin \text{ResAttend}$
  @grdResLostRetry $i \mapsto lr \notin \text{ResLost}$
grd_DuplicateInterrupt: $i \notin \text{IsDuplicate}$
grd_xor: $i \mapsto lr \notin \text{StopMsgReceived}$
grd_seq: $i \in \text{SetInitialAP}$
grd1: $i \mapsto lr \notin \text{notRequired}_ap$
grdResLostRetry: $i \mapsto lr \notin (\text{RemoveAlloc} \cup \text{RemoveBrokenRes})$
grd2: $pr \in (\text{alloc}^{-1}[[i]] \cap \text{alloc}_{LR}^{-1}[[lr]])$
isin_Mobilised: $pr_{\text{allocated \_sm}}(pr) = \text{Mobilised}$

then

act: $\text{ResAttend} := \text{ResAttend} \cup \{i \mapsto lr\}$

enter_InAttendance: $pr_{\text{allocated \_sm}}(pr) := \text{InAttendance}$

end

Event DeallocateRes (ordinary) $\triangleq$

extends DeallocateRes

any

  $lr$
**Appendix B Fire Dispatch Model with Exception Handling**

where

$\text{grd\_seq}: \quad i \mapsto lr \in (\text{ResAttend} \cup \text{StopMsgReceived})$

$\text{grd\_self}: \quad i \mapsto lr \notin \text{DeallocateRes}$

$\text{@grdResLostRetry} \quad i \mapsto lr \notin \text{ResLost}$

$\text{grd\_DuplicateInterrupt}: \quad i \notin \text{IsDuplicate}$

$\text{@grdResLostRetry} \quad i \mapsto lr \notin (\text{RemoveAlloc} \cup \text{RemoveBrokenRes})$

$\text{grd1}: \quad pr \in (\text{alloc}^{-1}[[i]] \cap \text{alloc\_LR}^{-1}[[lr]])$

manually

$\text{isin\_pr\_allocated}: \quad \text{pr\_available\_sm}(pr) = \text{pr\_allocated}$

then

$\text{act}: \quad \text{DeallocateRes} := \text{DeallocateRes} \cup \{i \mapsto lr\}$

$\text{act1}: \quad \text{alloc} := \{pr\} \leftarrow \text{alloc}$

manually

$\text{act2}: \quad \text{alloc\_LR} := \{pr\} \leftarrow \text{alloc\_LR}$

manually

$\text{leave\_pr\_allocated\_sm}: \quad \text{pr\_allocated\_sm}(pr) := \text{pr\_allocated\_sm\_NULL}$

$\text{enter\_Returning}: \quad \text{pr\_free\_sm}(pr) := \text{Returning}$

$\text{enter\_pr\_free}: \quad \text{pr\_available\_sm}(pr) := \text{pr\_free}$

end

**Event** $\text{IncidentManaged} \quad \langle \text{ordinary} \rangle \quad \triangleq$

**extends** $\text{IncidentManaged}$

any

$i$

where

$\text{grd\_self}: \quad i \notin \text{IncidentManaged}$

$\text{grd\_DuplicateInterrupt}: \quad i \notin \text{IsDuplicate}$

$\text{grd\_seq}: \quad i \in \text{SetInitialAP}$

$\text{grd\_par}: \quad \text{AllocateRes}[[i]] = \text{ResCompTask}[[i]]$

then

$\text{act}: \quad \text{IncidentManaged} := \text{IncidentManaged} \cup \{i\}$

end

**Event** $\text{RemoveAlloc} \quad \langle \text{ordinary} \rangle \quad \triangleq$

**extends** $\text{RemoveAlloc}$

any

$i$

$lr$

where
Appendix B Fire Dispatch Model with Exception Handling

307

\[ \text{grd2: } i \mapsto lr \notin \text{alloc}^{-1}; \text{alloc}_LR \]

\[ \text{ } \text{@grd3 (alloc}^{-1}[\{i\}] \cap \text{alloc}_LR^{-1}[\{lr\}]) = \emptyset \]

\[ \text{grd_self: } i \mapsto lr \notin \text{RemoveAlloc} \]

\[ \text{grd_seq: } i \in \text{SetInitialAP} \]

\[ \text{grd_par: } i \notin \text{IncidentManaged} \]

\[ \text{grd_xor: } i \mapsto lr \notin \text{RemoveBrokenRes} \]

\[ \text{grd_DuplicateInterrupt: } i \notin \text{IsDuplicate} \]

\[ \text{grd_Retry: } i \mapsto lr \notin \text{DeallocateRes} \]

\[ \text{grd1: } i \mapsto lr \in \text{AllocateRes} \]

\[ \text{then} \]

\[ \text{act: } \text{RemoveAlloc := RemoveAlloc} \cup \{i \mapsto lr\} \]

\[ \text{act1: } \text{AllocateRes := AllocateRes} \setminus \{i \mapsto lr\} \]

\[ \text{act2: } \text{ResAttend := ResAttend} \setminus \{i \mapsto lr\} \]

\[ \text{act3: } \text{StopMsgReceived := StopMsgReceived} \setminus \{i \mapsto lr\} \]

\[ \text{act4: } \text{DeallocateRes := DeallocateRes} \setminus \{i \mapsto lr\} \]

\[ \text{end} \]

\[ \text{Event } \text{RemoveBrokenRes } (\text{ordinary}) \triangleq \]

\[ \text{extends } \text{RemoveBrokenRes} \]

\[ \text{any} \]

\[ i \]

\[ lr \]

\[ pr \]

\[ \text{where} \]

\[ \text{grd_DuplicateInterrupt: } i \notin \text{IsDuplicate} \]

\[ \text{grd_self: } i \mapsto lr \notin \text{RemoveAlloc} \]

\[ \text{grd_seq: } i \in \text{SetInitialAP} \]

\[ \text{grd_par: } i \notin \text{IncidentManaged} \]

\[ \text{grd_xor: } i \mapsto lr \notin \text{RemoveBrokenRes} \]

\[ \text{grd1: } i \mapsto lr \in \text{AllocateRes} \]

\[ \text{grd2: } i \mapsto lr \notin \text{DeallocateRes} \]

\[ \text{grd3: } pr \in (\text{alloc}^{-1}[\{i\}] \cap \text{alloc}_LR^{-1}[\{lr\}]) \]

\[ \text{isin_pr_allocated: } \text{pr}_{\text{available}}_{\text{sm}}(pr) = \text{pr}_{\text{allocated}} \]

\[ \text{then} \]

\[ \text{act: } \text{RemoveBrokenRes := RemoveBrokenRes} \cup \{i \mapsto lr\} \]

\[ \text{act1: } \text{AllocateRes := AllocateRes} \setminus \{i \mapsto lr\} \]

\[ \text{act2: } \text{ResAttend := ResAttend} \setminus \{i \mapsto lr\} \]

\[ \text{act3: } \text{StopMsgReceived := StopMsgReceived} \setminus \{i \mapsto lr\} \]

\[ \text{act4: } \text{DeallocateRes := DeallocateRes} \setminus \{i \mapsto lr\} \]

\[ \text{act5: } \text{alloc := } \{pr\} \triangleleft \text{alloc} \]

\[ \text{manually} \]

\[ \text{act6: } \text{alloc}_LR := \{pr\} \triangleleft \text{alloc}_LR \]

\[ \text{manually} \]
\[ \text{leave}_{\text{pr\_allocated\_sm}}: \ pr\_allocated\_sm(pr) := \pr\_allocated\_sm\_NULL \]
\[ \text{leave}_{\text{pr\_available\_sm}}: \ pr\_available\_sm(pr) := \pr\_available\_sm\_NULL \]
\[ \text{leave}_{\text{pr\_free\_sm}}: \ pr\_free\_sm(pr) := \pr\_free\_sm\_NULL \]
\[ \text{enter}_{\text{Unavailable}}: \ pr\_status(pr) := \text{Unavailable} \]

**Event** Reset\_ResLost \(\langle\text{ordinary}\rangle\) \(\models\)

**extends** Reset\_ResLost

any

\(i\)
\(lr\)

where

\(\text{grd\_seq}: \ i \mapsto lr \in (\text{RemoveBrokenRes} \cup \text{RemoveAlloc})\)

then

act1: \(\text{RemoveAlloc} := \text{RemoveAlloc} \setminus \{i \mapsto lr\}\)

act2: \(\text{RemoveBrokenRes} := \text{RemoveBrokenRes} \setminus \{i \mapsto lr\}\)

**Event** CloseIncident \(\langle\text{ordinary}\rangle\) \(\models\)

**extends** CloseIncident

any

\(i\)

where

\(\text{grd\_seq}: \ i \notin \text{ran}(\text{alloc})\)

\(\text{grd\_seq\_FD}: \ (i \in \text{SendStopMsg} \land i \in \text{IncidentManaged})\)

\(\text{grd\_self}: \ i \notin \text{CloseIncident}\)

\(\text{grd\_DuplicateInterrupt}: \ i \notin \text{IsDuplicate}\)

\(\text{grd\_1}: \ \text{ap}[[i]] \setminus \text{notRequired}_\text{ap}[[i]] \subseteq \text{DeallocateRes}[[i]]\)

\(\text{to ensure all action plan items are satisfied}\)

then

act: \(\text{CloseIncident} := \text{CloseIncident} \cup \{i\}\)

**Event** IsDuplicate \(\langle\text{ordinary}\rangle\) \(\models\)

**extends** IsDuplicate

any

\(i\)

where

\(\text{grd\_seq\_DuplicateInterrupt}: \ i \in \text{CreateIncident}\)

\(\text{grd\_self}: \ i \notin \text{IsDuplicate}\)

\(\text{grd\_1}: \ i \notin \text{CloseIncident}\)

then

act: \(\text{IsDuplicate} := \text{IsDuplicate} \cup \{i\}\)
Appendix B Fire Dispatch Model with Exception Handling

```
end
Event HandleDupAlloc ⟨ordinary⟩ ⋆=
extends HandleDupAlloc
  any
  i
  lr
  pr
where
  grd_seq: i ∈ IsDuplicate
  grd_self: i ↦ lr /∈ HandleDupAlloc
  grd_exp: lr ∈ AllocateRes[[i]]
  grd1: pr ∈ (alloc^−1[[i]] ∩ alloc_LR^−1[[lr]])
  isin_pr_allocated: pr_available_sm(pr) = pr_allocated
then
  act: HandleDupAlloc := HandleDupAlloc ∪ {i ↦ lr}
  act1: alloc := {pr} ⪯ alloc
  act26: alloc_LR := {pr} ⪯ alloc_LR
  leave_pr_allocated_sm: pr_allocated_sm(pr) := pr_allocated_sm(NULL
  enter_Returning: pr_free_sm(pr) := Returning
  enter_pr_free: pr_available_sm(pr) := pr_free
end
Event CloseDuplicate ⟨ordinary⟩ ⋆=
extends CloseDuplicate
  any
  i
where
  grd_seq: HandleDupAlloc[[i]] = AllocateRes[[i]]
  grd_self: i /∈ CloseDuplicate
  grd1: i ∈ IsDuplicate
  grd2: i /∈ ran(alloc)
then
  act: CloseDuplicate := CloseDuplicate ∪ [i]
end
Event ResCompTask ⟨ordinary⟩ ⋆=
extends ResCompTask
  any
  lr
  i
where
  grd DuplicateInterrupt: i /∈ IsDuplicate
```
grd_seq: \( i \mapsto lr \in \text{DeallocateRes} \)

grd_self: \( i \mapsto lr \notin \text{ResCompTask} \)

\[ \text{then} \]

act: \( \text{ResCompTask} := \text{ResCompTask} \cup \{ i \mapsto lr \} \)

\[ \text{end} \]

**Event** FixRes (ordinary) \( \triangleq \)

any

pr

where

isin_Unavailable: \( pr\_status(pr) = \text{Unavailable} \)
grd1: \( pr \notin \text{dom}(\text{alloc}) \)

\[ \text{then} \]

enter_StationAvailable: \( pr\_free\_sm(pr) := \text{StationAvailable} \)
enter_pr_free: \( pr\_available\_sm(pr) := pr\_free \)
enter_Available: \( pr\_status(pr) := \text{Available} \)

\[ \text{end} \]

**Event** FreeResBreakdown (ordinary) \( \triangleq \)

any

pr

where

isin_pr_free: \( pr\_available\_sm(pr) = pr\_free \)

\[ \text{then} \]

leave_pr_free_sm: \( pr\_free\_sm(pr) := pr\_free\_sm\_NULL \)
leave_pr_available_sm: \( pr\_available\_sm(pr) := pr\_available\_sm\_NULL \)
leave_pr_allocated_sm: \( pr\_allocated\_sm(pr) := pr\_allocated\_sm\_NULL \)
enter_Unavailable: \( pr\_status(pr) := \text{Unavailable} \)

\[ \text{end} \]

**Event** BookResHome (ordinary) \( \triangleq \)

any

pr

where

isin_Returning: \( pr\_free\_sm(pr) = \text{Returning} \)

\[ \text{then} \]

enter_StationAvailable: \( pr\_free\_sm(pr) := \text{StationAvailable} \)

\[ \text{end} \]

**Event** UpdateLoc_RetRes (ordinary) \( \triangleq \)

any

pr

where

isin_Returning: \( pr\_free\_sm(pr) = \text{Returning} \)

\[ \text{then} \]
B.2.11 Tenth Refinement: M10

MACHINE M10
REFINES M9
SEES C5,M9,implicitContext

VARIABLES
  pr_status
  pr_available_sm
  pr_allocated_sm
  pr_free_sm
  CreateIncident
  SelectLocation
  SelectType
  CompInfo
  SetInitialAP
  AllocateRes
  ResAttend
  DeallocateRes
  StopMsgReceived
  IncidentManaged
  RequestAdditionalRes
  SendStopMsg
  CloseIncident
  IsDuplicate
  ap
  notRequired_ap
  iLocation
  iType
  DisablePriorityChange
  priority
  RemoveAlloc
  RemoveBrokenRes
  alloc allocate physical
  alloc_LR relate lr to pr
Appendix B Fire Dispatch Model with Exception Handling

HandleDupAlloc
CloseDuplicate
ResCompTask

rLocation location of a physical resource
duration duration from one location to another this is a variable because it depends on
time and not distance which can be affected by traffic

iLocType

INVARIANTS

inv_r_loc_type: rLocation ∈ PR → LOCATION
inv_duration_type: duration ∈ (LOCATION × LOCATION) → N

EVENTS

Initialisation (extended)

begin

act_CreateIncident: CreateIncident := Ø
act_SetInitialAP: SetInitialAP := Ø
act_CloseIncident: CloseIncident := Ø
act_IsDuplicate: IsDuplicate := Ø
act_SendStopMsg: SendStopMsg := Ø
act_ap: ap := Ø
act_notRequired_ap: notRequired_ap := Ø
act_RequestAdditionalRes: RequestAdditionalRes := Ø
act2: iLocation := Ø
act3: iType := Ø
act_loc_type: iLocType := Ø
act_SelectLocation: SelectLocation := Ø
act_SelectType: SelectType := Ø
act_CompInfo: CompInfo := Ø
act_DisablePriorityChange: DisablePriorityChange := Ø
act4: priority := Ø
act_StopMsgReceived: StopMsgReceived := Ø
act_AllocateRes: AllocateRes := Ø
act_Attend: ResAttend := Ø
act_DeallocateRes: DeallocateRes := Ø
act_IncidentManaged: IncidentManaged := Ø
act_RemoveAlloc: RemoveAlloc := Ø
act_RemoveBrokenRes: RemoveBrokenRes := Ø
act_alloc: alloc := Ø
act_alloc_LR: alloc_LR := Ø
act_HandleDupAlloc: HandleDupAlloc := Ø
act_CloseDuplicate: CloseDuplicate := Ø
Appendix B Fire Dispatch Model with Exception Handling

\[\text{act} \text{ ResCompTask}: \text{ResCompTask} := \emptyset\]
\[\text{init} \text{pr} \text{free}\_sm: \text{pr} \text{free}\_sm := PR \times \{\text{StationAvailable}\}\]
\[\text{init} \text{pr} \text{available}\_sm: \text{pr} \text{available}\_sm := PR \times \{\text{pr} \text{free}\}\]
\[\text{init} \text{pr} \text{status}: \text{pr} \text{status} := PR \times \{\text{Available}\}\]
\[\text{init} \text{pr} \text{allocated}\_sm: \text{pr} \text{allocated}\_sm := PR \times \{\text{pr} \text{allocated}\_sm\_NULL\}\]
\[\text{act} \text{r} \text{location}: \text{rLocation} := rl0\]
\[\text{act} \text{duration}: \text{duration} := \text{duration}0\]
\end{Event}

\text{CreateIncident (ordinary)} \triangleq
\text{extends} \text{CreateIncident}
\begin{any}
\text{where}
\text{grid}\_\text{self}: \ i \notin \text{CreateIncident}
\text{then}
\text{act}: \text{CreateIncident} := \text{CreateIncident} \cup \{i\}\end{any}
\end{Event}

\text{SelectLocation (ordinary)} \triangleq
\text{extends} \text{SelectLocation}
\begin{any}
\text{where}
\text{grid}\_\text{seq}: \ i \in \text{CreateIncident}
\text{grid}\_\text{self}: \ i \notin \text{SelectLocation}
\text{grid}\_\text{DuplicateInterrupt}: \ i \notin \text{IsDuplicate}
\text{grid}\_1: \ l \in \text{LOCATION}
\text{grid}\_2: \ lt \in \text{LOCATION\_TYPE}
\text{then}
\text{act}: \text{SelectLocation} := \text{SelectLocation} \cup \{i\}
\text{act}1: \text{Location}(i) := l
\text{act}2: \text{LocType}(i) := lt
\end{any}
\end{Event}

\text{SelectType (ordinary)} \triangleq
\text{extends} \text{SelectType}
\begin{any}
\end{any}
where
\[
\text{grd\_seq}.: \quad i \in \text{CreateIncident}
\]
\[
\text{grd\_self}.: \quad i \notin \text{SelectType}
\]
\[
\text{grd1}.: \quad t \in \text{INCIDENT}\_\text{TYPE}
\]
\[
\text{grd\_DuplicateInterrupt}.: \quad i \notin \text{IsDuplicate}
\]
\[
\text{grd2}.: \quad p = \text{default\_priority}(t)
\]
then
\[
\text{act}.: \quad \text{SelectType} := \text{SelectType} \cup \{i\}
\]
\[
\text{act1}.: \quad \text{iType}(i) := t
\]
\[
\text{act2}.: \quad \text{priority}(i) := p
\]
end

**Event** \ ChangePriority \ (ordinary) \ ⊳

**extends** \ ChangePriority

any
\[
i
\]
\[
p
\]
where
\[
\text{grd\_seq\_priority}.: \quad i \in \text{SelectType}
\]
\[
\text{grd0\_loop}.: \quad i \notin \text{DisablePriorityChange}
\]
\[
\text{grd1}.: \quad p \in 1..3
\]
\[
\text{grd2}.: \quad p \neq \text{priority}(i)
\]
\[
\text{grd3}.: \quad i \notin \text{CloseIncident}
\]
\quad \text{we\ can\ also\ set\ DisablePriorityChange\ in\ CloseIncident}
\]
\[
\text{grd\_DuplicateInterrupt}.: \quad i \notin \text{IsDuplicate}
\]
then
\[
\text{act1}.: \quad \text{priority}(i) := p
\]
end

**Event** \ DisablePriorityChange \ (ordinary) \ ⊳

**extends** \ DisablePriorityChange

any
\[
i
\]
where
\[
\text{grd\_DuplicateInterrupt}.: \quad i \notin \text{IsDuplicate}
\]
\[
\text{grd\_seq\_priority}.: \quad i \in \text{SelectType}
\]
\[
\text{grd\_self}.: \quad i \notin \text{DisablePriorityChange}
\]
\[
\text{grd1}.: \quad i \notin \text{CloseIncident}
\]
\quad \text{we\ can\ also\ set\ DisablePriorityChange\ in\ CloseIncident}
\]
then
\[
\text{act}.: \quad \text{DisablePriorityChange} := \text{DisablePriorityChange} \cup \{i\}
\]
end

**Event** \ CompInfo \ (ordinary) \ ⊳
extends CompInfo

any

i

where

grd DuplicateInterrupt: \( i \notin IsDuplicate \)
grd seq.: \( (i \in SelectLocation \land i \in SelectType) \)
grd self: \( i \notin CompInfo \)

then

act: CompInfo := CompInfo \cup \{i\}

end

Event SetInitialAP \( (\text{ordinary}) \supseteq \)

extends SetInitialAP

any

lrs

i

t

l

where

grd seq: \( i \in CompInfo \)
grd self: \( i \notin SetInitialAP \)
grd1: \( t = iType(i) \)
grd2: \( l = iLocType(i) \)
grd3: \( lrs = PDA(t \mapsto l) \)
grd DuplicateInterrupt: \( i \notin IsDuplicate \)

then

act: SetInitialAP := SetInitialAP \cup \{i\}
act1: ap := ap \cup (\{i\} \times lrs)

end

Event RequestAdditionalRes \( (\text{ordinary}) \supseteq \)

extends RequestAdditionalRes

any

i

lr

where

grd seq.planF: \( i \in SetInitialAP \)
grd self: \( i \mapsto lr \notin RequestAdditionalRes \)
grd InputExpression1: \( lr \in LR \)
grd0 par: \( i \notin SendStopMsg \)
grd DuplicateInterrupt: \( i \notin IsDuplicate \)
grd1: \( lr \notin ap[\{i\}] \)

then
\textbf{act1}: \( ap := ap \cup \{i \mapsto lr \} \)

\textbf{act}: \( \text{RequestAdditionalRes} := \text{RequestAdditionalRes} \cup \{i \mapsto lr\} \)

\textbf{end}

\textbf{Event} \( \text{SendStopMsg} \) (ordinary) \( \models \)
\textbf{extends} \( \text{SendStopMsg} \)

\begin{align*}
\textbf{any} & \\
& i \\
& ls \\
\textbf{where} & \\
& \text{grd}_ls: \; ls = ap[i] \setminus \text{ResAttend}[i] \\
& \text{grd}_\text{DuplicateInterrupt}: \; i \notin \text{IsDuplicate} \\
& \text{grd}_\text{seq}: \; i \in \text{SetInitialAP} \\
& \text{grd}_\text{self}: \; i \notin \text{SendStopMsg} \\
\textbf{then} & \\
& \text{act1}: \; \text{notRequired}_ap := \text{notRequired}_ap \cup \{i \times ls\} \\
& \text{act}: \; \text{SendStopMsg} := \text{SendStopMsg} \cup \{i\} \\
\textbf{end}
\end{align*}

\textbf{Event} \( \text{AllocateRes} \) (ordinary) \( \models \)
\textbf{extends} \( \text{AllocateRes} \)

\begin{align*}
\textbf{any} & \\
& i \\
& lr \\
& pr \\
\textbf{where} & \\
& \text{grd}_\text{seq_allocation}: \; i \in \text{SetInitialAP} \\
& \text{grd}_\text{self}: \; i \mapsto lr \notin \text{AllocateRes} \\
& \text{grd}_\text{InputExpression1}: \; lr \in ap[i] \\
& \text{grd}_\text{0.par}: \; i \notin \text{IncidentManaged} \\
& \text{grd}_\text{DuplicateInterrupt}: \; i \notin \text{IsDuplicate} \\
& \text{grd1}: \; i \mapsto lr \notin \text{notRequired}_ap \\
& \text{grdResLostRetry}: \; i \mapsto lr \notin (\text{RemoveAlloc} \cup \text{RemoveBrokenRes}) \\
& \text{grd}_2: \; \text{ptype}(pr) = \text{ltype}(lr) \\
& \text{grd}_3: \; pr \in \text{dom}(alloc) \Rightarrow \text{priority}(i) < \text{priority}(\text{alloc}(pr)) \\
& \text{isinMobilised_or_isin_pr_free}: \; (\text{pr_allocated_sm}(pr) = \text{Mobilised} \lor \text{pr_available_sm}(pr) = \text{pr_free}) \\
\textbf{then} & \\
& \textbf{act}: \; \text{AllocateRes} := \text{AllocateRes} \cup \{i \mapsto lr\} \\
& \text{act1}: \; \text{alloc}(pr) := i \\
& \text{act2}: \; \text{alloc}_{LR}(pr) := lr \\
& \text{leave_pr_free_sm}: \; \text{pr_free_sm}(pr) := \text{pr_free_sm_NULL} \\
& \text{enter_Mobilised}: \; \text{pr_allocated_sm}(pr) := \text{Mobilised}
\end{align*}
**Appendix B Fire Dispatch Model with Exception Handling**

**enter_pr_allocated:** $pr_{available}\_sm(pr) := pr_{allocated}$

**end**

**Event** UpdateResLocation $\langle$ordinary$\rangle \supseteq$

**extends** UpdateResLocation

**any**

$i$

$lr$

$pr$

$l$

$l0$

**where**

$grd\_seq\_allocation$: $i \mapsto lr \in AllocateRes$

$grd0\_loop$: $i \mapsto lr \notin (ResAttend \cup StopMsgReceived)$

@grdResLostRetry $i \mapsto lr \notin ResLost$

$grd\_Duplicate\_Interrupt$: $i \notin Is\_Duplicate$

$grd1$: $i \mapsto lr \notin not\_Required\_ap$

$grdResLostRetry$: $i \mapsto lr \notin (Remove\_Alloc \cup Remove\_Broken\_Res)$

$grd2$: $pr \in (alloc^{-1}[(i)] \cap alloc_{LR}^{-1}[(lr)])$

$isin\_Mobilised$: $pr_{allocated}\_sm(pr) = Mobilised$

$grd\_l0$: $l0 = r\_Location(pr)$

$grd\_l1$: $l \in LOCATION \setminus \{l0\}$

$grd\_dur$: $duration(l \mapsto i\_Location(i)) < duration(l0 \mapsto i\_Location(i))$

**then**

$act\_loc$: $r\_Location(pr) := l$

**end**

**Event** ReallocateQuickerRes $\langle$ordinary$\rangle \supseteq$

**extends** ReallocateQuickerRes

**any**

$i$

$lr$

$pr$

$pr\_quicker$

**where**

$grd\_seq\_allocation$: $i \mapsto lr \in AllocateRes$

$grd0\_loop$: $i \mapsto lr \notin (ResAttend \cup StopMsgReceived)$

@grdResLostRetry $i \mapsto lr \notin ResLost$

$grd\_Duplicate\_Interrupt$: $i \notin Is\_Duplicate$

$grd1$: $i \mapsto lr \notin not\_Required\_ap$

$grdResLostRetry$: $i \mapsto lr \notin (Remove\_Alloc \cup Remove\_Broken\_Res)$

$grd2$: $pr \in (alloc^{-1}[(i)] \cap alloc_{LR}^{-1}[(lr)])$

$grd3$: $pr\_quicker \notin dom(alloc)$
Appendix B Fire Dispatch Model with Exception Handling

\[ p_{\text{type}}(pr_{\text{quicker}}) = l_{\text{type}}(lr) \]

@\text{grid 5} p_{\text{type}}(pr) = l_{\text{type}}(lr)

\[ pr_{\text{available}}_{\text{sm}}(pr_{\text{quicker}}) = pr_{\text{free}} \]

manually

\[ pr_{\text{allocated}}_{\text{sm}}(pr) = \text{Mobilised} \]

manually

\[ \text{duration}(r_{\text{Location}}(pr_{\text{quicker}}) \mapsto i_{\text{Location}}(i)) < \text{duration}(r_{\text{Location}}(pr) \mapsto i_{\text{Location}}(i)) \]

\text{then}

\text{act 1:} \text{alloc} := \{\text{pr}\} \triangleleft \text{alloc} \cup \{pr_{\text{quicker}} \mapsto i\}

\text{act 2:} \text{alloc}_{LR} := \{\text{pr}\} \triangleleft \text{alloc}_{LR} \cup \{pr_{\text{quicker}} \mapsto lr\}

\text{act 3:} pr_{\text{available}}_{\text{sm}} := \{\{pr, pr_{\text{quicker}}\} \triangleleft \text{pr}_{\text{available}}_{\text{sm}}\} \cup \{\text{pr} \mapsto \text{pr}_{\text{free}}\} \cup \{pr_{\text{quicker}} \mapsto pr_{\text{allocated}}\}

manually

\text{act 4:} pr_{\text{free}}_{sm} := \{\{pr, pr_{\text{quicker}}\} \triangleleft \text{pr}_{\text{free}}_{sm}\} \cup \{\text{pr} \mapsto \text{Returning}\} \cup \{pr_{\text{quicker}} \mapsto \text{pr}_{\text{free}}_{sm\_\text{NULL}}\}

manually

\text{act 5:} pr_{\text{allocated}}_{sm} := \{\{pr, pr_{\text{quicker}}\} \triangleleft \text{pr}_{\text{allocated}}_{sm}\} \cup \{\text{pr} \mapsto \text{pr}_{\text{allocated}}_{sm\_\text{NULL}}\} \cup \{pr_{\text{quicker}} \mapsto \text{Mobilised}\}

manually

\text{end}

\text{Event} \text{StopMsgReceived} (\text{ordinary}) \equiv \text{extends} \text{StopMsgReceived}

\text{any}

\text{lr}

\text{i}

\text{pr}

\text{where}

\text{grid seq allocation:} \ i \mapsto lr \in \text{AllocateRes}

\text{grid self:} \ i \mapsto lr \notin \text{StopMsgReceived}

@\text{gridResLostRetry} i \mapsto lr \notin \text{ResLost}

\text{grid DuplicateInterrupt:} \ i \notin \text{IsDuplicate}

\text{grid xor:} \ i \mapsto lr \notin \text{ResAttend}

\text{grid1:} \ i \mapsto lr \in \text{notRequired}_ap

\text{gridResLostRetry:} \ i \mapsto lr \notin (\text{RemoveAlloc} \cup \text{RemoveBrokenRes})

\text{grid seq:} \ i \in \text{SetInitialAP}

\text{grid InputExpression:} \ lr \in \text{ap}[[\{i\}]]

\text{grid2:} \ pr \in (\text{alloc}^{-1}[[\{i\}]] \cap \text{alloc}_{LR}^{-1}[[lr]])

\text{then}

\text{act:} \text{StopMsgReceived} := \text{StopMsgReceived} \cup \{i \mapsto lr\}

\text{end}
**Event** ResAttend (ordinary) \( \triangleq \)

**extends** ResAttend

\[
\text{any} \\
  i \\
  lr \\
  pr
\]

**where**

- \( \text{grd} \text{InputExpression1}: lr \in \text{ap}\{i\} \)
- \( \text{grd seq allocation}: i \mapsto lr \in \text{AllocateRes} \)
- \( \text{grd self}: i \mapsto lr \notin \text{ResAttend} \)
  
  @\text{grdResLostRetry} i \mapsto lr \notin \text{ResLost} 

- \( \text{grd DuplicateInterrupt}: i \notin \text{IsDuplicate} \)
- \( \text{grd xor}: i \mapsto lr \notin \text{StopMsgReceived} \)
- \( \text{grd seq.}: i \in \text{SetInitialAP} \)
- \( \text{grd1}: i \mapsto lr \notin \text{notRequired}\_\text{ap} \)
- \( \text{grdResLostRetry}: i \mapsto lr \notin (\text{RemoveAlloc} \cup \text{RemoveBrokenRes}) \)
- \( \text{grd2}: pr \in (\text{alloc}^{-1}\{i\} \cap \text{alloc}\_\text{LR}^{-1}\{lr\}) \)
- \( \text{isin Mobilised}: \text{pr} \text{ allocated sm}(pr) = \text{Mobilised} \)
- \( \text{grd loc}: \text{rLocation}(pr) = \text{iLocation}(i) \)

**then**

\[
\text{act: ResAttend} := \text{ResAttend} \cup \{i \mapsto lr\} \\
\text{enter InAttendance}: \text{pr} \text{ allocated sm}(pr) := \text{InAttendance}
\]

**end**

**Event** DeallocateRes (ordinary) \( \triangleq \)

**extends** DeallocateRes

\[
\text{any} \\
  lr \\
  i \\
  pr
\]

**where**

- \( \text{grd seq.:} i \mapsto lr \in (\text{ResAttend} \cup \text{StopMsgReceived}) \)
- \( \text{grd self:} i \mapsto lr \notin \text{DeallocateRes} \)
  
  @\text{grdResLostRetry} i \mapsto lr \notin \text{ResLost} 

- \( \text{grd DuplicateInterrupt:} i \notin \text{IsDuplicate} \)
  
  @\text{grd1} lr \in \text{ap}\{i\} // \text{GRD PO}//@\text{grdResLostRetry} i \mapsto lr \notin \text{ResLost} 

- \( \text{grdResLostRetry:} i \mapsto lr \notin (\text{RemoveAlloc} \cup \text{RemoveBrokenRes}) \)
  
  @\text{grd1} lr \in \text{ap}\{i\} // \text{GRD PO}//@\text{grdResLostRetry} i \mapsto lr \notin \text{ResLost} 

- \( \text{grd1:} pr \in (\text{alloc}^{-1}\{i\} \cap \text{alloc}\_\text{LR}^{-1}\{lr\}) \)

  manually

- \( \text{isin pr allocated:} \text{pr} \text{ available sm}(pr) = \text{pr} \text{ allocated} \)

**then**

\[
\text{act: DeallocateRes} := \text{DeallocateRes} \cup \{i \mapsto lr\}
\]
Appendix B Fire Dispatch Model with Exception Handling

act1: \[\text{alloc} := \{\text{pr}\} \triangleleft \text{alloc}\]
manually

act2: \[\text{alloc}_{LR} := \{\text{pr}\} \triangleleft \text{alloc}_{LR}\]
manually

leave\_pr\_allocated\_sm: \[\text{pr\_allocated\_sm}(\text{pr}) := \text{pr\_allocated\_sm\_NULL}\]

enter\_Returning: \[\text{pr\_free\_sm}(\text{pr}) := \text{Returning}\]

enter\_pr\_free: \[\text{pr\_available\_sm}(\text{pr}) := \text{pr\_free}\]

end

Event IncidentManaged \langle\langle\text{ordinary}\rangle\rangle \equiv
extends IncidentManaged

\text{any}
\text{i}

where

grd\_self: \text{i} \notin \text{IncidentManaged}
grd\_Duplicate\_Interrupt: \text{i} \notin \text{IsDuplicate}
grd\_seq: \text{i} \in \text{SetInitialAP}
grd\_par: \text{AllocateRes}\{(\text{i})\} = \text{ResCompTask}\{(\text{i})\}

then

act: \[\text{IncidentManaged} := \text{IncidentManaged} \cup \{\text{i}\}\]

end

Event RemoveAlloc \langle\langle\text{ordinary}\rangle\rangle \equiv
extends RemoveAlloc

\text{any}
\text{i}
\text{lr}

where

grd2: \text{i} \mapsto \text{lr} \notin \text{alloc}^{-1}; \text{alloc}_{LR}
\tag{grd3} \text{alloc}^{-1}\{(\text{i})\} \cap \text{alloc}_{LR}^{-1}\{(\text{lr})\} = \emptyset

grd\_self: \text{i} \mapsto \text{lr} \notin \text{RemoveAlloc}
grd\_seq: \text{i} \in \text{SetInitialAP}
grd\_par: \text{i} \notin \text{IncidentManaged}
grd\_xor: \text{i} \mapsto \text{lr} \notin \text{RemoveBrokenRes}
grd\_Duplicate\_Interrupt: \text{i} \notin \text{IsDuplicate}
grd\_Retry: \text{i} \mapsto \text{lr} \notin \text{DeallocateRes}
grd1: \text{i} \mapsto \text{lr} \in \text{AllocateRes}

then

act: \[\text{RemoveAlloc} := \text{RemoveAlloc} \cup \{\text{i} \mapsto \text{lr}\}\]
act1: \[\text{AllocateRes} := \text{AllocateRes} \setminus \{\text{i} \mapsto \text{lr}\}\]
act2: \[\text{ResAttend} := \text{ResAttend} \setminus \{\text{i} \mapsto \text{lr}\}\]
act3: \[\text{StopMsgReceived} := \text{StopMsgReceived} \setminus \{\text{i} \mapsto \text{lr}\}\]
act4: \[\text{DeallocateRes} := \text{DeallocateRes} \setminus \{\text{i} \mapsto \text{lr}\}\]
end

Event RemoveBrokenRes (ordinary) \(\triangleq\)
extends RemoveBrokenRes

\[
\begin{aligned}
\text{any} & \quad i \\
& \quad lr \\
& \quad pr \\
\text{where} & \\
\text{grd\_DuplicateInterrupt: } & i \notin \text{IsDuplicate} \\
\text{grd\_self: } & i \mapsto lr \notin \text{RemoveBrokenRes} \\
\text{grd\_seq: } & i \in \text{SetInitialAP} \\
\text{grd\_par: } & i \notin \text{IncidentManaged} \\
\text{grd\_xor: } & i \mapsto lr \notin \text{RemoveAlloc} \\
\text{grd1: } & i \mapsto lr \in \text{AllocateRes} \\
\text{grd2: } & i \mapsto lr \notin \text{DeallocateRes} \\
\text{grd3: } & pr \in (\text{alloc}^{-1}[[i]] \cap \text{alloc\_LR}^{-1}[[lr]]) \\
\text{isin\_pr\_allocated: } & \text{pr\_available\_sm}(pr) = \text{pr\_allocated} \\
\text{then} & \\
\text{act: } & \text{RemoveBrokenRes} := \text{RemoveBrokenRes} \cup \{ i \mapsto lr \} \\
\text{act1: } & \text{AllocateRes} := \text{AllocateRes} \setminus \{ i \mapsto lr \} \\
\text{act2: } & \text{ResAttend} := \text{ResAttend} \setminus \{ i \mapsto lr \} \\
\text{act3: } & \text{StopMsgReceived} := \text{StopMsgReceived} \setminus \{ i \mapsto lr \} \\
\text{act4: } & \text{DeallocateRes} := \text{DeallocateRes} \setminus \{ i \mapsto lr \} \\
\text{act5: } & \text{alloc} := \{ pr \} \leftarrow alloc \\
& \quad \text{manually} \\
\text{act6: } & \text{alloc\_LR} := \{ pr \} \leftarrow \text{alloc\_LR} \\
& \quad \text{manually} \\
\text{leave\_pr\_allocated\_sm: } & \text{pr\_allocated\_sm}(pr) := \text{pr\_allocated\_sm\_NULL} \\
\text{leave\_pr\_available\_sm: } & \text{pr\_available\_sm}(pr) := \text{pr\_available\_sm\_NULL} \\
\text{leave\_pr\_free\_sm: } & \text{pr\_free\_sm}(pr) := \text{pr\_free\_sm\_NULL} \\
\text{enter\_Unavailable: } & \text{pr\_status}(pr) := \text{Unavailable} \\
\end{aligned}
\]

end

Event Reset\_ResLost (ordinary) \(\triangleq\)
extends Reset\_ResLost

\[
\begin{aligned}
\text{any} & \quad i \\
& \quad lr \\
\text{where} & \\
\text{grd\_seq: } & i \mapsto lr \in (\text{RemoveBrokenRes} \cup \text{RemoveAlloc}) \\
\text{then} & \\
\text{act1: } & \text{RemoveAlloc} := \text{RemoveAlloc} \setminus \{ i \mapsto lr \}
\end{aligned}
\]
act2: RemoveBrokenRes := RemoveBrokenRes \ {i ↦ lr}
end

Event CloseIncident ⟨ordinary⟩ ≡
extends CloseIncident
any
  i
where
  grd2: i ∉ ran(alloc)
  grd_seq_FD: (i ∈ SendStopMsg ∧ i ∈ IncidentManaged)
  grd_self: i ∉ CloseIncident
  grd_DuplicateInterrupt: i ∉ IsDuplicate
  grd1: ap[{i}] \notRequired ap[{i}] ⊆ DeallocateRes[{i}]
  to ensure all action plan items are satisfied
then
  act: CloseIncident := CloseIncident ∪ {i}
end

Event IsDuplicate ⟨ordinary⟩ ≡
extends IsDuplicate
any
  i
where
  grd_seq_DuplicateInterrupt: i ∈ CreateIncident
  grd_self: i ∉ IsDuplicate
  grd1: i ∉ CloseIncident
then
  act: IsDuplicate := IsDuplicate ∪ {i}
end

Event HandleDupAlloc ⟨ordinary⟩ ≡
extends HandleDupAlloc
any
  i
  lr
  pr
where
  grd_seq: i ∈ IsDuplicate
  grd_self: i ↦ lr ∉ HandleDupAlloc
  grd_exp: lr ∈ AllocateRes[{i}]
  grd1: pr ∈ (alloc⁻¹[{i}] ∩ alloc_LR⁻¹[{lr}])
isin_pr_allocated: pr_available_sm(pr) = pr_allocated
then
act: \( \text{HandleDupAlloc} := \text{HandleDupAlloc} \cup \{ i \mapsto lr \} \)
act1: \( \text{alloc} := \{ pr \} \triangleleft \text{alloc} \)
act26: \( \text{alloc} _{LR} := \{ pr \} \triangleleft \text{alloc} _{LR} \)
leave_pr_allocated_sm: \( \text{pr}_{\text{allocated_sm}}(pr) := \text{pr}_{\text{allocated_sm}}(pr) \text{_NULL} \)
enter_Returning: \( \text{pr}_{\text{free_sm}}(pr) := \text{Returning} \)
enter_pr_free: \( \text{pr}_{\text{available_sm}}(pr) := \text{pr}_{\text{free}} \)

end

Event CloseDuplicate (ordinary) \( \Rightarrow \)
extends CloseDuplicate

any

i

where

grd_seq: \( \text{HandleDupAlloc} \{ \{ i \} \} = \text{AllocateRes} \{ \{ i \} \} \)
grd_self: \( i \notin \text{CloseDuplicate} \)
grd1: \( i \in \text{IsDuplicate} \)
grd2: \( i \notin \text{ran} \left( \text{alloc} \right) \)
then
act: \( \text{CloseDuplicate} := \text{CloseDuplicate} \cup \{ i \} \)
end

Event ResCompTask (ordinary) \( \Rightarrow \)
extends ResCompTask

any

lr

i

where

grd_DuplicateInterrupt: \( i \notin \text{IsDuplicate} \)
grd_seq: \( i \mapsto lr \in \text{DealocateRes} \)
grd_self: \( i \mapsto lr \notin \text{ResCompTask} \)
then
act: \( \text{ResCompTask} := \text{ResCompTask} \cup \{ i \mapsto lr \} \)
end

Event FixRes (ordinary) \( \Rightarrow \)
extends FixRes

any

pr

where

isin_Unavailable: \( \text{pr}_{\text{status}}(pr) = \text{Unavailable} \)
grd1: \( pr \notin \text{dom} \left( \text{alloc} \right) \)
then
enter_StationAvailable: \( \text{pr}_{\text{free_sm}}(pr) := \text{StationAvailable} \)


\begin{verbatim}
enter_pr_free: pr_available_sm(pr) := pr_free
enter_Available: pr_status(pr) := Available
act1: rLocation(pr) := rl0(pr)

end

Event FreeResBreakdown (ordinary) \(\triangleq\)
extends FreeResBreakdown
  any
  pr
  where
    isin_pr_free: pr_available_sm(pr) = pr_free
  then
    leave_pr_free_sm: pr_free_sm(pr) := pr_free_sm_NULL
    leave_pr_available_sm: pr_available_sm(pr) := pr_available_sm_NULL
    leave_pr_allocated_sm: pr_allocated_sm(pr) := pr_allocated_sm_NULL
    enter_Unavailable: pr_status(pr) := Unavailable

end

Event BookResHome (ordinary) \(\triangleq\)
extends BookResHome
  any
  pr
  where
    isin_Returning: pr_free_sm(pr) = Returning
    grd_loc: rLocation(pr) \in ran(s_location)
    can make it rl0
  then
    enter_StationAvailable: pr_free_sm(pr) := StationAvailable

end

Event UpdateLoc_RetRes (ordinary) \(\triangleq\)
extends UpdateLoc_RetRes
  any
  pr
  l
  l0
  where
    isin_Returning: pr_free_sm(pr) = Returning
    grd0: l0 = rLocation(pr)
    grd1: l0 \notin ran(s_location))
    grd2: l \in LOCATION \setminus \{l0\}
  then
    act1: rLocation(pr) := l
\end{verbatim}
end

END
Appendix C

Event-B Model of the Travel Agency Workflow

In this chapter we present the complete version of the travel agency booking system presented in Chapter 7.

C.1 Static Part of the Model: Contexts

C.1.1 Context: C0

CONTEXT C0
SETS TRIP
END

C.1.2 Context: C1

CONTEXT C1
EXTENDS C0
SETS FLIGHT HOTEL CAR
END
Appendix C Event-B Model of the Travel Agency Workflow

C.2 Model Refinements

C.2.1 Abstract Level: M0

MACHINE M0
SEES C0
VARIABLES
StartTrip
TripItinerary
Quit

INVARIANTS
inv_StartTrip_type: \text{StartTrip} \subseteq \text{TRIP}
inv_FlightItinerary_f_seq: \text{TripItinerary} \subseteq \text{StartTrip}
inv_Quit_seq: \text{Quit} \subseteq \text{StartTrip}
inv_Interrupt: \text{partition}((\text{TripItinerary} \cup \text{Quit}), \text{TripItinerary}, \text{Quit})

EVENTS
Initialisation
begin
act_StartTrip: \text{StartTrip} := \emptyset
act_TripItinerary: \text{TripItinerary} := \emptyset
act_Quit: \text{Quit} := \emptyset
end
Event StartTrip (ordinary) \Rightarrow
any
\begin{align*}
t
\text{where} \quad \text{grd_self}: & \quad t \notin \text{StartTrip} \\
\text{then} \quad \text{act}: & \quad \text{StartTrip} := \text{StartTrip} \cup \{t\}
\end{align*}

Event TripItinerary (ordinary) \Rightarrow
any
\begin{align*}
t
\text{where} \quad \text{grd_seq}\_f: & \quad t \in \text{StartTrip} \\
\text{grd_self}: & \quad t \notin \text{TripItinerary} \\
\text{grd_interrupt}: & \quad t \notin \text{Quit} \\
\text{then} \quad \text{act}: & \quad \text{TripItinerary} := \text{TripItinerary} \cup \{t\}
\end{align*}
end

Event Quit (ordinary) \[\triangleq\]
any
  t
where
  \text{grd}\_seq: t \in \text{StartTrip}
  \text{grd}\_self: t \notin \text{Quit}
  \text{grd}\_interrupt: t \notin \text{TripItinerary}
then
  \text{act}: \text{Quit} := \text{Quit} \cup \{t\}
end

END

C.2.2 First Refinement: M1

MACHINE M1
REFINES M0
SEES C0

VARIABLES
  StartTrip
  FlightItinerary
  CarItinerary
  HotelItinerary
  Checkout
  Quit

INVARIANTS
  \text{inv}\_StartTrip\_type: \text{StartTrip} \subseteq \text{TRIP}
  \text{inv}\_FlightItinerary\_f\_seq: \text{FlightItinerary} \subseteq \text{StartTrip}
  \text{inv}\_CarItinerary\_f\_seq: \text{CarItinerary} \subseteq \text{StartTrip}
  \text{inv}\_HotelItinerary\_f\_seq: \text{HotelItinerary} \subseteq \text{StartTrip}
  \text{inv}\_Checkout\_f\_seq: \text{Checkout} \subseteq (\text{FlightItinerary} \cap \text{CarItinerary} \cap \text{HotelItinerary})
  \text{inv}\_Quit\_seq: \text{Quit} \subseteq \text{StartTrip}
  \text{inv}\_Checkout\_glu: \text{Checkout} = \text{TripItinerary}
  \text{inv}\_Interrupt: \text{partition}((\text{Checkout} \cup \text{Quit}), \text{Checkout}, \text{Quit})

EVENTS

Initialisation
begin
  \text{act}\_StartTrip: \text{StartTrip} := \emptyset
act_FlightItinerary: FlightItinerary := ∅
act_CarItinerary: CarItinerary := ∅
act_HotelItinerary: HotelItinerary := ∅
act_Checkout: Checkout := ∅
act_Quit: Quit := ∅
end

Event StartTrip (ordinary) ≡
refines StartTrip
  any
twhere
  grd_self: t /∈ StartTrip
then
  act: StartTrip := StartTrip ∪ {t}
end

Event FlightItinerary (ordinary) ≡
  any
twhere
  grd_seq_f: t ∈ StartTrip
  grd_self: t /∈ FlightItinerary
  grd_interrupt: t /∈ Quit
then
  act: FlightItinerary := FlightItinerary ∪ {t}
end

Event CarItinerary (ordinary) ≡
  any
twhere
  grd_seq_f: t ∈ StartTrip
  grd_self: t /∈ CarItinerary
  grd_interrupt: t /∈ Quit
then
  act: CarItinerary := CarItinerary ∪ {t}
end

Event Checkout (ordinary) ≡
refines TripItinerary
  any
twhere
Appendix C Event-B Model of the Travel Agency Workflow

grd\_interrupt: t \notin Quit
grd\_seq.f: (t \in FlightItinerary \land t \in CarItinerary \land t \in HotelItinerary)
grd\_self: t \notin Checkout

then
act: Checkout := Checkout \cup \{t\}
end

Event HotelItinerary (ordinary) \equiv

any

where

grd\_seq.f: t \in StartTrip
grd\_self: t \notin HotelItinerary
grd\_interrupt: t \notin Quit

then
act: HotelItinerary := HotelItinerary \cup \{t\}
end

Event Quit (ordinary) \equiv

refines Quit

any

where

grd\_seq: t \in StartTrip
grd\_self: t \notin Quit
grd\_interrupt: t \notin Checkout

then
act: Quit := Quit \cup \{t\}
end

END

C.2.3 Second Refinement: M2

MACHINE M2
REFINES M1
SEES C1
VARIABLES
StartTrip
FlightBooking
CompFlightItin
CarBooking
CompCarItin
HotelBooking
CompHotelItin
Checkout
Quit

**INvariants**

- \text{inv\_StartTrip\_type}: \text{StartTrip} \subseteq \text{TRIP}
- \text{inv\_FlightBooking\_type}: \text{FlightBooking} \subseteq \text{TRIP} \times \text{FLIGHT}
- \text{inv\_CarBooking\_type}: \text{CarBooking} \subseteq \text{TRIP} \times \text{CAR}
- \text{inv\_HotelBooking\_type}: \text{HotelBooking} \subseteq \text{TRIP} \times \text{HOTEL}
- \text{inv\_FlightBooking\_seq}: \text{dom(FlightBooking)} \subseteq \text{StartTrip}
- \text{inv\_FlightBooking\_rep}: \forall t \in \text{TRIP} \Rightarrow \text{FlightBooking}[[t]] \subseteq \text{FLIGHT}
- \text{inv\_CompFlightItin\_seq}: \text{CompFlightItin} \subseteq \text{StartTrip}
- \text{inv\_CompFlightItin\_glu}: \text{CompFlightItin} = \text{FlightItinerary}
- \text{inv\_CarBooking\_seq}: \text{dom(CarBooking)} \subseteq \text{StartTrip}
- \text{inv\_CarBooking\_rep}: \forall t \in \text{TRIP} \Rightarrow \text{CarBooking}[[t]] \subseteq \text{CAR}
- \text{inv\_CompCarItin\_seq}: \text{CompCarItin} \subseteq \text{StartTrip}
- \text{inv\_CompCarItin\_glu}: \text{CompCarItin} = \text{CarItinerary}
- \text{inv\_HotelBooking\_seq}: \text{dom(HotelBooking)} \subseteq \text{StartTrip}
- \text{inv\_HotelBooking\_rep}: \forall t \in \text{TRIP} \Rightarrow \text{HotelBooking}[[t]] \subseteq \text{HOTEL}
- \text{inv\_CompHotelItine\_seq}: \text{CompHotelItine} \subseteq \text{StartTrip}
- \text{inv\_CompHotelItine\_glu}: \text{CompHotelItine} = \text{HotelItinerary}
- \text{inv\_Checkout\_f\_seq}: \text{Checkout} \subseteq (\text{CompFlightItine} \cap \text{CompCarItine} \cap \text{CompHotelItine})

- \text{inv\_Quit\_seq}: \text{Quit} \subseteq \text{StartTrip}
- \text{inv\_interrupt}: \text{partition((Checkout} \cup \text{Quit), Checkout, Quit)}

**Events**

**Initialisation**

\text{\textbf{begin}}
\begin{align*}
\text{act\_StartTrip}: \text{StartTrip} & := \emptyset \\
\text{act\_FlightBooking}: \text{FlightBooking} & := \emptyset \\
\text{act\_CompFlightItin}: \text{CompFlightItin} & := \emptyset \\
\text{act\_CarBooking}: \text{CarBooking} & := \emptyset \\
\text{act\_CompCarItin}: \text{CompCarItin} & := \emptyset \\
\text{act\_HotelBooking}: \text{HotelBooking} & := \emptyset \\
\text{act\_CompHotelItine}: \text{CompHotelItine} & := \emptyset \\
\text{act\_Checkout}: \text{Checkout} & := \emptyset \\
\text{act\_Quit}: \text{Quit} & := \emptyset 
\end{align*}
\text{\textbf{end}}
**Event** FlightBooking \(\langle\text{ordinary}\rangle\) \(\triangleq\)

\[
\text{any} \\
t \\
f \\
\text{where} \\
grd\_seq.: \ t \in \text{StartTrip} \\
grd\_self: \ t \mapsto f \notin \text{FlightBooking} \\
grd0\_par: \ t \notin \text{CompFlightItin} \\
grd\_interrupt: \ t \notin \text{Quit} \\
\text{then} \\
\text{act}: \ \text{FlightBooking} := \text{FlightBooking} \cup \{t \mapsto f\} \\
\]

end

**Event** CarBooking \(\langle\text{ordinary}\rangle\) \(\triangleq\)

\[
\text{any} \\
t \\
c \\
\text{where} \\
grd\_seq.: \ t \in \text{StartTrip} \\
grd\_self: \ t \mapsto c \notin \text{CarBooking} \\
grd0\_par: \ t \notin \text{CompCarItin} \\
grd\_interrupt: \ t \notin \text{Quit} \\
\text{then} \\
\text{act}: \ \text{CarBooking} := \text{CarBooking} \cup \{t \mapsto c\} \\
\]

end

**Event** HotelBooking \(\langle\text{ordinary}\rangle\) \(\triangleq\)

\[
\text{any} \\
t \\
h \\
\text{where} \\
grd\_seq.: \ t \in \text{StartTrip} \\
grd\_self: \ t \mapsto h \notin \text{HotelBooking} \\
grd0\_par: \ t \notin \text{CompHotelItin} \\
grd\_interrupt: \ t \notin \text{Quit} \\
\text{then} \\
\text{act}: \ \text{HotelBooking} := \text{HotelBooking} \cup \{t \mapsto h\} \\
\]

end

**Event** StartTrip \(\langle\text{ordinary}\rangle\) \(\triangleq\)

\[\text{refines } \text{StartTrip}\]

\[
\text{any} \\
t \\
\text{where} \\
\]

\begin{verbatim}
  grd_self: \ t \notin \text{StartTrip} then
  act: \text{StartTrip} := \text{StartTrip} \cup \{t\}
end

Event CompFlightItin (ordinary) \cong
refines FlightItinerary
any
  t
where
  grd_seq: \ t \in \text{StartTrip}
  grd_self: \ t \notin \text{CompFlightItin}
  grd_interrupt: \ t \notin \text{Quit}
then
  act: \text{CompFlightItin} := \text{CompFlightItin} \cup \{t\}
end

Event CompCarItin (ordinary) \cong
refines CarItinerary
any
  t
where
  grd_interrupt: \ t \notin \text{Quit}
  grd_seq: \ t \in \text{StartTrip}
  grd_self: \ t \notin \text{CompCarItin}
then
  act: \text{CompCarItin} := \text{CompCarItin} \cup \{t\}
end

Event CompHotelItin (ordinary) \cong
refines HotelItinerary
any
  t
where
  grd_seq: \ t \in \text{StartTrip}
  grd_self: \ t \notin \text{CompHotelItin}
  grd_interrupt: \ t \notin \text{Quit}
then
  act: \text{CompHotelItin} := \text{CompHotelItin} \cup \{t\}
end

Event Checkout (ordinary) \cong
refines Checkout
any
\end{verbatim}
t

where
  grd_seq.f: (t ∈ CompFlightItin ∧ t ∈ CompCarItin ∧ t ∈ CompHotelItin)
  grd_self: t /∈ Checkout
  grd_interrupt: t /∈ Quit

then
  act: Checkout := Checkout ∪ \{t\}
end

Event Quit (ordinary) ≡
refines Quit
  any
    t
  where
    grd_seq: t ∈ StartTrip
    grd_self: t /∈ Quit
    grd_interrupt: t /∈ Checkout

then
  act: Quit := Quit ∪ \{t\}
end

END

C.2.4 Third Refinement: M3

MACHINE M3
REFINES M2
SEES C1
VARIABLES
  StartTrip
  SelectFlight
  BookFlight
  RemoveFlight
  CompFlightItin
  SelectCar
  BookCar
  RemoveCar
  CompCarItin
  SelectHotel
  BookHotel
  RemoveHotel
CompHotelItin
Checkout
Quit
flights
cars
hotels manually

INVARIANTS

inv_StartTrip_type: StartTrip ⊆ TRIP
inv_SelectFlight_type: SelectFlight ⊆ TRIP × FLIGHT
inv_SelectFlight_seq: dom(SelectFlight) ⊆ StartTrip
inv_SelectFlight_glu: SelectFlight = FlightBooking
inv_BookFlight_seq: BookFlight ⊆ SelectFlight
inv_RemoveFlight_seq: RemoveFlight ⊆ SelectFlight
inv_CompFlightItin_seq: CompFlightItin ⊆ StartTrip
inv_SelectCar_type: SelectCar ⊆ TRIP × CAR
inv_SelectCar_seq: dom(SelectCar) ⊆ StartTrip
inv_SelectCar_glu: SelectCar = CarBooking
inv_BookCar_seq: BookCar ⊆ SelectCar
inv_RemoveCar_seq: RemoveCar ⊆ SelectCar
inv_CompCarItin_seq: CompCarItin ⊆ StartTrip
inv_SelectHotel_type: SelectHotel ⊆ TRIP × HOTEL
inv_SelectHotel_seq: dom(SelectHotel) ⊆ StartTrip
inv_SelectHotel_glu: SelectHotel = HotelBooking
inv_BookHotel_seq: BookHotel ⊆ SelectHotel
inv_RemoveHotel_seq: RemoveHotel ⊆ SelectHotel
inv_CompHotelItin_seq: CompHotelItin ⊆ StartTrip
inv_CompFlightItin_par: \∀ t ∈ CompFlightItin \Rightarrow SelectFlight[t] = (BookFlight[t] ∪ RemoveFlight[t])
inv_CompCarItin_par: \∀ t ∈ CompCarItin \Rightarrow SelectCar[t] = (BookCar[t] ∪ RemoveCar[t])
inv_CompHotelItin_par: \∀ t ∈ CompHotelItin \Rightarrow SelectHotel[t] = (BookHotel[t] ∪ RemoveHotel[t])
inv_Checkout_f_seq: Checkout ⊆ (CompFlightItin ∩ CompCarItin ∩ CompHotelItin)

inv1_xorf: partition((BookFlight ∪ RemoveFlight), BookFlight, RemoveFlight)
inv2_xorf: partition((BookCar ∪ RemoveCar), BookCar, RemoveCar)
inv3_xorf: partition((BookHotel ∪ RemoveHotel), BookHotel, RemoveHotel)
inv.Quit_seq: Quit ⊆ StartTrip
Appendix C Event-B Model of the Travel Agency Workflow

inv\_interrupt:: partition((Checkout ∪ Quit), Checkout, Quit)
inv\_flights:: flights ∈ TRIP ⇔ FLIGHT
inv\_cars:: cars ∈ TRIP ⇔ CAR
inv\_hotels:: hotels ∈ TRIP ⇔ HOTEL
inv\_flights\_select:: flights ⊆ SelectFlight
inv\_hotels\_select:: hotels ⊆ SelectHotel
inv\_cars\_select:: cars ⊆ SelectCar
inv\_flights\_remove:: RemoveFlight ∩ flights = ∅
inv\_cars\_remove:: RemoveCar ∩ cars = ∅
inv\_hotels\_remove:: RemoveHotel ∩ hotels = ∅

EVENTS

Initialisation

begin
  act\_StartTrip: StartTrip := ∅
  act\_SelectFlight: SelectFlight := ∅
  act\_BookFlight: BookFlight := ∅
  act\_RemoveFlight: RemoveFlight := ∅
  act\_CompFlightItin: CompFlightItin := ∅
  act\_SelectCar: SelectCar := ∅
  act\_BookCar: BookCar := ∅
  act\_RemoveCar: RemoveCar := ∅
  act\_CompCarItin: CompCarItin := ∅
  act\_SelectHotel: SelectHotel := ∅
  act\_BookHotel: BookHotel := ∅
  act\_RemoveHotel: RemoveHotel := ∅
  act\_CompHotelItine: CompHotelItin := ∅
  act\_Checkout: Checkout := ∅
  act\_flights: flights := ∅
  act\_hotels: hotels := ∅
  act\_cars: cars := ∅
  act\_Quit: Quit := ∅
end

Event StartTrip ⟨ordinary⟩  ≜
refines StartTrip

any
  t
where
  grd\_self:: t ∉ StartTrip
then
  act: StartTrip := StartTrip ∪ {t}
Event SelectFlight \langle\text{ordinary}\rangle \trianglerighteq 
\text{refines} \text{ FlightBooking}

\text{any}
\hspace{1cm} t
\hspace{1cm} f

\text{where}
\hspace{1cm} \text{grd}\_\text{seq}:: t \in \text{StartTrip}
\hspace{1cm} \text{grd}\_\text{self}:: t \mapsto f \notin \text{SelectFlight}
\hspace{1cm} \text{grd}\_0\text{par}:: t \notin \text{CompFlightItin}
\hspace{1cm} \text{grd}\_\text{interrupt}:: t \notin \text{Quit}
\hspace{1cm} \text{grd}\_1:: f \notin \text{flights}\{t\}

\text{then}
\hspace{1cm} \text{act}: \text{SelectFlight} := \text{SelectFlight} \cup \{t \mapsto f\}
\hspace{1cm} \text{act1}: \text{flights} := \text{flights} \cup \{t \mapsto f\}

end

Event BookFlight \langle\text{ordinary}\rangle \trianglerighteq 

\text{any}
\hspace{1cm} t
\hspace{1cm} f

\text{where}
\hspace{1cm} \text{grd}\_\text{seq}:: t \mapsto f \in \text{SelectFlight}
\hspace{1cm} \text{grd}\_\text{self}:: t \mapsto f \notin \text{BookFlight}
\hspace{1cm} \text{grd}\_1\text{xor}:: t \mapsto f \notin \text{RemoveFlight}
\hspace{1cm} \text{grd}\_\text{interrupt}:: t \notin \text{Quit}

\text{then}
\hspace{1cm} \text{act}: \text{BookFlight} := \text{BookFlight} \cup \{t \mapsto f\}

end

Event RemoveFlight \langle\text{ordinary}\rangle \trianglerighteq 

\text{any}
\hspace{1cm} t
\hspace{1cm} f

\text{where}
\hspace{1cm} \text{grd}\_\text{seq}:: t \mapsto f \in \text{SelectFlight}
\hspace{1cm} \text{grd}\_\text{self}:: t \mapsto f \notin \text{RemoveFlight}
\hspace{1cm} \text{grd}\_1\text{xor}:: t \mapsto f \notin \text{BookFlight}
\hspace{1cm} \text{grd}\_\text{interrupt}:: t \notin \text{Quit}

\text{then}
\hspace{1cm} \text{act}: \text{RemoveFlight} := \text{RemoveFlight} \cup \{t \mapsto f\}
\hspace{1cm} \text{act1}: \text{flights} := \text{flights} \setminus \{t \mapsto f\}

end
Appendix C Event-B Model of the Travel Agency Workflow

**Event** CompFlightItin (ordinary) ≡

**refines** CompFlightItin

\[
\text{any}
\begin{align*}
t & \text{where} \\
\text{grd_seq: } & t \in \text{StartTrip} \\
\text{grd_self: } & t \notin \text{CompFlightItin} \\
\text{grd_par: } & \text{SelectFlight}[[t]] = \text{BookFlight}[[t]] \cup \text{RemoveFlight}[[t]] \\
\text{grd_interrupt: } & t \notin \text{Quit}
\end{align*}
\]

then

\[
\text{act: } \text{CompFlightItin} := \text{CompFlightItin} \cup \{t\}
\]

end

**Event** SelectCar (ordinary) ≡

**refines** CarBooking

\[
\text{any}
\begin{align*}
t & \text{c where} \\
\text{grd_seq: } & t \in \text{StartTrip} \\
\text{grd_self: } & t \mapsto c \notin \text{SelectCar} \\
\text{grd0_par: } & t \notin \text{CompCarItin} \\
\text{grd_interrupt: } & t \notin \text{Quit} \\
\text{grd1: } & c \notin \text{cars}[[t]]
\end{align*}
\]

then

\[
\text{act: } \text{SelectCar} := \text{SelectCar} \cup \{t \mapsto c\} \\
\text{act1: } \text{cars} := \text{cars} \cup \{t \mapsto c\}
\]

end

**Event** BookCar (ordinary) ≡

\[
\text{any}
\begin{align*}
t & \text{c where} \\
\text{grd_seq: } & t \mapsto c \in \text{SelectCar} \\
\text{grd_self: } & t \mapsto c \notin \text{BookCar} \\
\text{grd0_xor: } & t \mapsto c \notin \text{RemoveCar} \\
\text{grd_interrupt: } & t \notin \text{Quit}
\end{align*}
\]

then

\[
\text{act: } \text{BookCar} := \text{BookCar} \cup \{t \mapsto c\}
\]

end

**Event** RemoveCar (ordinary) ≡
any
tc

where
  grd_seq: \( t \mapsto c \in \text{SelectCar} \)
  grd_self: \( t \mapsto c \notin \text{RemoveCar} \)
  grd1_xor: \( t \mapsto c \notin \text{BookCar} \)
  grd_interrupt: \( t \notin \text{Quit} \)

then
  act: \text{RemoveCar} := \text{RemoveCar} \cup \{ t \mapsto c \}
  act1: \text{cars} := \text{cars} \setminus \{ t \mapsto c \}

end
Event CompCarItin (ordinary) \( \cong \)
refines CompCarItin

any
t

where
  grd_seq: \( t \in \text{StartTrip} \)
  grd_self: \( t \notin \text{CompCarItin} \)
  grd_par: \( \text{SelectCar}[[t]] = \text{BookCar}[[t]] \cup \text{RemoveCar}[[t]] \)
  grd_interrupt: \( t \notin \text{Quit} \)

then
  act: \text{CompCarItin} := \text{CompCarItin} \cup \{ t \}

end
Event SelectHotel (ordinary) \( \cong \)
refines HotelBooking

any
th

where
  grd_seq: \( t \in \text{StartTrip} \)
  grd_self: \( t \mapsto h \notin \text{SelectHotel} \)
  grd0_par: \( t \notin \text{CompHotelItin} \)
  grd_interrupt: \( t \notin \text{Quit} \)
  grd1: \( h \notin \text{hotels}[[t]] \)

then
  act: \text{SelectHotel} := \text{SelectHotel} \cup \{ t \mapsto h \}
  act1: \text{hotels} := \text{hotels} \cup \{ t \mapsto h \}

end
Event BookHotel (ordinary) \( \cong \)

any
where
$\text{grd}_\text{seq}$: $t \mapsto h \in \text{SelectHotel}$
$\text{grd}_\text{self}$: $t \mapsto h \notin \text{BookHotel}$
$\text{grd1}_\text{xor}$: $t \mapsto h \notin \text{RemoveHotel}$
$\text{grd}_\text{interrupt}$: $t \notin \text{Quit}$

then
$\text{act}$: $\text{BookHotel} := \text{BookHotel} \cup \{t \mapsto h\}$

end

Event RemoveHotel (ordinary) $\triangleq$
aany
t
h
where
$\text{grd}_\text{seq}$: $t \mapsto h \in \text{SelectHotel}$
$\text{grd}_\text{self}$: $t \mapsto h \notin \text{RemoveHotel}$
$\text{grd1}_\text{xor}$: $t \mapsto h \notin \text{BookHotel}$
$\text{grd}_\text{interrupt}$: $t \notin \text{Quit}$

then
$\text{act}$: $\text{RemoveHotel} := \text{RemoveHotel} \cup \{t \mapsto h\}$
$\text{act1}$: $\text{hotels} := \text{hotels} \setminus \{t \mapsto h\}$

end

Event CompHotelItin (ordinary) $\triangleq$
refines CompHotelItin

aany
t
where
$\text{grd}_\text{seq}$: $t \in \text{StartTrip}$
$\text{grd}_\text{self}$: $t \notin \text{CompHotelItin}$
$\text{grd}_\text{par}$: $\text{SelectHotel}[\{t\}] = \text{BookHotel}[\{t\}] \cup \text{RemoveHotel}[\{t\}]$
$\text{grd}_\text{interrupt}$: $t \notin \text{Quit}$

then
$\text{act}$: $\text{CompHotelItin} := \text{CompHotelItin} \cup \{t\}$

end

Event Checkout (ordinary) $\triangleq$
refines Checkout
aany
t
where
$\text{grd}_\text{seq} f$: $(t \in \text{CompFlightItin} \land t \in \text{CompCarItin} \land t \in \text{CompHotelItin})$
Appendix C Event-B Model of the Travel Agency Workflow

\[
\text{grd} \_\text{self}: \ t \not\in \text{Checkout} \\
\text{grd} \_\text{interrupt}: \ t \not\in \text{Quit}
\]

\text{then}

\text{act}: \ \text{Checkout} := \text{Checkout} \cup \{t\}

\text{end}

\text{Event} \ \text{Quit} \ \text{(ordinary)} \Rightarrow \text{refines} \ \text{Quit}

\text{any}

\text{t}

\text{where}

\text{grd} \_\text{seq}: \ t \in \text{StartTrip} \\
\text{grd} \_\text{self}: \ t \not\in \text{Quit} \\
\text{grd} \_\text{interrupt}: \ t \not\in \text{Checkout}

\text{then}

\text{act}: \ \text{Quit} := \text{Quit} \cup \{t\} \\
\text{act1}: \ \text{flights} := \{t\} \triangleleft \text{flights} \\
\text{act2}: \ \text{hotels} := \{t\} \triangleleft \text{hotels} \\
\text{act3}: \ \text{cars} := \{t\} \triangleleft \text{cars}

\text{end}

\text{END}

C.2.5 Fourth Refinement: M4

\text{MACHINE} \ M4 \\
\text{REFINES} \ M3 \\
\text{SEES} \ C1 \\
\text{VARIABLES}

\text{StartTrip} \\
\text{SelectFlight} \\
\text{BookFlight} \\
\text{RemoveFlight} \\
\text{CompFlightItin} \\
\text{SelectCar} \\
\text{BookCar} \\
\text{RemoveCar} \\
\text{CompCarItin} \\
\text{SelectHotel} \\
\text{BookHotel} \\
\text{RemoveHotel}
CompHotelItin
Checkout
Quit
flights
cars
hotels manually
time manually
deadline manually
tFlight
tCar
tHotel time selected

**INVARIANTS**
\[ \text{inv}\_\text{time} : \text{time} \in \mathbb{N} \]
\[ \text{inv}\_\text{deadline} : \text{deadline} \in \mathbb{N} \]
\[ \text{inv}\_\text{tFlight} : \text{tFlight} \in \text{flights} \to \mathbb{N} \]
\[ \text{inv}\_\text{tCar} : \text{tCar} \in \text{cars} \to \mathbb{N} \]
\[ \text{inv}\_\text{tHotel} : \text{tHotel} \in \text{hotels} \to \mathbb{N} \]

**EVENTS**

**Initialisation** (extended)
\[ \text{act}\_\text{StartTrip} : \text{StartTrip} := \emptyset \]
\[ \text{act}\_\text{SelectFlight} : \text{SelectFlight} := \emptyset \]
\[ \text{act}\_\text{BookFlight} : \text{BookFlight} := \emptyset \]
\[ \text{act}\_\text{RemoveFlight} : \text{RemoveFlight} := \emptyset \]
\[ \text{act}\_\text{CompFlightItin} : \text{CompFlightItin} := \emptyset \]
\[ \text{act}\_\text{SelectCar} : \text{SelectCar} := \emptyset \]
\[ \text{act}\_\text{BookCar} : \text{BookCar} := \emptyset \]
\[ \text{act}\_\text{RemoveCar} : \text{RemoveCar} := \emptyset \]
\[ \text{act}\_\text{CompCarItin} : \text{CompCarItin} := \emptyset \]
\[ \text{act}\_\text{SelectHotel} : \text{SelectHotel} := \emptyset \]
\[ \text{act}\_\text{BookHotel} : \text{BookHotel} := \emptyset \]
\[ \text{act}\_\text{RemoveHotel} : \text{RemoveHotel} := \emptyset \]
\[ \text{act}\_\text{CompHotelItine} : \text{CompHotelItine} := \emptyset \]
\[ \text{act}\_\text{Checkout} : \text{Checkout} := \emptyset \]
\[ \text{act}\_\text{flights} : \text{flights} := \emptyset \]
\[ \text{act}\_\text{hotels} : \text{hotels} := \emptyset \]
\[ \text{act}\_\text{cars} : \text{cars} := \emptyset \]
\[ \text{act}\_\text{Quit} : \text{Quit} := \emptyset \]
\[ \text{act}\_\text{time} : \text{time} := 0 \]
act\_deadline: \( \text{deadline} \in \mathbb{N}_1 \)
act\_tFlight: \( t_{\text{Flight}} := \emptyset \)
act\_tCars: \( t_{\text{Car}} := \emptyset \)
act\_tHotels: \( t_{\text{Hotel}} := \emptyset \)

\textbf{Event} StartTrip \langle \text{ordinary} \rangle \triangleq
\textbf{extends} StartTrip
\begin{align*}
\text{any} & \\
\text{where} & \\
\text{grd\_self: } & t \notin \text{StartTrip} \\
\text{then} & \\
\text{act: } & \text{StartTrip} := \text{StartTrip} \cup \{t\} \\
\text{end} \\
\textbf{Event} & \text{SelectFlight} \langle \text{ordinary} \rangle \triangleq
\textbf{extends} \text{SelectFlight}
\begin{align*}
\text{any} & \\
\text{t} & \\
\text{where} & \\
\text{grd\_seq.: } & t \in \text{StartTrip} \\
\text{grd\_self: } & t \mapsto f \notin \text{SelectFlight} \\
\text{grd0\_par: } & t \notin \text{CompFlightItin} \\
\text{grd\_interrupt: } & t \notin \text{Quit} \\
\text{grd1: } & f \notin \text{flights}[\{t\}] \\
\text{then} & \\
\text{act: } & \text{SelectFlight} := \text{SelectFlight} \cup \{t \mapsto f\} \\
\text{act1: } & \text{flights} := \text{flights} \cup \{t \mapsto f\} \\
\text{act2: } & \text{tFlight}(t \mapsto f) := \text{time} \\
\text{end} \\
\textbf{Event} & \text{BookFlight} \langle \text{ordinary} \rangle \triangleq
\textbf{extends} \text{BookFlight}
\begin{align*}
\text{any} & \\
\text{t} & \\
\text{f} & \\
\text{where} & \\
\text{grd\_seq.: } & t \mapsto f \in \text{SelectFlight} \\
\text{grd\_self: } & t \mapsto f \notin \text{BookFlight} \\
\text{grd1\_xor: } & t \mapsto f \notin \text{RemoveFlight} \\
\text{grd\_interrupt: } & t \notin \text{Quit} \\
\end{align*}
Appendix C Event-B Model of the Travel Agency Workflow

\[ \text{grd1: } (t \mapsto f) \in \text{flights} \]
\[ \text{grd2: } \text{time} \leq (t\text{Flight}(t \mapsto f) + \text{deadline}) \]
\[ \text{then} \]
\[ \text{act: } \text{BookFlight} := \text{BookFlight} \cup \{t \mapsto f\} \]
\[ \text{end} \]

**Event** RemoveFlight \(\langle \text{ordinary} \rangle \)

**extends** RemoveFlight

any
\[ t \]
\[ f \]
where
\[ \text{grd\_seq.: } t \mapsto f \in \text{SelectFlight} \]
\[ \text{grd\_self: } t \mapsto f \notin \text{RemoveFlight} \]
\[ \text{grd1\_xor: } t \mapsto f \notin \text{BookFlight} \]
\[ \text{grd\_interrupt: } t \notin \text{Quit} \]
\[ \text{grd2: } (t \mapsto f) \in \text{flights} \]
\[ \text{then} \]
\[ \text{act: } \text{RemoveFlight} := \text{RemoveFlight} \cup \{t \mapsto f\} \]
\[ \text{act1: } \text{flights} := \text{flights} \setminus \{t \mapsto f\} \]
\[ \text{act2: } t\text{Flight} := \{t \mapsto f\} \triangleleft t\text{Flight} \]
\[ \text{end} \]

**Event** CompFlightItin \(\langle \text{ordinary} \rangle \)

**extends** CompFlightItin

any
\[ t \]
where
\[ \text{grd\_seq.: } t \in \text{StartTrip} \]
\[ \text{grd\_self: } t \notin \text{CompFlightItin} \]
\[ \text{grd\_par: } \text{SelectFlight}[[\{t\}]] = \text{BookFlight}[[\{t\}]] \cup \text{RemoveFlight}[[\{t\}]] \]
\[ \text{grd\_interrupt: } t \notin \text{Quit} \]
\[ \text{then} \]
\[ \text{act: } \text{CompFlightItin} := \text{CompFlightItin} \cup \{t\} \]
\[ \text{end} \]

**Event** SelectCar \(\langle \text{ordinary} \rangle \)

**extends** SelectCar

any
\[ t \]
\[ c \]
where
\[ \text{grd\_seq.: } t \in \text{StartTrip} \]
\[\text{Appendix C Event-B Model of the Travel Agency Workflow}\]

\[\begin{array}{l}
\text{grd\_self: } t \mapsto c \notin \text{SelectCar} \\
\text{grd0\_par: } t \notin \text{CompCarItin} \\
\text{grd\_interrupt: } t \notin \text{Quit} \\
\text{grd1: } c \notin \text{cars}[\{t\}] \\
\text{then} \\
\text{act: } \text{SelectCar} := \text{SelectCar} \cup \{t \mapsto c\} \\
\text{act1: } \text{cars} := \text{cars} \cup \{t \mapsto c\} \\
\text{act2: } t\text{Car}(t \mapsto c) := \text{time} \\
\end{array}\]

\[\text{Event BookCar \langle ordinary \rangle } \equiv \]

\[\text{extends BookCar} \]

\[\begin{array}{l}
\text{any} \\
\text{t} \\
\text{c} \\
\text{where} \\
\text{grd\_seq.: } t \mapsto c \in \text{SelectCar} \\
\text{grd\_self: } t \mapsto c \notin \text{BookCar} \\
\text{grd1\_xor: } t \mapsto c \notin \text{RemoveCar} \\
\text{grd\_interrupt: } t \notin \text{Quit} \\
\text{grd1: } (t \mapsto c) \in \text{cars} \\
\text{grd2: } \text{time} \leq (t\text{Car}(t \mapsto c) + \text{deadline}) \\
\text{then} \\
\text{act: } \text{BookCar} := \text{BookCar} \cup \{t \mapsto c\} \\
\end{array}\]

\[\text{Event RemoveCar \langle ordinary \rangle } \equiv \]

\[\text{extends RemoveCar} \]

\[\begin{array}{l}
\text{any} \\
\text{t} \\
\text{c} \\
\text{where} \\
\text{grd\_seq.: } t \mapsto c \in \text{SelectCar} \\
\text{grd\_self: } t \mapsto c \notin \text{RemoveCar} \\
\text{grd1\_xor: } t \mapsto c \notin \text{BookCar} \\
\text{grd\_interrupt: } t \notin \text{Quit} \\
\text{grd1: } (t \mapsto c) \in \text{cars} \\
\text{then} \\
\text{act: } \text{RemoveCar} := \text{RemoveCar} \cup \{t \mapsto c\} \\
\text{act1: } \text{cars} := \text{cars} \setminus \{t \mapsto c\} \\
\text{act2: } t\text{Car} := \{t \mapsto c\} \triangleq t\text{Car} \\
\end{array}\]

\[\text{Event CompCarItin \langle ordinary \rangle } \equiv \]
extends CompCarItin

any

  
where

  grd_seq: t ∈ StartTrip
  grd_self: t /∈ CompCarItin
  grd_par: SelectCar[{t}] = BookCar[{t}] ∪ RemoveCar[{t}]
  grd_interrupt: t /∈ Quit

then

  act: CompCarItin := CompCarItin ∪ {t}

end

Event SelectHotel ⟨ordinary⟩ ≜

extends SelectHotel

any

  t
  h

where

  grd_seq: t ∈ StartTrip
  grd_self: h /∈ SelectHotel
  grd0_par: t /∈ CompHotelItin
  grd Interrupt: t /∈ Quit
  gdi: h /∈ hotels[{t}]

then

  act: SelectHotel := SelectHotel ∪ {t ↦ h}
  act1: hotels := hotels ∪ {t ↦ h}
  act2: tHotel(t ↦ h) := time

end

Event BookHotel ⟨ordinary⟩ ≜

extends BookHotel

any

  t
  h

where

  grd_seq: t ↦ h ∈ SelectHotel
  grd_self: t ↦ h /∈ BookHotel
  grd1_xor: t ↦ h /∈ RemoveHotel
  grd_interrupt: t /∈ Quit
  gdi: (t ↦ h) ∈ hotels
  gdi2: time ≤ (tHotel(t ↦ h) + deadline)

then

  act: BookHotel := BookHotel ∪ {t ↦ h}
Event RemoveHotel \(\langle\text{ordinary}\rangle\) \(\equiv\)
extends RemoveHotel
any
\( t \)
\( h \)
where
\( \text{grd\_seq:} \ t \mapsto h \in \text{SelectHotel} \)
\( \text{grd\_self:} \ t \mapsto h \notin \text{RemoveHotel} \)
\( \text{grd\_xor:} \ t \mapsto h \notin \text{BookHotel} \)
\( \text{grd\_interrupt:} \ t \notin \text{Quit} \)
\( \text{grd2:} \ (t \mapsto h) \in \text{hotels} \)
then
act: \( \text{RemoveHotel} := \text{RemoveHotel} \cup \{t \mapsto h\} \)
act1: \( \text{hotels} := \text{hotels} \setminus \{t \mapsto h\} \)
act2: \( t_{Hotel} := \{t \mapsto h\} \triangleleft t_{Hotel} \)
end
Event CompHotelItin \(\langle\text{ordinary}\rangle\) \(\equiv\)
extends CompHotelItin
any
\( t \)
where
\( \text{grd\_seq:} \ t \in \text{StartTrip} \)
\( \text{grd\_self:} \ t \notin \text{CompHotelItin} \)
\( \text{grd\_par:} \ \text{SelectHotel}\{t\} = \text{BookHotel}\{t\} \cup \text{RemoveHotel}\{t\} \)
\( \text{grd\_interrupt:} \ t \notin \text{Quit} \)
then
act: \( \text{CompHotelItin} := \text{CompHotelItin} \cup \{t\} \)
end
Event Checkout \(\langle\text{ordinary}\rangle\) \(\equiv\)
extends Checkout
any
\( t \)
where
\( \text{grd\_seq:} \ (t \in \text{CompFlightItin} \land t \in \text{CompCarItin} \land t \in \text{CompHotelItin}) \)
\( \text{grd\_self:} \ t \notin \text{Checkout} \)
\( \text{grd\_interrupt:} \ t \notin \text{Quit} \)
then
act: \( \text{Checkout} := \text{Checkout} \cup \{t\} \)
end
**Event** Quit (ordinary) ≝
refines Quit
extends Quit
any
   t
   th
   tf
   tc

where
   grd_seq:  \( t \in \text{StartTrip} \)
   grd_self:  \( t \notin \text{Quit} \)
   grd_interrupt:  \( t \notin \text{Checkout} \)
   grd1:  \( th = \{t\} \triangleleft \text{hotels} \)
   grd2:  \( tf = \{t\} \triangleleft \text{flights} \)
   grd3:  \( tc = \{t\} \triangleleft \text{cars} \)

then
   act:  Quit := Quit \cup \{t\}
   act1:  flights := \{t\} \triangleleft \text{flights}
   act2:  hotels := \{t\} \triangleleft \text{hotels}
   act3:  cars := \{t\} \triangleleft \text{cars}
   act4:  tFlight := tf \triangleleft \text{tFlight}
   act5:  tHotel := th \triangleleft \text{tHotel}
   act6:  tCar := tc \triangleleft \text{tCar}

end

**Event** clock (ordinary) ≝
begin
   act:  time := time + 1
end

END
Appendix D

Functions Used in Formalising the ERS Translation Rules

In this chapter we present the auxiliary functions reused from Salehi Fathabadi (2012) and show how they are updated to include the newly added ERS extensions.

D.1 Predecessor and Successor Functions

In Figure D.1, the predecessor function is updated to include the par-replicator, which is treated similar to a loop by finding its predecessor due to the zero execution case in both combinators. The abstract flow is distinguished from a refined flow by having no solid lines, and in the case there is no predecessor, sequencing is not defined however the type of the leaf will be defined.

<table>
<thead>
<tr>
<th>Function</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{predecessor} (child, *par)</td>
<td>\texttt{where} (i &gt; 1) and (\text{child}_{i-1} \neq \text{loop/par-rep})</td>
</tr>
<tr>
<td>\texttt{predecessor} (child, *par, sw)</td>
<td>\texttt{where} (i &gt; 1) and (\text{child}_{i-1} = \text{loop/par-rep})</td>
</tr>
<tr>
<td>\texttt{“no predecessor”}</td>
<td>\texttt{where} (i = 1) and (\text{sw} = 0)</td>
</tr>
<tr>
<td>\texttt{“no predecessor”}</td>
<td>\texttt{where} (i = 1) and (\text{sw} = 1) and (\text{parentFlow(child)}) is an abstract flow</td>
</tr>
<tr>
<td>\texttt{predecessor} (parent(child), *par, sw)</td>
<td>\texttt{where} (i = 1) and (\text{sw} = 1) and (\text{parentFlow(child)}) is not (all/some/one) child</td>
</tr>
<tr>
<td>\texttt{predecessor} (parent(child), *par / p, sw)</td>
<td>\texttt{where} (i = 1) and (\text{sw} = 1) and (\text{parentFlow(child)}) is a (all/some/one) child</td>
</tr>
</tbody>
</table>

Figure D.1: Updated \texttt{predecessor} Function
In Figure D.2 the *successor* function is updated to include the *par-rep*, which is treated similar to a *loop*, whenever encountered find its *successor*. The *successor* function is used in the *par-replicator* and the *loop* translation rules, the current definition will work since both *cannot* be the last events in a flow, and since we introduced the restriction that the follow-on event of a *loop* and a *par-replicator* *cannot* be an exception handler (*interrupt*, *retry*).

<table>
<thead>
<tr>
<th>successor (child; child/cons-child, parnum: int)</th>
</tr>
</thead>
<tbody>
<tr>
<td>• (child_{i+1}, parnum) where (i &lt; n) and (child_{i+1} ≠ loop/par-rep/interrupt/retry)</td>
</tr>
<tr>
<td>• successor(child_{i+1}, parnum) where (i &lt; n) and (child_{i+1} = loop/par-rep)</td>
</tr>
</tbody>
</table>

Figure D.2: Updated *successor* Function

### D.2 Traversing Functions

The traversing functions are functions that help in traversing down the tree to get to the last refinement level. In all of these functions we do not consider the case of the *loop*, because the *loop* does not have a variable that records its execution, hence the restriction the *loop* cannot be the first child of a flow (Salehi Fathabadi, 2012). All these functions are updated to include the *par-replicator* and the exception handling combinators. In the case of the *par-replicator* the function is applied to the *par-replicator* child and its *successor* child, while in the case of an exception handling combinator, the function will be applied to both the *normal-child* and the *interrupting-child*.

The traversing functions include three functions, the *conjunction_of_leaves* function, which is presented in Figure D.3, the output of this function is the conjunction of the guards. The *disjunction_of_leaves* as shown in Figure D.4 will result in the disjunction of the guards. Finally the *union_of_leaves* function which will result in the union of the control variables of the leaf events, as presented in Figure D.5.

### D.3 Sequencing Functions

The sequencing of a leaf event depends on its predecessor. In Figure D.6, we have updated the *build_seq grd* function to include the exception handling combinators, where in the case of *retry* we only care about the *normal-child* whereas an *interrupt-combinator* takes into account both the *normal-child* and the *interrupting-child*. The *build_seq grd* depends on the function *seq grd* (Figures D.7 and D.8) which is still the same as described in Salehi Fathabadi (2012), as this function only checks the common parameters and the *all-replicator* parameter. The *build_seq inv* function is also updated in a similar way to
### Figure D.3: Updated *conjunction_of_leaves* Function

The *build_seq_grd* function, but it is not shown here because it is not used in our newly introduced translation rules presented in Chapter 8.

In Figure D.8, the *seq_grd* function (Figure D.8) needs to determine the possible common parent parameters \((p_1 \cdots p_i)\), which are the first in the parameter list, since any added new parameters will be added to the end of the parameter list. We also need the *all-replicator* parameter which affects the sequencing guards and invariants. In Figure D.7 from (Salehi Fathabadi, 2012) presents the definitions of \(X\), \(Y\), \(W\) and \(K\) which are used in defining the sequencing between two leaves.
### disjunction_of_leaves (ch: child, pnum: int)

**Output Operation:**

<table>
<thead>
<tr>
<th>f (leaf (leaf-name)) =</th>
<th>leaf-name</th>
<th>pnum = 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>f (leaf (leaf-name)) =</td>
<td>leaf-name ≠ ∅</td>
<td>pnum &gt; 0</td>
</tr>
</tbody>
</table>

**Pattern Matching:**

- $f (\text{constructor} (c_1, ..., c_n), \text{pnum}) = f (c_1, \text{pnum}) \lor ... \lor f (c_n, \text{pnum})$
  where constructor = and/or/xor

- $f (\text{exception-combinator} (c_{\text{normal-child}}, c_{\text{interrupting-child}}, \text{pnum}) = f (c_{\text{normal-child}}, \text{pnum}) \lor f (c_{\text{interrupting-child}}, \text{pnum})$
  where exception-combinator = interrupt/retry

- $f (\text{replicator}(c), \text{pnum}) = f (c, \text{pnum} + 1)$
  where replicator = all/some/one

- $f (\text{par-rep} (\text{par}, c), \text{pnum}) = f (c, \text{pnum} + 1) \lor f (\text{nextChild}, \text{pnum})$
  where nextChild = successor$(c, n)$

- $f (\text{1\textsuperscript{st} flow}, \text{pnum}) = f (\text{flow}_1, \text{pnum}) \lor ... \lor f (\text{flow}_n, \text{pnum})$

- $f (\text{flow} (\text{name}, \star \text{par}, 1), \text{pnum}) = f (\text{child}_1, \text{pnum})$
  where child$_1$ is the first child of the solid flow

- $f (\text{flow} (\text{name}, \star \text{par}, 0), \text{pnum}) = f (\text{child}_u, \text{pnum})$
  where child$_u$ is the solid child of the weak flow

---

Figure D.4: Updated disjunction_of_leaves Function
Appendix D Functions Used in Formalising the ERS Translation Rules

<table>
<thead>
<tr>
<th>union_of_leaves(ch: child/cons-child, n: int)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output Operation:</td>
</tr>
</tbody>
</table>

union_of_leaves(leaf (name), n) = \( \text{dom}_1(\ldots \text{dom}_n(\text{name}) \ldots) \)

|Traversing Steps:|

- \( f(\text{constructor}(c_1, \ldots, c_n), n) = f(c_j, n) \cup \ldots \cup f(c_p, n) \)
  
  \text{where constructor = and/or/xor}

- \( f(\text{exception-combinator}(c_{\text{normal-child}}, c_{\text{interrupting-child}}), n) = f(c_{\text{normal-child}}, n) \cup f(c_{\text{interrupting-child}}, n) \)
  
  \text{where exception-combinator = interrupt/retry}

- \( f(\text{replicator}(\text{par}, c), n) = f(c, n+1) \)
  
  \text{where replicator = all/some/one}

- \( f(\text{par-rep}(\text{par}, c), n) = f(c, n+1) \cup f(\text{nextChild}, n) \)
  
  \text{where nextChild = successor}(c, n)

- \( f(1^* \text{flow}, n) = f(\text{flow}, n) \cup \ldots \cup f(\text{flow}_n, n) \)

- \( f(\text{flow}(\text{name}, \ast \text{par}, 1), n) = f(\text{child}_1, n) \)
  
  \text{where child}_1 \text{ is the first child of the strong flow}

- \( f(\text{flow}(\text{name}, \ast \text{par}, 0), n) = f(\text{child}_1, n) \)
  
  \text{where child, is the solid child of the weak flow}

Figure D.5: Updated \text{union_of_leaves} Function
**build_seq_grd** (predecessor: child/cons-child, *par1: parameter list of predecessor, l: leaf, *par2: parameter list of l)

**Output Operation:**

\[ \text{build_seq_grd}(\text{leaf, } *\text{par1}, l, *\text{par2}) = \text{seq_grd}(\text{leaf, } *\text{par1}, l, *\text{par2}) \]

**Pattern Matching:**

- \( f(\text{and} (c_1, ..., c_n), *\text{par1}, l, *\text{par2}) = f(c_1, *\text{par1}, l, *\text{par2}) \land \ldots \land f(c_n, *\text{par1}, l, *\text{par2}) \)
- \( f(\text{or/xor} (c_1, ..., c_n), *\text{par1}, l, *\text{par2}) = f(c_1, *\text{par1}, l, *\text{par2}) \lor \ldots \lor f(c_n, *\text{par1}, l, *\text{par2}) \)
- \( f(\text{interrupt} (c_{\text{normal-child}}, c_{\text{interrupting-child}}), *\text{par1}, l, *\text{par2}) = f(c_{\text{normal-child}}, *\text{par1}, l, *\text{par2}) \lor f(c_{\text{interrupting-child}}, *\text{par1}, l, *\text{par2}) \)
- \( f(\text{retry} (c_{\text{normal-child}}, c_{\text{interrupting-child}}), *\text{par1}, l, *\text{par2}) = f(c_{\text{normal-child}}, *\text{par1}, l, *\text{par2}) \)
- \( f(\text{replicator} (p, c), *\text{par1}, l, *\text{par2}) = f(c, (*\text{par1}, p), l, *\text{par2}) \quad \text{where replicator = all/some/one} \)
- \( f(1^* \text{flow}, *\text{par1}, l, *\text{par2}) = f(\text{flow}_1, *\text{par1}, l, *\text{par2}) \lor \ldots \lor f(\text{flow}_{\text{nr}}, *\text{par1}, l, *\text{par2}) \)
- \( f(\text{flow} (...); 1), *\text{par1}, l, *\text{par2}) = f(\text{child}_1, *\text{par1}, l, *\text{par2}) \quad \text{where child, is the last child of the strong flow} \)
- \( f(\text{flow} (...); 0), *\text{par1}, l, *\text{par2}) = f(\text{child}_0, *\text{par1}, l, *\text{par2}) \quad \text{where child, is the solid child of the weak flow} \)

Figure D.6: Updated **build_seq_grd** Function
Figure D.7: Sequencing between Two Leaf Events (Salehi Fathabadi, 2012)
**Seq_grd** (e1: leaf, p_1...p_n; parameter list of e1, 
e2: leaf, p_1...p_m; parameter list of e2) =

- **e1 = TRUE** where (n = 0)

- **e1 ≠ ∅** where (n ≠ 0) and (m = 0) and
  (there is no all-replicator parameter in (p_1...p_n))

- **W = Type(p_k)** where (n ≠ 0) and (m = 0) and
  (p_k is an all-replicator parameter in (1 ≤ k ≤ n))

- **e1 ≠ ∅** where (n ≠ 0) and (m ≠ 0) and
  (there is no common parent parameter) and
  (there is no all-replicator parameter in (p_1...p_m))

- **W = Type(p_k)** where (n ≠ 0) and (m ≠ 0) and
  (there is no common parent parameter) and
  (p_k is an all-replicator parameter in (1 ≤ k ≤ n)) and
  (there is no parameter in (p_1...p_m) with same type as p_k)

- **p_i ∈ W** where (n ≠ 0) and (m ≠ 0) and
  (there is no common parent parameter) and
  (p_k is an all-replicator parameter in (1 ≤ k ≤ n)) and
  (Type(p_k) = Type(p_i) (1 ≤ l ≤ m))

- **p_1 ↦ ... ↦ p_i ∈ X** where (n ≠ 0) and (m ≠ 0) and
  (p_1...p_i is list of common parent parameter) and
  (there is no all-replicator parameter in (p_{i+1}...p_n))

- **Z [(p_1 ↦ ... ↦ p_i)] = Type(p_k)** where (n ≠ 0) and (m ≠ 0) and
  (p_1...p_i is list of common parent parameter) and
  (p_k is an all-replicator parameter in (p_{i+1}...p_n))

---

Figure D.8: Original *seq_grd* Function (Salehi Fathabadi, 2012)
References


REFERENCES


REFERENCES


