Feature-oriented Reuse with Event-B & Rodin

by

Ali Ameer Gondal

Thesis for the degree of Doctor of Philosophy

June 2013
This thesis presents our knowledge gained and work done to support our research contribution of introducing reusability of formal specifications using Event-B formal method. We are particularly interested in investigating approaches to reuse formal specifications for software product line (SPL) modelling using Event-B. Event-B is a recent formalism for specification and verification of software and hardware systems. It is supported by an extensible tool called Rodin. SPLs are a set of related products having commonalities built from shared assets. Feature modelling is a well-known technique for product line development.

In order to explore feature-oriented reuse approaches for SPL modelling in Event-B, we experimented with two case-study examples using feature-based approaches of syntactic cut-and-paste as well as established Event-B methods of (de)composition. Our first case-study of Production Cell - a metal processing plant - was modelled using Event-B in different ways to explore reusability, evaluate existing Event-B (de)composition techniques and suggest patterns for generic reuse. Our second case-study - ATM - suggested a modelling pattern by combining the two (de)composition styles of Event-B. By following this pattern, Event-B models could be reused and composed to build various products of an SPL without the need to reprove the composed models. Further features could be developed to enhance the product line and to build more products by reusing already modelled and proved specifications, saving lot of user time and effort. We have developed a prototype tool for feature modelling in Event-B as a Rodin plug-in that can be used to build feature models for SPLs and these feature models could be configured to model various members of the product line. This tool extends our feature composition tool to compose Event-B models in a fairly loose structure using cut-and-paste approach appropriate for feature modelling. We also suggested modelling guidelines in the form of modelling patterns for feature-based development in Event-B. Both the case-studies highlight a number of future directions to further enhance this reuse approach. This work needs to be applied to more case-studies and further tool development will be needed to automate various phases of the suggested modelling patterns.
## Contents

Declaration of Authorship ........................................................... xv
Acknowledgements ................................................................. xvii
Nomenclature ........................................................................ xxii

1 Introduction ................................................................................. 1
   1.1 Problem Area ................................................................. 1
   1.2 Research Focus .............................................................. 1
   1.3 Research Methodology .................................................... 2
   1.4 Research Contribution ................................................... 3
   1.5 Thesis Structure ............................................................. 4

2 Literature Review : Formal Methods ........................................... 5
   2.1 Introduction ................................................................. 5
   2.2 Formal Methods ........................................................... 5
   2.3 Formal Specification languages ....................................... 6
      2.3.1 Z notation ............................................................ 7
      2.3.2 VDM ................................................................. 10
      2.3.3 B-Method ........................................................... 12
      2.3.3.1 ProB Tool ......................................................... 13
      2.3.4 Event-B .............................................................. 14
      2.3.4.1 Rodin ............................................................ 17
   2.4 Brief Comparison of Event-B, B & Z notation ..................... 18
   2.5 Other formalisms ........................................................... 19
      2.5.1 CSP .................................................................. 19
      2.5.2 Action System ..................................................... 19
   2.6 Composition ................................................................. 21
      2.6.1 Composition in B, Z & VDM .................................... 21
      2.6.2 Decomposition & Composition in Event-B .................. 22
      2.6.2.1 Decomposition in Event-B ................................ 22
      2.6.2.2 Composition in Event-B .................................... 24
      2.6.3 Related Work ...................................................... 25
      2.6.3.1 Generic Instantiation ....................................... 25
   2.7 Conclusion ................................................................. 27

3 Literature Review : MDA, Product Lines and Feature Modelling .... 29
   3.1 Introduction ................................................................. 29
3.2 Model-Driven Architecture (MDA) ........................................ 29
  3.2.1 Introduction .................................................. 29
  3.2.2 Benefits of MDA ............................................. 31
  3.2.3 Eclipse Modelling Framework (EMF) ......................... 32
  3.2.4 Graphical Modelling Framework (GMF) ....................... 33
  3.2.5 Model Transformation ...................................... 34
    3.2.5.1 Introduction ......................................... 34
    3.2.5.2 Transformation Languages .............................. 36
3.3 Software Product Lines ........................................... 40
  3.3.1 Introduction ................................................ 40
  3.3.2 Commonality & Variability in SPLs ......................... 40
  3.3.3 SPLs Engineering Approaches ............................... 41
3.4 Feature Modelling ............................................... 41
  3.4.1 Introduction ................................................ 41
  3.4.2 Existing Feature Modelling Notations & Tool Support .......... 42
  3.4.3 Related Work .............................................. 45
3.5 Conclusion ...................................................... 46

4 Feature Modelling in Event-B & Tool Support .................. 49
  4.1 Introduction .................................................. 49
  4.2 A Feature Modelling Approach for Event-B .................... 49
    4.2.1 Why to reinvent the wheel? .............................. 50
    4.2.2 Our Feature Modelling Notation ......................... 51
    4.2.3 Metamodelling .......................................... 53
    4.2.4 Feature Model Validation ................................ 54
  4.3 Current Feature Modelling Tool Support for Event-B .......... 55
    4.3.1 Feature Model Editor .................................... 56
      4.3.1.1 Implementing Feature Model Validation .............. 56
    4.3.2 Model To Metamodel Transformation .................... 57
    4.3.3 Feature Configurator .................................... 58
      4.3.3.1 Conflict Resolution .................................. 58
      4.3.3.2 Feature Composition ................................ 60
      4.3.3.3 Limitations of Feature Composition ................ 63
  4.4 Conclusion .................................................... 64

5 Case Study - Production Cell .................................... 65
  5.1 Introduction .................................................. 65
  5.2 Roadmap ....................................................... 65
  5.3 Production Cell .............................................. 66
  5.4 PC Component-based ......................................... 67
    5.4.1 Development Structure .................................. 67
    5.4.2 PC Abstract Model (PC_0) ............................... 68
    5.4.3 PC First Refinement (PC_1) ............................. 69
    5.4.4 PC Second Refinement (PC_2) ......................... 71
    5.4.5 PC Third Refinement (PC_3) ......................... 72
    5.4.6 PC Fourth Refinement (PC_4) ......................... 74
    5.4.7 Decomposition ........................................... 75
| 5.4.7.1 | Shared-Variable Decomposition (SVD) | 76 |
| 5.4.7.2 | Shared-Event Decomposition (SED) | 77 |
| 5.4.7.3 | Discussion | 79 |
| 5.4.8 | Press First Refinement Model (Press_1) | 81 |
| 5.4.9 | Press Second Refinement Model (Press_2) | 83 |
| 5.4.10 | Press Third Refinement Model (Press_3) | 83 |
| 5.4.11 | Building Variants of PC | 84 |
| 5.4.12 | Evaluation | 89 |
| 5.5 | PC Controller-based | 93 |
| 5.5.1 | Magnet Controller | 95 |
| 5.5.2 | Movement Controller | 97 |
| 5.5.3 | Instantiation & Composition | 98 |
| 5.5.4 | Evaluation | 102 |
| 5.6 | PC – Domain-specific SPL modelling through contexts | 103 |
| 5.6.1 | Evaluation | 104 |
| 5.7 | Conclusion | 105 |

6 Modelling ATM Case-study

6.1 Introduction | 109
6.2 ATM | 109
6.3 Roadmap | 110
6.4 Event-B Modelling of ATM Features | 111
6.4.1 Refinement of Transfer Feature | 113
6.4.2 Refinement of Deposit Feature | 118
6.4.3 Composing Sub-components of Two Features | 122
6.4.4 Modelling Second ATM Product By Reusing Existing Features | 126
6.4.4.1 Refinement of Withdraw Feature | 127
6.4.4.2 Discussion | 127
6.4.5 Evaluation | 129
6.5 Conclusion | 131

7 Guidelines for Feature Modelling in Event-B & Tooling Requirements

7.1 Introduction | 133
7.2 Guidelines for Feature Modelling in Event-B | 133
7.2.1 Modelling a System as a Set of Generic Features | 134
7.2.1.1 Requirements Manipulation | 134
7.2.1.2 Event-B Development | 135
7.2.1.3 Instantiation and Composition | 136
7.2.2 Modelling a System as a Set of Separable Features | 136
7.2.2.1 Problem Decomposition | 137
7.2.2.2 Architectural Decomposition | 139
7.2.2.3 Architectural Composition | 139
7.2.2.4 Modelling Second Product Through Reuse | 140
7.3 Future Tool Support Requirements | 142
7.3.1 Refactoring Support | 142
7.3.2 Feature Modelling and Topology Tool | 142
7.3.3 SVC Tool | 143
CONTENTS

7.3.4 Composition Replay Support ........................................ 143
7.4 Conclusion ..................................................................... 144

8 Conclusion & Future Work ................................................ 145
  8.1 Summary .................................................................... 145
  8.2 Research Contribution ................................................ 147
  8.3 Future Work .................................................................. 149

A Production Cell Requirements Specification ....................... 153

B ATM Requirements Specification ....................................... 157

References ......................................................................... 161
List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Structure of Z schema [47]</td>
<td>8</td>
</tr>
<tr>
<td>2.2</td>
<td>Single-line structure of Z schema [47]</td>
<td>8</td>
</tr>
<tr>
<td>2.3</td>
<td>Structure of operational schema in Z [47]</td>
<td>9</td>
</tr>
<tr>
<td>2.4</td>
<td>Example of schema inclusion [47]</td>
<td>9</td>
</tr>
<tr>
<td>2.5</td>
<td>Example of operational schema</td>
<td>10</td>
</tr>
<tr>
<td>2.6</td>
<td><em>BirthdayBook</em> schema [47]</td>
<td>10</td>
</tr>
<tr>
<td>2.7</td>
<td>Structure of a VDM module</td>
<td>10</td>
</tr>
<tr>
<td>2.8</td>
<td>VDM example</td>
<td>11</td>
</tr>
<tr>
<td>2.9</td>
<td>Structure of a B Machine</td>
<td>13</td>
</tr>
<tr>
<td>2.10</td>
<td>Example of a B Machine</td>
<td>14</td>
</tr>
<tr>
<td>2.11</td>
<td>Event-B Machine &amp; Context structure</td>
<td>15</td>
</tr>
<tr>
<td>2.12</td>
<td>Event-B refinement structure</td>
<td>16</td>
</tr>
<tr>
<td>2.13</td>
<td>Event-B example of Birthday Book</td>
<td>17</td>
</tr>
<tr>
<td>2.14</td>
<td>Shared-variable decomposition</td>
<td>23</td>
</tr>
<tr>
<td>2.15</td>
<td>Shared-event decomposition</td>
<td>23</td>
</tr>
<tr>
<td>2.16</td>
<td>Fusion composition example</td>
<td>24</td>
</tr>
<tr>
<td>2.17</td>
<td>Refinement in fusion composition</td>
<td>26</td>
</tr>
<tr>
<td>2.18</td>
<td>Instantiation of a generic chain of refinements [128]</td>
<td>26</td>
</tr>
<tr>
<td>3.1</td>
<td>MDA software development lifecycle [95]</td>
<td>30</td>
</tr>
<tr>
<td>3.2</td>
<td>GMF overview [7]</td>
<td>34</td>
</tr>
<tr>
<td>3.3</td>
<td>Model transformation process [33]</td>
<td>35</td>
</tr>
<tr>
<td>3.4</td>
<td>ATL module example [33]</td>
<td>37</td>
</tr>
<tr>
<td>3.5</td>
<td>MTF rule example</td>
<td>38</td>
</tr>
<tr>
<td>3.6</td>
<td>ETL example</td>
<td>39</td>
</tr>
<tr>
<td>3.7</td>
<td>Example of a feature model (FODA style) [93]</td>
<td>43</td>
</tr>
<tr>
<td>3.8</td>
<td>Example of a feature model [29]</td>
<td>44</td>
</tr>
<tr>
<td>3.9</td>
<td>Feature configuration diagram and its specialisation [29]</td>
<td>45</td>
</tr>
<tr>
<td>4.1</td>
<td>Example feature model</td>
<td>53</td>
</tr>
<tr>
<td>4.2</td>
<td>Feature modelling metamodel</td>
<td>53</td>
</tr>
<tr>
<td>4.3</td>
<td>Feature modelling tool architecture</td>
<td>55</td>
</tr>
<tr>
<td>4.4</td>
<td>PC feature model of Figure 4.1 transformed into metamodel for further</td>
<td>58</td>
</tr>
<tr>
<td></td>
<td>instantiation</td>
<td></td>
</tr>
<tr>
<td>4.5</td>
<td>Feature configuration editor screenshot</td>
<td>59</td>
</tr>
<tr>
<td>4.6</td>
<td>Event-B specification of machines M1 and M2</td>
<td>59</td>
</tr>
<tr>
<td>4.7</td>
<td>Simple composition of Event-B machines M1 and M2</td>
<td>61</td>
</tr>
<tr>
<td>4.8</td>
<td>Disjoint composition of Event-B machines M1 and M2</td>
<td>62</td>
</tr>
<tr>
<td>Chapter</td>
<td>Figure</td>
<td>Description</td>
</tr>
<tr>
<td>---------</td>
<td>--------</td>
<td>-------------</td>
</tr>
<tr>
<td>4.9</td>
<td>Composing non-conformal refinement chains</td>
<td>64</td>
</tr>
<tr>
<td>5.1</td>
<td>Top view of production cell plant [104]</td>
<td>67</td>
</tr>
<tr>
<td>5.2</td>
<td>Component-based PC modelling</td>
<td>68</td>
</tr>
<tr>
<td>5.3</td>
<td>PC abstract model</td>
<td>69</td>
</tr>
<tr>
<td>5.4</td>
<td>Partial model of PC first refinement</td>
<td>70</td>
</tr>
<tr>
<td>5.5</td>
<td>$PC_1$ loadFeedBelt event</td>
<td>71</td>
</tr>
<tr>
<td>5.6</td>
<td>Atomicity refinement of $PC_0$ event in $PC_1$</td>
<td>71</td>
</tr>
<tr>
<td>5.7</td>
<td>Atomicity refinement of loadTable event in $PC_2$</td>
<td>72</td>
</tr>
<tr>
<td>5.8</td>
<td>Invariants of $PC_3$ for safety requirements of production cell</td>
<td>73</td>
</tr>
<tr>
<td>5.9</td>
<td>Refinement of event loadFeedBelt in $PC_4$</td>
<td>75</td>
</tr>
<tr>
<td>5.10</td>
<td>loadTable as internal and external event in Table and Press components respectively after SVD</td>
<td>76</td>
</tr>
<tr>
<td>5.11</td>
<td>Event splitting example for SED</td>
<td>78</td>
</tr>
<tr>
<td>5.12</td>
<td>Example of preparing an event for SED</td>
<td>79</td>
</tr>
<tr>
<td>5.13</td>
<td>Event of Figure 5.12 decomposed into two events</td>
<td>79</td>
</tr>
<tr>
<td>5.14</td>
<td>Variables and events of Press.0 resulting from SVD of $PC_4$</td>
<td>80</td>
</tr>
<tr>
<td>5.15</td>
<td>Event refinement for press component</td>
<td>82</td>
</tr>
<tr>
<td>5.16</td>
<td>Events of Press.1 resulting from atomicity decomposition of movePressToLow event of Press.0</td>
<td>82</td>
</tr>
<tr>
<td>5.17</td>
<td>Events of Press.2 resulting from atomicity decomposition of stopMotorAtLowPos event of Press.1</td>
<td>83</td>
</tr>
<tr>
<td>5.18</td>
<td>Events of Press.3 resulting from atomicity decomposition of stopMotorAtLowPos event of Press.2</td>
<td>85</td>
</tr>
<tr>
<td>5.19</td>
<td>PC Topology1</td>
<td>85</td>
</tr>
<tr>
<td>5.20</td>
<td>PC Topology2</td>
<td>86</td>
</tr>
<tr>
<td>5.21</td>
<td>Events refactoring for modelling topology2 after topology1 - 1st refinement</td>
<td>88</td>
</tr>
<tr>
<td>5.22</td>
<td>Events refactoring for modelling topology2 after topology1 - 2nd, 3rd and 4th refinement</td>
<td>88</td>
</tr>
<tr>
<td>5.23</td>
<td>Refinement architecture for modelling the two topologies</td>
<td>89</td>
</tr>
<tr>
<td>5.24</td>
<td>Event instantiation example for PC topology2</td>
<td>90</td>
</tr>
<tr>
<td>5.25</td>
<td>Event instantiation example for PC topology2</td>
<td>90</td>
</tr>
<tr>
<td>5.26</td>
<td>Proof obligations refactoring for an event of topology 2</td>
<td>91</td>
</tr>
<tr>
<td>5.27</td>
<td>Controller-based PC modelling</td>
<td>94</td>
</tr>
<tr>
<td>5.28</td>
<td>Atomicity refinement of magnet controller</td>
<td>96</td>
</tr>
<tr>
<td>5.29</td>
<td>Magnet controller third refinement model</td>
<td>98</td>
</tr>
<tr>
<td>5.30</td>
<td>Movement controller events and invariants for first refinement</td>
<td>99</td>
</tr>
<tr>
<td>5.31</td>
<td>Crane instantiation</td>
<td>100</td>
</tr>
<tr>
<td>5.32</td>
<td>Event specialisation of magnet controller for crane</td>
<td>100</td>
</tr>
<tr>
<td>5.33</td>
<td>Event specialisation of movement controller for crane</td>
<td>100</td>
</tr>
<tr>
<td>5.34</td>
<td>Guard strengthening of events during composition</td>
<td>101</td>
</tr>
<tr>
<td>5.35</td>
<td>Guard strengthening of events during composition</td>
<td>101</td>
</tr>
<tr>
<td>5.36</td>
<td>Development architecture for SPL modelling through context switching</td>
<td>103</td>
</tr>
<tr>
<td>5.37</td>
<td>Partial Event-B model of domain-specific modelling through context switching</td>
<td>105</td>
</tr>
<tr>
<td>6.1</td>
<td>ATM feature model</td>
<td>110</td>
</tr>
<tr>
<td>6.2</td>
<td>Integral ATM abstract model for two features</td>
<td>111</td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>--------</td>
<td>-------------------------------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>6.3</td>
<td>ATM integral model decomposed using SVD</td>
<td>112</td>
</tr>
<tr>
<td>6.4</td>
<td>ATM integral model decomposed using SVD into transfer and deposit features</td>
<td>113</td>
</tr>
<tr>
<td>6.5</td>
<td>Transfer model’s invariants and events of first refinement</td>
<td>114</td>
</tr>
<tr>
<td>6.6</td>
<td>Event refinement of transfer feature</td>
<td>115</td>
</tr>
<tr>
<td>6.7</td>
<td>Transfer model’s events of second refinement</td>
<td>116</td>
</tr>
<tr>
<td>6.8</td>
<td>Transfer fourth refinement model with events ready for architectural decomposition</td>
<td>117</td>
</tr>
<tr>
<td>6.9</td>
<td>Splitting external event Deposit into two during architectural decomposition</td>
<td>119</td>
</tr>
<tr>
<td>6.10</td>
<td>An event of transfer feature’s fourth refinement model split into two events using SED</td>
<td>120</td>
</tr>
<tr>
<td>6.11</td>
<td>Deposit feature’s events of second refinement</td>
<td>121</td>
</tr>
<tr>
<td>6.12</td>
<td>Event refinement of deposit feature</td>
<td>122</td>
</tr>
<tr>
<td>6.13</td>
<td>Deposit feature’s fourth refinement model with events ready for architectural decomposition</td>
<td>122</td>
</tr>
<tr>
<td>6.14</td>
<td>Refinement and (de)composition architecture for transfer and deposit features</td>
<td>123</td>
</tr>
<tr>
<td>6.15</td>
<td>Composing bank components of transfer and deposit features</td>
<td>125</td>
</tr>
<tr>
<td>6.16</td>
<td>Refinement preservation experiment of the suggested pattern</td>
<td>126</td>
</tr>
<tr>
<td>6.17</td>
<td>Refinement and (de)composition architecture for 2nd ATM product</td>
<td>127</td>
</tr>
<tr>
<td>6.18</td>
<td>Integral ATM abstract model for 2nd ATM product of Figure 6.17</td>
<td>128</td>
</tr>
<tr>
<td>6.19</td>
<td>Event refinement of withdraw feature</td>
<td>128</td>
</tr>
<tr>
<td>6.20</td>
<td>Feature-oriented refinement and (de)composition pattern</td>
<td>130</td>
</tr>
<tr>
<td>7.1</td>
<td>Modelling a product line as a set of generic features</td>
<td>135</td>
</tr>
<tr>
<td>7.2</td>
<td>Example feature model for a product line</td>
<td>137</td>
</tr>
<tr>
<td>7.3</td>
<td>Modelling a product line as a set of separable features</td>
<td>138</td>
</tr>
<tr>
<td>7.4</td>
<td>Product 1 configurations of our example feature model of Figure 7.2</td>
<td>138</td>
</tr>
<tr>
<td>7.5</td>
<td>Modelling a product line variant by reusing existing features</td>
<td>141</td>
</tr>
<tr>
<td>7.6</td>
<td>Product 2 configuration of our example feature model of Figure 7.2</td>
<td>141</td>
</tr>
</tbody>
</table>
List of Tables

3.1 Feature modelling notation by Czarnecki et al. [29] .................................. 43
4.1 Our feature modelling graphical notation .................................................... 52
5.1 Event distribution among components of PC based on SVD .................... 76
5.2 Number of local and shared events of PC components based on SED ........ 77
5.3 Events distribution of PC components when performed SVD on a model prepared for SED ................................................................. 78
5.4 Number of events at different refinement levels of press component decomposed using SVD and SED ......................................................... 84
5.5 Proofs statistics of component-based PC modelled for SED/SVD ............ 84
5.6 Proof obligations statistics for two topology developments ..................... 91
5.7 Events and invariants statistics for topology 1 and 2 ............................... 92
5.8 Requirements description for Table and Movement features of PC .......... 94
5.9 Requirements description for Crane and Magnet features of PC ............. 95
5.10 Proof obligations statistics for crane development ................................. 101
6.1 Proof statistics for transfer refinements ...................................................... 115
6.2 Proof statistics for deposit refinements ...................................................... 121
6.3 Proof statistics for deposit and transfer components and their composition 125
6.4 Proof obligations for withdraw component and the composite models .... 129
Declaration of Authorship

I, Ali Ameer Gondal, declare that the thesis entitled Feature-oriented Reuse with Event-B & Rodin 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: [118], [80], [79] and [78]

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

Date:.................................................................................................................................

xv
Acknowledgements

I would like to thank my supervisor Dr. Mike Poppleton for his continuous support throughout these years. My sincere thanks to Prof. Michael Butler, Dr. Bernd Fischer and Dr. Colin Snook for their guidance. I would like to thank all my colleagues in our research group. This work would not have been possible without the love and support of my family. Special thanks to all my friends who have always been there whenever I needed them.
To My Parents
Nomenclature

ATM    Auto-Teller Machine
EMF    Eclipse Modelling Framework
FMT    Feature Modelling Tool
FOSD   Feature-Oriented Software Development
GMF    Graphical Modelling Framework
MDA    Model-Driven Architecture
PC     Production Cell
PL     Product Line
SEC    Shared-Event Composition
SED    Shared-Event Decomposition
SVC    Shared-Variable Composition
SVD    Shared-Variable Decomposition
SPL    Software Product Line
XML    Extensible Markup Language
XMI    XML Metadata Interchange
Chapter 1

Introduction

The following sections briefly discuss the problem that we are studying and the focus of our research. Moreover, we outline the approach that we have chosen to tackle this problem. The thesis structure is given towards the end of the chapter highlighting the remaining chapters of the thesis.

1.1 Problem Area

Engineering software is becoming increasingly complex because of the changing nature of businesses, rapidly changing user requirements and the advances in technology. The growing need to release products before others to compete in the market also makes it difficult to carry out extensive testing and fixing of bugs. This often leads to the products delivered having bugs and then spending out of budget amounts during the maintenance. On the other hand, building systems by reusing parts of the existing systems effectively increases quality and reduces the software development costs, time to market and risk of project failures [131]. The idea of developing systems in such a way that these systems can be reused in future has many advantages. Also, the use of formal methods for specification and verification of software and hardware systems involves a lot of effort in terms of modelling and mathematical proofs. The research problem is, how to maximise the reuse of existing specifications and their associated proofs when formally specifying similar systems in Event-B using Rodin tool.

1.2 Research Focus

The research is focused on developing prototype tools and methodological support for reusability of formal specifications and models using Event-B [22]. Event-B is a relatively new formal method. It is based on set-theory and first-order logic. The method is applied
Chapter 1 Introduction

for specification and verification of software and hardware systems. It is supported by an Eclipse [2] based open-source toolkit called Rodin [23], which includes editors, provers, animator, model checker and various other plugins. The mathematical language of Event-B forms the core of Rodin platform, which can easily be extended by developing additional plug-ins on the top of Rodin core. We are particularly interested in applying software product line (SPL) reuse in Event-B. An SPL is a set of related products, which share a common base having significant variabilities to meet user requirements [66]. Each member of an SPL is distinct from other members in terms of functionality or behaviour, while sharing the common base of product line features. SPL development promises the benefits of reusability, i.e., improved quality and reduced effort and time to market. Feature modelling is a well known technique for SPL engineering.

Our aim is to provide SPL reuse infrastructure for Event-B modellers where they can build a repository of specifications for product line features, which can then be reused to build further members of the product family. The research area, i.e., using formal methods for SPL engineering, is quite recent [124, 62, 88, 94], unlike the application of formal methods to software reuse which has been around for nearly two decades [61]. We are exploring this within Event-B language by looking at both the theoretical aspects as well as tooling side to see if it actually helps in increasing productivity. We will see whether existing Event-B tools and techniques are capable enough to model SPLs and if not, we will suggest techniques and modelling guidelines to gain the benefits of reusability in Event-B. The major benefits we envisage are in reducing the efforts of modelling, refinement and proving while building similar systems by reusing existing specifications.

At present, Event-B and Rodin tool do not support SPL reuse. There is no support to embed variability and commonality that exists in modelling a product family - one aspect that is on the core of SPL development. There is a limited support for composing specifications in Event-B, which makes SPL modelling difficult. Another important issue is redoing the proof effort while reusing already proven specifications. These shortcomings suggest the need of a specific SPL reuse approach to model a product line and improve productivity by reducing the proving efforts.

1.3 Research Methodology

In order to address our research problem of introducing product line reuse in Event-B, we have looked at the approaches for SPL development. We adapted and extended a well known feature-oriented modelling approach where we could specify features of a product line and these features could then be configured and composed in different ways to model product line members. We defined the notion of a ‘feature’ as a complete Event-B development. This is required due to the refinement mechanism of Event-B
language where an Event-B development starts at a higher abstraction level which is then refined to a concrete specification in a number of steps.

We have modelled two case-studies in Event-B, i.e., production cell (PC) and auto-teller machine (ATM), to explore reuse opportunities and to apply feature modelling approach for SPL development. Our first case-study (PC) suits the typical top-down refinement approach of Event-B. At first, this case-study did not show obvious reuse potential in terms of functional requirements features; variability arises as different possible connection topologies between PC components. We then modelled it in a different way by considering a fine-grained view of features as controllers that were generic and revealed more reuse opportunity for PC modelling. By modelling PC in different ways, we explored Event-B’s capability for feature modelling using its decomposition and genericity techniques to exploit reuse.

In order to support our findings of the PC case-study, we then modelled ATM case-study. ATM example is more reuse-oriented in terms of functional features and suits the traditional feature modelling approach, where we can have a product line of ATM systems having different features. This has revealed a pattern of modelling that could be used as a guideline for SPL modelling in Event-B for future users. Both case-studies use existing (de)composition approaches of Event-B and also provide examples where these fall short.

We have also developed a prototype tool for feature modelling in Event-B, as a Rodin plug-in. The tool can be used to specify SPLs along with the variability among different product line members, configuring and composing these to model instances of product lines. Our experiment to use existing feature modelling tool for SPL engineering with Event-B was not as successful as desired. This required a lot of extra work and since the Event-B and Rodin are being improved constantly, a feasible solution is to have Rodin compatible tool and notation for feature modelling in Event-B. Based on our case-study experiments, we have also provided guidelines for feature modelling in Event-B for future users and how a system can be modelled to improve reusability of specification and their proofs by following a particular pattern of modelling. We also highlight the tooling support required to improve this reuse approach in Event-B.

1.4 Research Contribution

Our research contribution presented in this thesis can be summarised as:

- Modelling of two real-life examples in different ways using Event-B to exploit reuse
- Exploiting Event-B’s existing capabilities for feature-oriented modelling
- Providing a methodology for feature-oriented reuse in Event-B
• Tool support for feature-modelling in Event-B

• Guidelines for modelling product lines in Event-B

• Future research directions to improve feature-oriented reuse in Event-B and tool support requirements

1.5 Thesis Structure

Chapter 2 covers the formal methods background, various formal specification languages and existing reuse and composition approaches of Event-B. Chapter 3 provides an overview of model-driven architecture, software product lines and feature modelling. We present our feature modelling notation and the developed tool support in Chapter 4. Chapter 5 discusses our production cell case-study and how it has been modelled in different ways followed by another case-study ATM in Chapter 6. Guidelines for modelling SPLs in Event-B and future tool requirements are presented in Chapter 7, followed by conclusion and future work directions presented in Chapter 8.
Chapter 2

Literature Review : Formal Methods

2.1 Introduction

In this chapter, we give an overview of formal methods and tool support available for some of these. Although there are many formal specification languages being used, we only discuss a few which are related to or have lead to the invention of Event-B formal method (which is the focus of our research). We also present existing composition/decomposition and reuse approaches of Event-B language, since some of our research contribution is based on top of these. In later chapters, we also explore to what extent we can use these approaches to enhance reusability in Event-B and suggest further tools and techniques, based on our case-study experiments, to fully benefit from reuse of formal specifications.

2.2 Formal Methods

“Formal methods are mathematically based languages, techniques, and tools for specifying and verifying hardware and software systems” [20, 57]. We can not guarantee complete correctness by using formal methods; however, these can greatly help us in understanding the system and finding out any ambiguities and inconsistencies earlier in the software development life cycle (SDLC), which are difficult to find otherwise [49, 63]. Formal methods can also provide mechanisms for mathematically proving that a software system or component does what it is supposed to do. It is also well known that cost of finding and fixing the bugs later in the SDLC is much more expensive than in the earlier requirements analysis and design phase [63]. Therefore, formal modelling, which is carried out in the earlier phase, helps in finding such bugs and reducing the
overall cost for fixing these [49]. The reduction in testing cost to offset increased cost in requirements/design phase can be seen in various industrial examples. Although, when formal modelling is used, more time is spent in the earlier phase of SDLC, significant reduction in the overall development time has been reported [140]. Another viewpoint for defining formal methods is that they assure to some extent that the system being specified will be correct by construction. Correctness by construction is achieved by specifying certain behavioural properties and mathematically proving that the method being used is correct. The following are some distinct characteristics of Correctness by Construction systems as detailed in [71]: use of static verification; use of small, verifiable design steps; generation of certification and validation evidence as a side effect to the development process; appropriate use of formality; and finally, the use of right notations and tools for the job.

Formal methods have been around for many years but unlike other software engineering methodologies, e.g., object-oriented programming, these have not been widely used in the industry. This is mainly because of the lack of robust tools and techniques. Those available are quite complex to understand, hard to use by ordinary software engineers and need more specific expertise such as theorem proving skills [83]. Until a few years back, there were only a small number of industrial projects using formal methods, which were not enough to convince practicing software and hardware engineers to start using formal methods. Another reason that formal methods are not so popular in the industry is that it is considered only suitable for critical systems which involve human life. This is not always true as other mission-critical projects involving high costs such as financial and banking systems can also benefit from the proper use of formal methods.

Successful application of formal methods in safety-critical systems can be seen in aerospace, transportation, defence and medical sectors [48, 44, 21, 140]. Recent improvement in formal specification languages, verification techniques and tools helped in reviving the hope that software engineers might start using formal methods soon. Particularly, including industrial partners in the Deploy [1] project such as Nokia, Bosch, SAP etc. can be seen as a positive sign towards the adoption of formal methods in industrial projects.

2.3 Formal Specification languages

Formal specification languages are used to specify systems which can then be validated and verified. In software engineering, “validation means building the right systems and verification refers to building the systems right” [121]. Formal specifications can be verified using mathematical proofs. The results of the proofs would then provide sufficient verification evidence to a system’s developers that the system, as formally specified in their specification language of choice in their model, is correct [71]. This is achieved by using provers and model-checkers. Formal specification languages are
different from programming languages in a way that these specify systems at higher levels of abstraction and leave implementation level details to be introduced through ‘refinement’ later on.

Refinement is a process of introducing more details in each model transformation step starting from an abstract specification to the concrete one, which is closer to the implementation. This is also termed step-wise refinement. Each step towards the concrete model must be verified as a valid refinement of the abstract model. This is usually achieved by discharging the refinement proof obligations (POs). POs are the verification conditions automatically generated by the tool and then discharged using theorem provers to verify correctness of the model. There are two types of refinements called data and operation refinements. In data refinement, abstract data types are replaced by increasingly more implementable data types, ultimately arrays, records, etc. In operation refinement, an abstract operation specification is refined by introducing algorithmic detail, ultimately a deterministic procedure implementable in a compilable high level language. The refinement process can also be classified as horizontal and vertical refinement [22]. In horizontal refinement, more details are added to the abstract model to elaborate the existing specification or introduce further requirements of the system being modelled. In vertical refinement, the focus is on design decisions, i.e., transforming and enriching data types and the elaboration of algorithms. These concepts have been used while modelling our case-studies and building the tool support.

There are many formal modelling languages, some of the state-based formal methods include: Z [47], B [20], VDM [90], Action System [39], Event-B [22] etc. and process-based formal methods include: CSP [85], ACP [43], CCS [109] etc. Following is an overview of few state-based formal specification languages (Z, VDM, B, Event-B). We will focus more on Event-B as our research is based on this.

2.3.1 Z notation

Z notation was introduced by Jean-Raymond Abrial and others while working at Oxford University in late 70’s. It is a formal specification language based on set theory and mathematical logic. The set theory used in Z notation includes standard set operators, set comprehensions, cartesian products, and power sets. The mathematical logic is a first-order predicate calculus. Together, they form a mathematical language that is easy to learn and to apply. The focus of designing Z notation was on making it more human readable rather than machine executable. Hence, Z is more suitable for documentation. [47]

Z notation is specified using the box structure called “schema”. The schemas model mathematical objects and their properties to specify the static and dynamic aspects of the system. The static aspect includes the states that a system can have and the
conditions (invariants) needed to be satisfied as the system moves from one state to another. The dynamic aspect includes the operations of the system, the relationship between the input and the output and the changes of the state that can take place [134]. It also makes use of the types to represent different mathematical objects. There are several tools available for type-checking of Z notation. Another important aspect of Z notation is refinement. It is supported through data and operational refinement. An abstract specification can be refined in a number of steps depending on the complexity of the system to produce a concrete specification. The concrete specification is often so close to the implementation that it can be easily translated into executable code.

The box structure of schema is divided into two parts as shown in the example schema of Figure 2.1. The StateSpace specifies the state variables \(x_1, x_2, \ldots, x_n\) and their associated types \(S_1, S_2, \ldots, S_n\) in the upper half of the schemas. Any constraints on these variables are shown in the lower half as invariants \((\text{Inv})\). Schemas can also be written in a single line if needed. The single line schema shown in Figure 2.2 is equivalent to the one in Figure 2.1. The order is not important for variables and predicates in Z schemas.

In an operation schema, the variables decorated with a prime (') show the after (next) state and those without it show the before (current) state. The input variables are decorated with a question mark (?) and output variables are denoted by an exclamation mark (!). The structure of operational schema is given in Figure 2.3. Here, ‘Pre’ means the preconditions, ‘Inv’ are the invariants and ‘Op’ specifies the state changes. [47]

Z notation allows schemas to be included within other schemas and is known as ‘schema inclusion’. This makes the specification short, more readable and helps to understand larger specifications. For example, we can include StateSpace schema into Operation schema as shown in Figure 2.4, which abbreviates the schemas of Figures 2.2 and Figure 2.3. The inclusion operator \(\Delta\) allows changing the state space of the included schema whereas the other inclusion operator \(\Xi\) leaves the state unchanged. This means that \(\Xi\)
\( \Delta \) when the state variables (e.g., \( x \)) of the included schema are not affected by the operation of an operational schema (i.e., \( x' = x \)).

An example specification using Z schema is given in Figure 2.5. This specifies the operation of removing a birthday entry from a birthday book. This is an extension of the birthday book example discussed in [47]. It includes the BirthdayBook schema given in Figure 2.6, where each person can have only one birthday specified using partial function from \( NAME \) to \( DATE \). The variable \( known \) contains names of the people added to the birthday book and must be in the domain of the \( birthday \). In \( RemoveBirthday \) schema, the input variable \( name \) is of the type \( NAME \) and the other input variable \( date \) is of type \( DATE \). Both the \( name \) and \( date \) entry must be an element of \( birthday \) set. The last predicate actually removes the name and date mapping from the variable birthday using set subtraction. The details of Z schema calculus can be found in [134, 139, 50].

Z notation is not intended for specifying non-functional properties of a system such as reliability, performance etc. and same is the case for specifying timing and concurrent behaviour of a system [139]. However, it still can be modelled. It has been used for large system specifications and a large number of tools exist to support the Z notation. A descriptive comparison of some of the tools for Z notation is given in the Z survey [86]. An open source project called Community Z Tools (CZT) provides an integrated toolset for Z notation. The CZT is built using Java and also provides support for Z extensions.
This schema specifies a state space in which $x_1, \ldots, x_n$ are the state variables and $S_1, \ldots, S_n$ are expressions from which their types may be systematically derived. Z types are sets – $x_1, \ldots, x_n$ should not occur free in $S_1, \ldots, S_n$, or if they do, they refer instead to other occurrences of these variables already in scope (e.g., globally defined variables).

$Inv(x_1, \ldots, x_n)$ is the state invariant, relating the variables in some way for all possible allowed states of the system during its lifetime.

Note that unlike in an ordered tuple, the variables in a schema are essentially unordered – reordering them would give the same schema – and the variable names do not come into scope until the bottom half of the schema. Thus any interdependencies must be defined here. (A common mistake by initial users of Z is to define interdependencies in the declaration part, but a type-checker will very quickly detect this.)

3.6.1 Example specification

The ‘Birthday Book’ is a well known example from Chapter 1 of Spivey’s widely used book on Z [381]. It is a system for recording birthdays. It uses the following basic types (or given sets):

\[
\begin{align*}
\text{NAME}, \\
\text{DATE}
\end{align*}
\]

The state space of the Birthday Book is specified by:

\[
\begin{align*}
\Delta \text{BirthdayBook} \\
\text{name?} &: \text{NAME} \\
\text{date?} &: \text{DATE} \\
\end{align*}
\]

\[
\begin{align*}
\text{name?} &\in \text{known} \\
\{\text{name?} \mapsto \text{date?}\} &\in \text{birthday} \\
\text{birthday}' &= \text{birthday} \setminus \{\text{name?} \mapsto \text{date?}\}
\end{align*}
\]

Figure 2.5: Example of operational schema

\[
\begin{align*}
\text{BirthdayBook} \\
\text{known} &: \mathcal{P} \text{NAME} \\
\text{birthday} &: \text{NAME} \mapsto \text{DATE} \\
\text{known} &= \text{dom} \text{birthday}
\end{align*}
\]

Figure 2.6: BirthdayBook schema [47]

\[
\begin{align*}
\text{module MODULE-NAME} \\
\text{...} \\
\text{definitions types} \\
\text{...} \\
\text{state} \\
\text{...} \\
\text{functions} \\
\text{...} \\
\text{operations} \\
\text{...} \\
\text{end MODULE-NAME}
\end{align*}
\]

Figure 2.7: Structure of a VDM module

such as Object-Z [130], Circus [141], TCOZ [107], etc. Other tools for Z specification include ProofPower [12], Z/EVES [122], CADiZ [136] and Formaliser [106], etc.

2.3.2 VDM

VDM (Vienna Development Method) [91, 76, 28, 16] originated in 1970’s at the IBM labs in Vienna. It is one of the most mature and longest-established formal methods for specification of software systems [98]. The original VDM specification language (VDM-SL) [90] was standardised by the International Organisation for Standardisation (ISO) in 1996 [116, 99]. The VDM-SL has been extended to VDM++ [74], which supports modelling of object-oriented and concurrent systems.
module BIRTHDAY
parameter types NAME, DATE
export operations
REMOVEBIRTHDAY : NAME × DATE
definitions
types
BIRTHDAY = NAME → DATE
State
state of birthday : BIRTHDAY
  init (mk_State(birthday0)) ≡ birthday0 = {}
end;
Operations
REMOVEBIRTHDAY(name:NAME, date: DATE)
ext wr birthday : BIRTHDAY
Pre  name ∈ dom birthday
    date = birthday(name)
Post  birthday = birthday - {name → date}
end BIRTHDAYBOOK

Figure 2.8: VDM example

VDM uses set-theoretic types to represent system data and provides the refinement mechanism for transforming an abstract model to a concrete one and then to executable code (Java, C++). Refinement includes data refinement and operations decomposition at each refinement step. VDM has been widely used in a variety of domains and has extensive commercial and academic tool support. Some of the tools for VDM include VDMTools [17], SpecBox (a free tool for checking VDM specifications) [14], Overture [10] etc.

VDM specification is represented using modules and the structure of a VDM module is shown in Figure 2.7. It contains the types to be used in the module, state variables, initialisation of state, functions (without arguments) and operations (with arguments). An example module for the birthday book is shown in Figure 2.8. There are two types, i.e., NAME and DATE. A variable birthday is used to store birthdays, which is an initialized empty set. An operation for removing birthday entries is specified to take name and date as the input arguments. If the birthday database contains the given name/date entry, it will be removed as specified in the post statement. The symbol denotes the previous state of the birthday variable in the post statement.
2.3.3 B-Method

“B is a method for specifying, designing and coding software systems” [20]. It is a tool-supported formal method based on AMN (Abstract Machine Notation), used in the development of computer software. B was originally developed by Jean-Raymond Abrial in France and the UK. It is related to the Z notation (also originated by Abrial and others) and supports development of programming language code from specifications. B, inter alia, has been used in the development of several large safety-critical system applications (such as the Paris Metro Line 14 [42], the VAL shuttle for Roissy Charles de Gaulle airport [40] etc.), and is attracting increasing interest in the industry. B Method is also expressed as: “a proven construction approach (referred to as correct) based on B Language, refinement and proof” [11].

B development starts with specifying the system using an abstract machine, which utilises a number of mathematical concepts such as sets, relations, functions, sequences and trees. The abstract machine is then refined in a number of ways to gradually introduce more details of the system design. Refinement proof obligations (POs) are generated at each refinement step ensuring that it is a correct refinement of the abstract machine. During the process of refinement, the user interface (operations accessible to the user) remains the same as that of the abstract machine. The final refinement, also called ‘implementation’ (concrete machine), can then be translated into executable code in one of imperative programming languages. The notion of refinement from abstract model to concrete model and then to a specific language also helps in resolving portability issues across different platforms.

The structure of a B machine is shown in Figure 2.9. The machine has a name. It may include other machines using INCLUDES, SEES or USES clause. The INCLUDES clause provides read and write access to the state space of the included machine. The SEES clause provides read-only access to the machine that includes it. The USES clause is a generalisation of the SEES clause which provides read-only access, and state space of the included machine can be used within the invariants of the including machine [125]. The IMPORTS clause is another construct allowed in B, which links implementation (final refinement) to the abstract machine. The types or sets are represented in the SETS construct. Constants are identified in the CONSTANTS section. Any properties of the constants and sets are defined in the PROPERTIES clause. The dynamic data or variables are specified in the VARIABLES clause and the type of the variables, constraints or conditions on the variables are given in the INVARIANTS clause. A variable must have an invariant, at least to specify its type. All the invariants must remain true before and after the execution of any operation. The initial state of all the variables is given in the INITIALISATION clause. All the operations of a machine are provided in the OPERATIONS section. A part of the birthday book example used in the above section is specified in Figure 2.10 using the B language.
Generally, there are many ways to model a system but the goal is always to verify that the system is correct. This means that the system specification does not allow for any inconsistencies or ambiguities. This can be ensured by demonstrating that invariants always hold and refinements always correctly simulate abstractions. This is achieved with the help of tools such as animators, model-checkers and theorem provers. B-Method has good commercially available tool support for specification, design, proof and code-generation. Some of the tools available for B Method include ProB [101], Atelier B [65], B4free [64] and B-Toolkit [103] etc. Below is the brief introduction of ProB tool that we used for learning the B-Method.

2.3.3.1 ProB Tool

ProB [101] is an animator and model checker for the B-Method. It allows animation and automatic model-checking of B specifications, and can be used to systematically check a specification for errors. It was developed as a part of the research conducted within the EPSRC funded projects and continued within the EU funded project RODIN\(^1\). ProB has been successfully used on various industrial specifications (e.g., from Nokia Research) [15].

ProB has limited support for multiple machines and refinements. ProB can also be used for automated refinement checking. It also supports CSP [85] process descriptions to

\(^1\)RODIN - Rigorous Open Development Environment for Open Systems: EU Project IST-511599. http://rodin.cs.ncl.ac.uk
Chapter 2 Literature Review : Formal Methods

Figure 2.10: Example of a B Machine

MACHINE birthdayBook
SETS
  NAME; DATE
VARIABLES
  known, birthday
INVARINTS
  known ∈ NAME
  birthday ∈ NAME ↔ DATE
INITIALISATION
  known := {} || birthday := {}
OPERATIONS
removeBirthday(name, date) =
  PRE
    name ∈ NAME ∧ date ∈ DATE ∧
    name ∈ dom(birthday) ∧
    date ∈ ran(birthday) ∧
    date = birthday(name)
  THEN
    birthday := { name } ≼ birthday
END
END

guide B machines, or for property validation. The state space of the B machines can be graphically visualised. The state graph can represent the running states of the machine for a particular set of animations.

2.3.4 Event-B

Event-B [19] is a successor of the B-Method invented by Jean-Raymond Abrial and others. “It is defined in terms of a few simple concepts that describe a discrete event system and proof obligations that permit verification of properties of the event system” [24]. Event-B shares many concepts with the B-Method but is different in terms of the scope of modelling. The B-Method is primarily used for the development of correct by construction software whereas the Event-B can be used for the development of entire systems which includes hardware, software and the environment. Event-B and the B-Method
have commonalities but Event-B enables system-level modelling. It is also important to mention that most of the tooling for the B-Method was not open-source and hence it was difficult for the users to modify and extend the language and tools. Therefore, it was a need to have a language that could provide almost all the functionalities of B-Method and much more with extensible and open-source tooling framework.

An Event-B model consists of a machine and a context. The machine represents the dynamic part of the model whereas the context contains the static part. Figure 2.11 shows the structure and relation of Event-B machine and context. The machine structure in Event-B is similar to the Action System [38] formalism where an action system performs some guarded actions on the state variables.

The context contains sets, constants, axioms and theorems. Sets are used to define the types and axioms describe the properties of the constants. An Event-B machine contains variables, invariants, events, variants and theorems. Invariants specify types of the variables and certain conditions that should be maintained throughout the machine execution.

The state transition mechanism is accomplished through events which modify the machine state. An event can have guard predicates which must be true in order to enable the event to perform actions, e.g., assignment etc. Event parameters (also known as local variables) express input to, output from and local choice data within the event, as required. Variables are initialised by a special event called Initialisation which is unguarded. An event has the following syntax:

\[ e = \text{any } t \text{ where } G(t,v) \text{ then } A(v,t) \text{ end} \]

An event \( e \) having parameters \( t \) can perform actions \( A \) on variables \( v \) if the guards \( G \) on \( t \) and \( v \) are true. A model is said to be consistent if all events preserve the invariants, i.e., the invariant predicates must be true after any state updates. These invariant
preservation properties, called proof obligations (POs), are the verification conditions automatically generated by the tool and then discharged using theorem provers to verify correctness of the model.

During the refinement process, existing events can be refined and new events can be added to the refined machine. The new events should refine an implicit skip, which means that these events should not modify the abstract state variables. The new events must not diverge altogether. This can be ensured by defining a variant which makes sure that the model eventually comes to a terminating state and not run forever. The convergent events introduced in the refinement must be proved to decrease variant. POs are generated to verify all this. Theorems must be proved to follow from axioms and help in proving the model.

A machine can see more than one context and similarly a context can be seen by more than one machine. The machines can be refined and the contexts can be extended when moving from the abstract to the concrete models as shown in Figure 2.12 [19].

Event-B specifications can be refined in two ways, i.e., data refinement and event refinement. Usually the refinement process involves both of these. In data refinement, the state of the abstract machine is refined by introducing more variables and constants in the refined machine and context respectively. The state of the abstract and refinement machine is linked using invariants called gluing invariants. In event refinement, an event is refined by one or more events. The refinement process can also be classified as horizontal and vertical as discussed earlier and the application of both types of refinement can be seen in Chapters 5 and 6.

An Event-B machine for specifying the birthday book example is shown in Figure 2.13 (left). The machine has a variable birthday, which maintains the birth date for a given name specified using a partial function from the sets NAME to DATE. The sets are specified in the context seen by the machine as shown in Figure 2.13 (right). The variable birthday is initialised to an empty set in the ‘Initialisation’ event. We have another event removeBirthday for removing a birthday entry from the birthday variable. It has guards...
Figure 2.13: Event-B example of Birthday Book

to make sure that the given name and date entry is present in the variable `birthday`, which is then removed from it. Detailed examples of using Event-B will be discussed in our case-studies in the later chapters. The Rodin [55] tool has been specifically developed for modelling and verifying systems using Event-B language and is briefly introduced below.

2.3.4.1 Rodin

The Rodin [55] tool is used to automate modelling and verification of systems specified in Event-B language. It is based on an open source Eclipse [2] IDE using Java programming language and also provides the facility of extending it by developing more plug-ins.

Rodin has a built-in static checker, which checks for any static errors in the Event-B model when it is saved. The user is provided with information if there is any error and where it occurred. Various types of proof obligations are automatically generated by Rodin after a model is saved. There are different provers integrated into Rodin. The automatic prover tries to prove the proof obligations. The ones which can not be proved by the automatic prover can be proved interactively by the user with the help of the tool. Rodin provides good support for refinement and each refinement step can be verified.
using the prover by discharging the proof obligations. It has an improved user interface since it conforms to the GUI for a well established Eclipse IDE.

There are several plug-ins that are already available as part of the Rodin toolset for modelling, animation, visualisation, documentation and code generation etc. These are continuously being improved by the Rodin community and additional plug-ins are being developed as the research progresses. We have also developed a plug-in for the Rodin tool, as part of this research, which is explained in Chapter 4. We have modelled two case-studies using the Rodin tool and Event-B which are presented in later chapters.

2.4 Brief Comparison of Event-B, B & Z notation

All of the three formal methods were invented by Abrial and others. The Z notation was introduced in early 70’s and the B-Method in 80’s whereas Event-B is a recent formalism resulting from the experience of earlier ones along with CSP [85] and Action Systems [38] (briefly discussed in the next section). The concept of refinement is present in Event-B, B and Z notation. Event-B uses some of the notations of B-Method but has a different structure. While a B model contains all the information in a single machine, it is separated in to a context and a machine for specifying static and dynamic behaviour of the system being modelled. Event-B is an extensible language and provides a richer notion of refinement compared to B. For example, an operation in a B machine can be refined by only one operation whereas in Event-B an event can be refined by multiple events and new events can be introduced to model complex systems. The absence of complex structuring mechanisms of B (i.e., INCLUDES, SEES, USES, IMPORTS) in Event-B reduces modelling complexity but removes modular structuring mechanisms. The Rodin (tool for Event-B) is open source and can be easily extended as compared to the tools for B.

Z, B and Event-B are based on the same underlying set theory. Z remains rather strictly a specification notation whereas B is “wide-spectrum”, in that it has imperative programming constructs as part of the notation. This allows refinement down to code within the same semantic framework. The layered development notion of Event-B/B allows a complex development to be decomposed in a rich variety of ways using a small number of basic constructs. This makes the refinement of industrial-scale systems practical. Although many commercial quality tools are available for B and Z, there is limited tool support for refinement and proofs of Z specifications except ProofPower [12]. Event-B/Rodin is also becoming more popular with continuous improvements and enhancements.

Z notation is more focused on readability through the use of ‘schema inclusion’ where schemas defined separately can be used within other schemas making them precise and
readable. B language allows using entire machines into other machines through INCLUDES, SEES and USES clauses. Event-B does not provide any inclusion mechanism apart from separation of static and dynamic parts. This becomes difficult in understanding larger specification easily at lower levels of abstraction. This issue can be dealt with using the decomposition mechanism of Event-B discussed below.

Z notation specifies input and output variables separately and B method allows input and output parameters in its operations. Event-B does not have any specific mechanism for specifying input and output variables.

2.5 Other formalisms

2.5.1 CSP

CSP (Communicating Sequential Processes) [85] is a formalism for modelling concurrent systems, devised by C.A.R. Hoare in 1978 [84]. It is a well-known event-based approach for modelling concurrency through communicating processes and uses a set of events, basic processes, and CSP operators. In a system specified in CSP, processes communicate with the environment through a number of atomic events. For example, a process that engages with an event $e$ and behaving as $P$ can be shown by $e \rightarrow P$ where the operator $\rightarrow$ describes the sequencing of events.

The choice operator $\Box$ can be used to describe external choice of behaviour. For example, $P \Box Q$ represents the process that offers the choice to the environment between behaving as $P$ or as $Q$. CSP also has a separate nondeterministic choice operator $\sqcap$, e.g., $P \sqcap Q$ represents the process that internally chooses between behaving as $P$ or as $Q$ and the environment has no control over the choice.

Following is a simple example of specifying a system using CSP. A vending machine VM (discussed in [85]), which can either vend coffee or tea after the user inserts a coin, can be specified in CSP as:

$$VM = coin \rightarrow tea \Box coffee$$

The operator $\Box$ gives a choice to the user to select the drink and the statement above can be read as “coin then (tea choice coffee)”.

2.5.2 Action System

The Action System [38] formalism was introduced by Back and Kurki-Suonio in early 80’s [39], which can be used to specify the global behaviour of parallel and distributed
systems. It is a general formalism which can be used to provide syntactic description to transition systems. The behaviour of a system is specified in terms of actions that are performed by various processes of the system in cooperation with each other. An action system statement can be represented using the command language in the following form [36, 34]:

\[ A = [\text{var } x; S_0; \text{ do } A_1[...[A_m \text{ od}]] : z. \]

Here, \( x \) represents the local variables of the action system \( A \) and global variables of the system are represented by \( z \). Both local and global variables are distinct (i.e., \( x \cap z = \emptyset \)). \( S_0 \) shows the initialisation statement of the system. Actions of the system are represented by \( A_1 ... A_m \). An action can be represented in the form of:

\[ A_i = g_i \rightarrow S_i \]

where the guard of an action \( A_i \) is shown by \( g_i \) and \( S_i \) shows the body or statement of the action which is executed once the action is enabled after its guard becomes \( \text{true} \). Enabled actions of the system are non-deterministically selected for execution and independent actions of an action system can be executed in parallel as far as they do not share the state with other actions of the system (i.e., do not have any shared variables). Actions are atomic and their execution must be completed without any interference from other actions. An abstract action system can be refined to a concrete implementable system using stepwise refinement approach. [36, 35, 37]

Action systems can be composed using parallel composition [51]. For example, consider the following two action systems:

\[ A = [\text{var } x; S_0; \text{ do } A_1[...[A_m \text{ od}]] : z. \]
\[ B = [\text{var } y; T_0; \text{ do } B_1[...[B_m \text{ od}]] : w. \]

Their parallel composition, provided that \( x \cap y = \emptyset \), may result in:

\[ C = [\text{var } x; S_0; y; T_0; \text{ do } A_1[...[A_m[...[B_1[...[B_m \text{ od}]]]]] : z \cup w. \]

The composed action system \( C \) combines the state space of both action systems \( A \) and \( B \) where global variables are merged and local variables of the actions are kept distinct.
2.6 Composition

Software composition plays an important role when it comes to reusability and provides advantages of modularity. Software components developed separately can be reused in building different applications and their composition has always been a difficult task. Smaller components that make up large and complex systems are easier to test and maintain compared to the whole system. There is a lot of research being done in this area and the software composition research community is quite active. The software composition conference (ICSC) has been running yearly since 2002. The conference brings together the researchers to discuss interesting results and future directions for software composition to improve software engineering practices.

Composition is equally important within our formal methods domain. Below is the discussion on how composition works in some of the formal specification languages. Our approach towards composing Event-B specifications is given in Chapter 4.

2.6.1 Composition in B, Z & VDM

Composition in the B-Method can be achieved by using one of the four primitives provided by the language itself [120]. These are INCLUDES, IMPORTS, USES and SEES as discussed in Section 2.3.3. The B-Method also provides an operator for parallel composition (∥). This operator can be used with inclusion primitives where pre-conditions for the operations are conjoined and action statements are executed in parallel as shown below. Here \( P_1 \) and \( P_2 \) are pre-conditions and \( S_1 \) and \( S_2 \) are the actions performed by the operations.

\[
\text{PRE } P_1 \text{ THEN } S_1 \text{ END } \parallel \text{PRE } P_2 \text{ THEN } S_2 \text{ END} \equiv \text{PRE } P_1 \land P_2 \text{ THEN } S_1 \parallel S_2 \text{ END}
\]

The composition mechanism in Z notation can be achieved through the use of schema inclusion. Schemas can be included in other schemas to achieve composition and reusability. This is discussed in Section 2.3.1, where a RemoveBirthday operation includes the BirthdayBook schema. The schemas being composed must be type compatible which means that the common variables must be of the same type. There are two operators used for schema inclusion, i.e., \( \Delta \) and \( \Xi \). The inclusion operator \( \Delta \) allows changing the state space of the included schema whereas the operator \( \Xi \) allows read-only access of its state space to the including schema. Schema calculus [50] can also be used to compose Z schemas by using predicates over schemas. For example, two schemas \( S \) and \( T \) can be combined using ‘\&’ as \( S \land T \). We can also combine schemas using the sequential composition operator ‘;’, e.g., \( S \) and \( T \) can be sequentially composed by expressing as
Chapter 2 Literature Review: Formal Methods

S; T, provided the shared state of the two schemas is type compatible. The piping operator ‘>>’ can be used to combine S and T as S >> T if the two schemas are acting on disjoint state variables.

The composition in Z notation can also be done by using views [87, 45]. A view is considered as a partial specification derived from the requirements and several views are composed to build the entire specification. A view consists of state space and a set of operations and the composition of views is complemented by linking the state space using invariants.

Composition in VDM has similarities with the Z and B-Method. Operations defined in a module can be used by other modules if these are explicitly marked as such. If a function defined in a VDM module with an ‘export’ primitive, it can be used in another module using an ‘import’ command. This allows parts of a module to be visible outside the module and could be reused while composing this with any other module.

2.6.2 Decomposition & Composition in Event-B

2.6.2.1 Decomposition in Event-B

When a model becomes too big to be easily refined or team efforts are required, we need to decompose it into various sub-models (components), which can then be refined independently. In effect, this is complexity management by reducing the size of models, which keeps them understandable and reduces the number of POs to be proved for each model. This also allows the refinement of components in parallel by different teams. There are two types of decomposition in Event-B known as shared-variable decomposition (SVD) [25] and shared-event [52] decomposition (SED). Like the Event-B language, these techniques are influenced by earlier formalisms such as CSP [85] and Action Systems [38]. The refinement preserving nature of these decomposition techniques differentiates these from the feature-based decomposition within the feature-oriented software development (FOSD) community. This is due to the fact that FOSD is generally aimed at the implementation level components, usually written in a high-level programming language. Also, the top-down modelling approach of Event-B (which focuses on abstraction and step-wise refinement) is different from decomposition in FOSD as it could take place at various levels of refinement depending on the architectural design of the system.

In shared-variable decomposition style, shared variables are kept in all the components, and events are partitioned between components. Each shared variable \( v \) in each component \( C \) is affected, i.e., by possible transitions defined by every event \( e \) in every other component acting on that variable. To model this, for each such event \( e \), an external event \( e_{ext} \) is added to \( C \), which is an abstraction of \( e \) that contains the predicates defined over the shared variable, abstracting away from the local effect of \( e \). When a component is refined, shared variables and external events must not be refined. This type
Figure 2.14: Shared-variable decomposition

of decomposition corresponds to asynchronous shared-memory communication between components. Figure 2.14 is an example of SVD where machine M is decomposed, with shared variable v2, by partitioning events into machines M1 and M2. Thus event e3', a new external event in M1, models the effect on v2 of e3 in M2. Similarly, e2' is an external event in M2 modelling the effect on v2 in M2 of e2.

Figure 2.15: Shared-event decomposition

The shared-event style is based on shared events rather than shared variables. During the decomposition, variables are partitioned between components and shared events are split. A shared event is the one accessing variables residing in different components. Figure 2.15 (right) is an example of SED where machine N is decomposed by partitioning variables into machines N1 (with v1) and N2 (with v2). Since event e2 works on variables
v1 and v2, it will be split between N1 and N2. The part of event e2 (e2a) that deals with variable v1 becomes an event of N1 and its other part (e2b) that deals with v2 becomes event of N2. Event splitting is achieved by decomposing its parameters, guards and actions into two parts. This type of decomposition is considered appropriate for systems based on synchronous message passing.

Both the SVD and SED approaches have semantic support for modular refinement. This means that it has been shown for both approaches that decomposition preserves refinement: if we were to recompose components, even after further refinement steps, the composite would refine the single abstract model.

In practice, the designer might choose to recompose - e.g., all code to run on a single processor - or might not - e.g., where component models are deployed on separate physical devices. The key point is that the final model is ‘correct by construction’. A decomposition plug-in [129] has been developed for the Rodin tool which can be used to demonstrate both styles of decomposition.

2.6.2.2 Composition in Event-B

Since we are interested in composition, we would like to use the decomposition styles discussed above by inverting the decomposition method. For the shared-event style, composition is straightforward. Whether one is composing all or just a subset of components, we can compose these provided these do not have any shared state. For the shared variable style, composition works as an inverse of decomposition. It can only be applied if the components being composed are a result of SVD otherwise the components or their refinements may not be dealing with shared variables in the same way. This is the reason for having external events to mimic the same behaviour on the shared state across all components. Hence, this brings up a tooling requirement to automatically generate external events for the components being composed. We could manually do this but it will be cumbersome and even more difficult when composing large number of components with many events. We will discuss this further in our case-studies later.

![Figure 2.16: Fusion composition example](image)
Event-B models can also be composed using Fusion, suggested by Poppleton [119]. This style of composition, inspired by the above two composition styles, promises the support for reuse of models through composition as envisaged in feature-based development. This allows composing Event-B models having shared state by merging events. During the event fusion, guards are conjoined and actions are concatenated, with possible renaming of the composite event with a common name. For example, consider two Event-B models $M$ and $N$. The model $M$ has variables $v_1$ and $v_2$ with invariants $I_m$ and event $E_m$ whereas the model $N$ contains variables $v_2$ and $v_3$ with invariant $I_n$ and event $E_n$. Both models share the variable $v_2$. Their composition yields model $M$ (as shown in Figure 2.16), having invariants from both the models and the fused event $E_p$ as $E_m \odot E_n$. The event fusion is shown below where guards ($G_1, G_2$) acting on parameters ($\alpha, \beta$) and variables are conjoined and actions ($X, Y$) are concatenated:

\[
E_m = \text{any } \alpha \text{ where } G_1(\alpha, v_1, v_2) \quad \text{then } X := F_1(\alpha, v_1, v_2) \\
E_n = \text{any } \beta \text{ where } G_2(\beta, v_2, v_3) \quad \text{then } Y := F_2(\beta, v_2, v_3)
\]

\[
E_m \odot E_n = \text{any } \alpha, \beta \text{ where } G_1(\alpha, v_1, v_2) \wedge G_2(\beta, v_2, v_3) \quad \text{then } X := F_1(\alpha, v_1, v_2) || Y := F_2(\beta, v_2, v_3)
\]

The refinement preservation is also guaranteed by this style. This means that the fusion of refined events correctly refines the fusion of abstract events as shown in Figure 2.17. The model $Pr$ resulting from fusion of models $Mr$ and $Nr$ (which are refinements of $M$ and $N$ respectively) correctly refines model $P$ which is a fusion of $M$ and $N$. This is constrained by the fact that the shared variables must be refined in the same manner across all the models being composed.

2.6.3 Related Work

2.6.3.1 Generic Instantiation

Generic Instantiation [128] provides a reuse approach for Event-B models. The generic Event-B developments (called patterns) are instantiated to specific models through parametrisation achieved by refactoring. Such a pattern can include a refinement chain. This means that machine elements (e.g., variables, event names etc.) can be renamed and context parameters (e.g., sets, constants etc.) can be replaced with elements of the same type. This helps in reusing patterns to build models having similar characteristics and also maintains the validity of already discharged POs through to the instantiated models.
Figure 2.17: Refinement in fusion composition

Figure 2.18: Instantiation of a generic chain of refinements [128]

Figure 2.18\(^2\) shows an example of the generic instantiation for a generic pattern as a refinement chain (P\(_1\) to P\(_m\)). A development D\(_1\) is refined by D\(_n\) which can be refined by instantiating pattern P\(_1\) through a user provided context resulting in D\(_{n+m_{\text{abs}}}\). The final refinement of pattern (P\(_m\)) can be instantiated to a specific development (D\(_{n+m}\)) which turns out to be a refinement of D\(_{n+m_{\text{abs}}}\). Since the pattern has already discharged all the refinement POs, there is no need to prove the refinement POs for the instantiated developments (i.e., between D\(_{n+m_{\text{abs}}}\) and D\(_{n+m}\)). The final refinement

\(^2\)The instantiation arrow points in opposite direction to that of the figure in [128]
of instantiated development \((D_{n+m})\) can be further refined as usual. This work shows the reuse of generic patterns and the refinement POs. There is no tool support for generic instantiation as of yet and we suggest similar approach in our feature-oriented modelling framework discussed later but that requires more than just refactoring while instantiating Event-B developments.

### 2.7 Conclusion

We have given an overview of formal methods and the benefits of using these for building software and hardware systems. We have briefly discussed a number of formal specification languages such as Z, VDM, B-Method and Event-B with the help of simple examples and the tool support available for each of these. Event-B is a recent formal method with extensible and robust tool support and a variety of composition and decomposition techniques. The refinement-driven and proof-based verification approach of Event-B provides scalability of architectures and also supports modelling and verification of distributed systems. The extensible nature of the language and its tools with an active research community and industrial partners, who are constantly working on improving it, makes it a good choice for feature-oriented modelling and reuse, as explained in later chapters.

In order to achieve our objective of introducing feature-oriented reuse in Event-B, it is important to explore the composition approaches within formal methods. We have also summarised the two types of (de)composition mechanisms currently available for Event-B and we will use these in our case-study examples later to see if these are enough to do feature-based modelling in Event-B. Although the related work on Generic Instantiation [128] provides us some guidelines for our reuse approach, we will need a more flexible approach discussed later, for feature reuse that could facilitate product lines modelling in Event-B.
Chapter 3

Literature Review : MDA, Product Lines and Feature Modelling

3.1 Introduction

This chapter provides an overview of Model-Driven Architecture (MDA), Software Product Lines (SPLs) and Feature Modelling. MDA will be used while designing and implementing our prototype tools to experiment with our case-study examples. We discuss eclipse-based modelling frameworks that will be used during the tool development and various transformation languages. Feature modelling is a well known technique for modelling SPLs and we will use this to introduce feature-based reuse approach in Event-B. Model-driven SPL development is gaining more recognition and it provides the advantages of generative techniques [68].

3.2 Model-Driven Architecture (MDA)

3.2.1 Introduction

MDA is a software development framework defined by Object Management Group (OMG). Software development in MDA is driven by the activity of modelling the system [95]. It is different from the traditional software development because it uses models in the software development process, i.e., the major difference is in the artefacts produced. These artefacts in MDA lifecycle are formal models which can be understood by the computer. Figure 3.1 shows the MDA software development life cycle. The core models within MDA are PIM, PSM and Code (explained below).
PIM is a *platform independent model* which is designed at a higher level of abstraction and specifies the system to understand its requirements. PIM is modelled without considering any implementation technology. Within a PIM, the system is modelled from the viewpoint of how it best supports the business. Whether the system will be deployed on a desktop computer or a mainframe plays no role in a PIM. An abstract model specified in Event-B is an example of PIM which does not consider any implementation level details.

PSM stands for *platform specific model* and within MDA PSMs are automatically generated from PIM using the transformation tools. A PIM may be transformed into one or more PSMs, one for each specific technology. PSM is tailored to specify the system in terms of implementation constructs that are available in one specific implementation technology. Since most of the systems these days are implemented on different platforms, several PSMs exist for each PIM. For example, a PIM for a browser will have PSMs tailored for Microsoft Windows, Macintosh and Linux operating systems. Similarly, an Event-B abstract model (PIM) can be refined in multiple PSMs for different platforms.

The final step in the development of software within MDA is the transformation of PSM to executable code. This transformation is relatively easy comparing to the one from PIM to PSM. It is because PSM is already closer to the implementation technology. Several
formal specification languages allow refinement to produce a concrete transformation of an abstract model that can be easily converted to executable code, e.g., Java [81].

MDA defines the relation between these models and how these can be transformed from one to another automatically through the tools to solve the software development issues of portability, productivity, interoperability and maintenance. MDA principles can also apply to other areas such as business process modelling where the PIM is translated to either automated or manual processes.

MDA requires models to be expressed in a language based on MOF (Meta Object Facility). This guarantees that the models can be stored in an MOF-compliant repository, parsed and transformed by MOF-compliant tools, and rendered into XMI (XML Metadata Interchange) for transport over a network. The types of models you can use are not constrained by this. MOF-based languages today model application structure, behaviour (in many different ways), and data. OMG’s UML and CWM (Common Warehouse Metamodel) [117] are good examples of MOF-based modelling languages and there are other languages apart from these as well. [8]

3.2.2 Benefits of MDA

Following a long history of the use of models to represent key ideas in both problem and solution domains, MDA provides a conceptual framework for using models and applying transformations between them as part of a controlled, efficient software development process. Models help people understand and communicate complex ideas and analysing the problem domains. Many different kinds of elements can be modelled, depending on the context. The popularity of UML in the software industry shows how MDA can be beneficial in software engineering. In general, MDA can help in improving the following aspects throughout the software development process [95].

**Productivity:**

Productivity gain through MDA can be achieved by using tools to automate the transformation from PIMs to PSMs and from PSMs to code. This means that much of the information about the application must be incorporated in the PIM or the generation tool. Although the developers will need to put more effort in writing the transformations and detailed PIMs, generally it is worth the efforts invested in it in the long-run, e.g., reduces maintenance efforts and costs. In general, productivity is improved in two ways. Firstly, PIM developers will be doing less work as platform specific and other technical details are not considered and they will need to write very little or ideally no code. Secondly, by focusing more on the PIM and thus paying more attention to solving the business problem at hand.

**Portability:**
Portability is obvious from the definition of PIM as it can be transformed into various PSMs for different platforms. Thus the specification at PIM level is completely portable. This also depends on the robustness of the tools that make the actual transformation from PIM to PSM.

**Interoperability:**

Different PSMs for a particular PIM might need to interact with each other for correct functionality. For example, a product module running on one operating system needs to communicate with another module of the product running on a different operating system to perform various tasks. In MDA, this relationship is called as a bridge. Defining this bridge is not complex as we already know how the constructs of a PIM map to elements of different PSMs, otherwise it's not the correct PIM to PSM transformation. Cross-platform interoperability for applications can be achieved through tools that also generate the bridges along with the PSMs from a particular PIM. This also helps in handling the issues of technology changes that occur at different platforms.

**Maintenance:**

In MDA, PIM serves the purpose of high-level documentation of the system. PIM is not abandoned even after the generation of the PSMs and code and any change taking place must go through the PIM. Otherwise the MDA may not be able to serve its purpose properly. This also requires that the transformation tools should keep all the information about the decisions that take place so that the same can be done when regenerating the PSM from PIM after making the changes and so on to the next level of PSM to code.

### 3.2.3 Eclipse Modelling Framework (EMF)

The Eclipse Modelling Framework, a part of Model Driven Architecture (MDA), is a Java based framework which provides facility for designing and implementing structured models. It also helps in generating efficient, correct, and easily customisable Java code from these models to build tools and other applications. EMF focuses on the key concepts of meta-data, code generation and serialisation. It helps in increasing productivity and facilitates interoperability with other EMF-based tools and application. [3, 110]

EMF consists of three fundamental parts, i.e., EMF Core, EMF Edit and EMF Codegen. The core framework presents a metamodel (Ecore) which provides model description and runtime support for any changes occurring in the models and persistence through default XMI serialisation. It also provides an efficient API for dealing with EMF objects. EMF Edit provides facility for building EMF model editors through generic reusable classes. These include content and label provider classes, property source support and other classes that allow displaying EMF models using standard desktop viewers and property sheets. The framework helps building rich editors with facilities such as automatic undo
EMF Codegen provides a code generator to build a complete editor plug-in for the EMF models with the facility of customising this code while maintaining its connection to the models. The code generation options can be specified using the GUI which is also used to invoke the code generators.

EMF models are defined using XMI. Thus EMF models can be produced through various ways such as writing the XMI document directly, exporting XMI document using modelling tools such as Rational Rose, annotating Java interfaces or through XML schemas. After having an EMF model, the generator can be used to create Java implementation classes for the model. These classes can be edited for adding further methods and properties and any additions will be preserved when regenerating Java code from the EMF models. If the additions depend on the part of the model that has been changed then this will require updating the additional code.

### 3.2.4 Graphical Modelling Framework (GMF)

The Graphical Modelling Framework (GMF) is an Eclipse modelling project which provides infrastructure for building graphical editors based on models. It provides a generative bridge between EMF (discussed above) and GEF (Graphical Editing Framework). The GMF project was inspired by the use of EMF and GEF, which became a common practice before the GMF project.

The GEF is a framework for building rich graphical editors from models of existing applications. It can be used along with EMF or other technologies. It consists of two plug-ins. The draw2D plug-in provides rendering facilities for graphical display and the GEF plug-in provides common functionalities for building the editors such as redrawing figures after a user action or dealing with event handlers etc. GEF is not application dependant and can be used or extended to build any application such as class diagram editors, activity diagrams, state-machines and WYSIWYG editors [7].

Figure 3.2 shows GMF development process which starts with creating a GMF project and an EMF model for the application domain. The graphical definition model is then created which specifies the graphical elements that will be provided in the editor and has no connection to the domain model directly. A tooling definition model can also be generated for designing palette, menus and tool bars etc. Since these two models are independent of the application domain, these can be reused for different applications requiring similar elements. A mapping model is then created that links the domain model to the graphical and tooling definition models. Based on the mapping model, a generator model is created and can then be used to generate the editor. The generated code can then be customised by the user and can be tagged not to be overwritten when the code is regenerated upon changing any of the model. This is important not to loose the customised code upon regeneration because rewriting it could involve lot of user
effort. Ideally, there should not be much customised code and any changes required must be done in the model rather than in the code as this would not then serve the benefits of MDD. But there could be situations where required changes can not be expressed in the model due to limitations of the modelling language and tools; hence the generated code then requires customisation. Usually, any conflicts arising from code regeneration can be picked up by the tool. In practice, automated tests are created to make sure that the model changes do not break the core functionality.

3.2.5 Model Transformation

3.2.5.1 Introduction

In the scope of model-driven architecture, model transformation is a key activity and aims to provide a means to specify the way to produce target models from a number of source models. There are various ways to carry out model transformation, for example, refinement is a form of vertical transformation, refactoring is a kind of horizontal transformation [108] and composition is also a form of transformation which aims at extending the existing functionality. Literally, anything that takes one or more models as input and outputs one or more models with some modification which then serves some purpose can be called a model transformation activity. The importance of this activity is quite obvious as discussed above and is at the heart of MDA.

In model-driven architecture, a model is defined according to the semantics provided by its metamodel, i.e., a model is said to conform to its metamodel. The metamodel
being a model itself has to be defined according to the semantics defined in the metametamodel. The meta-metamodel can be a metamodel for itself. Model transformation is also defined as a model itself and must conform to the transformation metamodel to validate that it is the right transformation. Figure 3.3 shows the model transformation process where a Model \( M_a \) conforming to its metamodel \( MM_a \) is transformed to Model \( M_b \) conforming to the metamodel \( MM_b \). Both \( MM_a \) and \( MM_b \) conform to the meta-metamodel \( MMM \). The transformation model \( M_t \) conforms to the metamodel \( MM_t \) which eventually conforms to the top level meta-metamodel \( MMM \). From the users’ perspective, they will provide \( M_a \) and will get \( M_b \) where the transformations will be defined by the developer of the transformation tool executed by the transformation engine. According to Batory [41], this is analogous to relational databases where a database conforms to its schema and the schema is based on the information schema. Here database corresponds to a model, schema to metamodel and information schema to a meta-metamodel. Database transactions are similar to transformations in MDA.

Model transformation plays an important role in Model-Driven Engineering (MDE) approach. It is expected that writing model transformation definitions will become a common task in software development. Software engineers should be supported in performing this task by mature tools and techniques in the same way as they are supported now by IDEs, compilers, and debuggers in their everyday work [92]. The main reason why we are interested in model transformation is because we want to use it for the development of our feature modelling tool for Event-B.
3.2.5.2 Transformation Languages

There are various ways of transforming models and different domains have different requirements for transformation. There are commonalities among transformation languages as all of them are developed to define the rules for carrying out the actual transformation. There are many transformation languages around, e.g., ATL [92], IBM’S MTF [9], VIATRA [18], GReAT [26], MT [137], SiTra [27] (a transformation approach using Java & C#) etc. It is impossible to choose the best language, since each one is serving the different purposes in its particular domain. Following are three model transformation approaches that we have investigated because they are Eclipse [2] based and are easier to integrate with the Rodin tool.

- **ATL**

The Atlas Transformation Language (ATL) is a hybrid of declarative and imperative programming. The preferred style of transformation writing is the declarative one as it allows a modeller to simply express mappings between the source and target model elements. However, ATL also provides imperative constructs in order to ease the specification of mappings that are hard to express declaratively.

An ATL transformation program is composed of rules that define how source model elements are matched and navigated to create and initialize the elements of the target models. Besides basic model transformations, ATL defines an additional model querying facility to specify requests onto models. ATL also allows for code factorization through the definition of ATL libraries. Thus the language enables us to define three kinds of ATL units: ATL transformation modules, ATL queries and ATL libraries. ATL modules help a developer in specifying the ways to produce target models from source models. The source and target models in a module must be typed through their respective metamodels. ATL module can only produce a fixed number of target models from a fixed number of source models. Therefore, the number of input and output models must be given. An ATL Query is used to compute primitive values from a number of source models. For example, the generation of an output string from the source models. The computation can be a Boolean or a numerical value as well. The structure of an ATL query is: `query query_name = expression;`. ATL library helps in defining helpers written in ATL which can be called from different ATL units. ATL libraries can only be used within other ATL units and cannot be executed independently.

Figure 3.4 shows a brief example of an ATL transformation program as given in [33]. Author2Person is the name of the ATL module. The `create` command specifies the creation of Person model from the given Author model. It transforms Author elements of the Author metamodel into Person elements of the Person metamodel. The source ‘a’ is of type Author!Author (i.e., metamodel ! model_entity) and the target ‘p’ is
of type Person!Person. The actual transformation copies the name and surname of source entity Author to the created target entity Person. The metamodels for both the Author and Person along with the Author model must be given when executing this transformation program to produce the Person model.

- Model Transformation Framework

IBM’s Model Transformation Framework (MTF) provides a set of tools for implementing transformations between EMF models, making comparisons and checking consistency among them. It has a simple declarative language for defining mappings between the models and a transformation engine to execute these mappings. MTF is distributed as Eclipse plug-ins including facilities to write, run and debug the mapping rules using Rule Editor, Debugger and Mapping View. The Mapping View enables to view the results of completed transformations. The mapping rules are also called relations. [9, 72]

MTF is flexible in transforming models and can accept, relate and update multiple models. The direction of transformation is defined when the transformation engine is invoked. This can be useful in dealing with round-tripping issues. The language to define the relations in MTF is called Relation Definition Language (RDL) which is parsed and evaluated by the transformation engine. RDL has a simple syntax which is composed of a few keywords and the standard Boolean operators. The relation is defined using the keyword ‘relate’ as shown in the transformation mapping rule of Figure 3.5 [72]. The signature contains the name (MyRelation) and the parameters (source & target). These parameters are constrained with conditions to filter matching values. The body contains the correspondences that are applied to matched values. The ‘equals’ keyword
relate MyRelation(s:S source, t:T target) {
    equals(source.name, target.name),
    MyOtherRelation(source.value, target.value)
}

Figure 3.5: MTF rule example

is one such correspondence that allows finding the equality of strings or other types. Other rules can be called from within the rules (e.g., MyOtherRelation).

Although RDL is a succinct language, it supports essential concepts of model transformation. The correspondence between elements expressed using the relation is dependent on the actual parameters supplied in a particular context. Hence, different results can be seen in different contexts. [72]

- **ETL**

The Epsilon Transformation Language (ETL) [96] is part of the Epsilon framework which comprises different inter-operable languages for transformation, validation, merging, comparison and code generation based on EMF models. All such languages are based on the core Epsilon Object Language (EOL), which makes it easy to interact with other languages in Epsilon. The Epsilon framework is Eclipse based and provides various tools to facilitate all the major tasks of model-driven development approach.

ETL is a rule-based language providing standard features for a model-to-model transformation language and also supports the transformation of many input models to many output models. It allows modifying, querying and navigation of both the source and target models and models of different metamodels. Some of its main features include automated rule execution, multiple rule inheritance, guarded rules and declarative rules with imperative bodies. All the usual programming constructs are also supported such as while and for loops, statement sequencing, variables etc. It also allows us to create and call Java objects. Support for caching and use of OCL constraints is also part of the EOL. Operations from different libraries can be reused even written in any Epsilon language other than ETL.

Figure 3.6 shows an example of a model to metamodel ETL transformation rule which transforms elements from a source model ‘FeatureModel’ of metamodel FM to a target model of metamodel EM (Ecore). For all the elements in the source model, it creates an ECore Class with the same name as the source element. It creates a ‘Cardinality’ attribute in the class if the element is of type ‘Group’ or ‘Feature’ and another attribute ‘IsGroup’ for elements of type ‘Group’. The properties for the attributes are assigned
rule FM2Ecore
  transform source : EMFFeatureModel
to target : EMFEPackage {

  for(f in FM.allInstances)
  {
    var ec : new EMFEClass;
    ec.name = f.name;
    target.eClassifiers.add(ec);
    
    if(f.isTypeOf(feature) or f.isTypeOf(Group))
    {
      var card : new EMFEAttribute;
      card.name = 'Cardinality';
      card.eType = EcoreEDatatype.all.
      selectOne(dt|dt.name = 'EString');
      card.lowerBound = f.min;
      card.upperBound = f.max;
      ec.eAttributes.add(card);

      if(f.isTypeOf(Group))
      {
        var isGroup : new EMFEAttribute;
        isGroup.name = 'isGroup';
        isGroup.eType = EcoreEDatatype.all.
        selectOne(dt|dt.name = 'EBoolean');
        isGroup.defaultValueLiteral = 'true';
        ec.eAttributes.add(isGroup);
      }
    }
  }
}

Figure 3.6: ETL example

accordingly. A rule can also have a pre and post blocks to execute any statements before and after the rule or a set of rules are executed. The actual models to be transformed and their metamodels are specified in the Eclipse runtime configuration used by ETL execution engine.

Discussion

We can use any of these languages for model transformation as far as our tool development is concerned, though there were some issues while using ATL and MTF. Mainly there was lack of support available if some aspects of the related tools could not be plugged in to the latest release of underlying development framework. We found ETL more useful than ATL and MTF in terms of documentation and online support available to the users. ETL’s handbook [96] provides detailed description of the language and its usage, and case-studies can be found online to facilitate its understanding. Moreover, online forums for ETL proved quite effective in answering user queries. We only switched
to ETL after trying out both the ATL and MTF. MTF is no longer maintained and does not work well with the latest releases of the Eclipse and its related EMF plugins. We also had problems while implementing the transformation rules using ATL for our Rodin plug-in which could not be resolved and we had to use ETL. Another important factor is the Epsilon’s other languages which could later be used to do other tasks such as validation etc.

### 3.3 Software Product Lines

#### 3.3.1 Introduction

Software product lines (SPLs) focus on the problem of software reuse by providing ways to build software products having commonalities and managing the variations within the products of the same family. “A software product line is a set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way” [66]. SPL engineering is useful in rapid development of similar products where core functionality is repeated among a number of related products.

The Eclipse platform can be considered as an example of a product line where the core assets (plug-ins) are common in any Eclipse-based product as a basic functionality. Variability is present in the form of additional plug-ins that can be packaged along with the core plug-ins to build a specific product. Common plug-ins could be the basic editors, navigator, project explorer, primary views and some of its products can have modelling functionalities, other with improved GUIs and different types of editors etc. The Rodin tool can be considered as a member of the Eclipse product line.

The concept of product lines has been around for a long time now and a lot of research has been done in this area. The benefits of reusing the common base to develop variations within the family of products are obvious and have been proven effective. These days research is being done on predicting the evolution of product lines to ensure that it can be incorporated into the architecture of the product line. [97]

#### 3.3.2 Commonality & Variability in SPLs

Commonality and variability are key concepts to consider when building product lines. These help in discovering what is common and different among various products of the product line. Commonality determines the common characteristics of the products in the product line and the variability identifies the variable characteristics among them. If we take the example of a set S as mentioned by Coplien et al. [67], commonality is the
assumption that all the elements of the set S will have the same value for a particular
attribute, whereas, variability assumes that two or more elements within S will have
different values for the same attribute. For example, a set of shapes containing a square
and a triangle will have ‘area’ as a commonality and the way the area is computed for
each shape, i.e., ‘area formula’ represents the variability [67]. Similarly, the ‘number of
sides’ in each shape of set S can be different and hence represent variability. A recent
comparison of variability modelling approaches for SPLs is presented by Czarnecki et
al. [69].

3.3.3 SPLs Engineering Approaches

There are two approaches for adopting product lines, i.e., proactive and reactive. In
the proactive approach (as in predictive), core assets are developed by predicting the
future products within the product line with built-in variation points. These core assets
can then be configured and composed to build the desired product. The advantage
of this approach is rapid development and short time of delivery of the product. The
disadvantage can be in the upfront efforts involved in building the core assets even if they
may not be required immediately. In reactive approach, the core assets are produced
incrementally and added to the product line whenever there is a need for a new product
in the product family. Although this approach has the advantage of reduced upfront
efforts, it increases the time to market. Sometimes it can be useful to combine the two
approaches, for example, taking the proactive one in the earlier phase of SDLC (e.g.,
architecture level) and reactive towards the end (e.g., implementation level) [13, 69].

There are also different approaches for specification of product lines. The decision
to select a specific approach varies from project to project and the demands of the
stakeholder along with other factors such as time, cost etc. These include use-case
modelling, object-modelling, feature modelling etc. [60]. The following section gives an
overview of the feature modelling approach that we have decided to use for investigating
product line reuse in Event-B.

3.4 Feature Modelling

3.4.1 Introduction

Feature modelling was proposed as part of the feature-oriented domain analysis (FODA)
method [93] in early 90’s and has been used in many domains, product line projects and
organisations, e.g., Microsoft is using this for their software factories [82]. “Feature
modelling is a technique for representing the commonalities and the variabilities among
a set of systems in concise, taxonomic form” [68]. In the early phase of software product
line (SPL) development, this can be used for deciding the scope of the product line, i.e.,
which features a product-family should support. In the design phase, it can be used to
map product variability to the product line architecture. In the actual product develop-
ment, feature modelling helps in requirements analysis and estimating costs and effort.
Feature modelling is also useful for generative software development where specification
of a product family in a textual or graphical domain-specific language can be used to
generate members of the SPL automatically. Hence it can also be used for scoping and
developing domain-specific languages [70].

Feature modelling provides means to organise features into a feature model and config-
ure them in order to build products of a product line. The feature has been defined as
"a logical unit of behaviour specified by a set of functional and non-functional require-
ments" [46] and usually referred to a property of the system that is of some value to the
stakeholders. It is considered as a unit of reuse, specialisation and composition (usually
written in a programming language) in SPL engineering.

3.4.2 Existing Feature Modelling Notations & Tool Support

There are various notations used for feature modelling that basically extend the original
feature modelling notations from FODA. A feature model consists of one or more fea-
ture diagrams. The most basic of these notations include a feature diagram with a root
feature which represents a product family. The root feature can have a set of features
organised in a tree-structured hierarchy. A feature can be optional, mandatory or mutu-
ally exclusive to another feature. The graphical representation of optional feature is an
empty circle on top of the feature. An arc between two features is used for mutual ex-
clusion. The parent-child relationship is represented by lines. Various composition rules
are also supplied in textual form along with the feature diagrams. Figure 3.7 shows an
example of a basic feature model drawn using FODA notations.

One of the common notations for feature modelling currently used is ‘cardinality-based
feature modelling’ by Czarnecki et al. [70]. They extended the FODA notations by
introducing feature and group cardinalities, feature attributes and feature diagram ref-
rences. A feature model can comprise multiple feature diagrams. A feature diagram
has a root feature, which can have further features. The mandatory and optional feature
is graphically represented by filled and blank circle respectively. The cardinality of a
feature or a group is provided as an interval (e.g., m..n). A feature can refer to another
feature diagram which improves reusability by referring the singleton feature diagram at
various points in the feature model. The features in a group do not contain cardinality as
it is considered redundant and also helps during the specialisation of feature diagrams.

Figure 3.8 shows an example of the feature model which describes the product line for
electronic shops (EShop) as discussed in [29] and the graphical notations used are given
in Table 3.1. This feature model has four feature diagrams each with a root feature. The root feature ‘payment’ has two sub-features indicated by mandatory and optional symbols (i.e., having cardinalities [1..1] and [0..1] respectively). The feature ‘Payment Types’ has a group of three features, i.e., ‘CreditCard’, ‘DebitCard’ and ‘PurchaseOrder’. Its feature group symbol (OR Group) implicitly indicates the cardinality [1..n] where ‘n’ is the number of features in the group (size). Therefore, one of these payment types must be present in an instance of EShop. Other feature group symbol (XOR Group) indicates the cardinality [1..1], if it’s not explicitly given as in the case of ‘Chars’ sub-feature <2-4>. The feature ‘EShop’ refers to the features defined above in the same feature model where ‘Payment’ and ‘PasswordPolicy’ are mandatory and shipping can be optional.

Different configurations of features in a feature model will result in different instances of the system or members of the product line. This is done through a feature configuration diagram as shown in Figure 3.9 (left). Based on the selection of certain features, a specialisation of EShop is given in Figure 3.9 (right box).
Several tools have been developed to support feature modelling for product line software engineering [29, 100]. Some of these focus on the demonstration of their extended notations and others implement the existing notations in different perspective and for different domains. Some of these also reflect the improvements of the previously existing tools.

FeaturePlugin [29] was developed to support the ‘cardinality-based feature modelling’ as discussed above. This is an Eclipse plug-in providing EMF based tree structured editor for building feature models, their specialization and configuration. CaptainFeature [135] is another similar tool and the major difference to the above is in the rendering style for feature models. XFeature [114] is another Eclipse-based tool that requires the feature models to be expressed in XML using the XML editor. The editor is supported by an XML Schema representing the metamodel of feature modelling notations. Pure::Variants [77] is a commercial tool for SPL development which does not use cardinality based notations and provides constraint-oriented configuration through a Prolog-based constraint
solver. FeatureHouse [31] is a tool that allows the composition of artefacts and supports various languages with option to include more.

3.4.3 Related Work

- Feature Interactions

Feature interactions [58] is a well known problem in telecommunication systems and is equally applicable to all kinds of feature-based systems now. Feature interaction occurs when adding a feature modifies the behaviour of the system or affects the existing features leading to an undesired or probably unexpected behaviour of the overall system. For example [59], if a user has activated ‘call forwarding when busy’ feature on his telephone line and also subscribes to ‘call waiting’ feature, what would be the behaviour of the incoming call when the user is already on call? If the incoming call is forwarded,
then the call waiting feature is compromised and vice versa. The user will not obtain the expected behaviour in either case. Another example [142] of feature interactions is when a user wants to access voice mail of his mobile phone from his landline phone. The voice mail feature requires the user to press the ‘#’ key for listening to the received messages and the same key on his landline phone means ending the call. This sort of behaviour leads to undesirable feature interactions which must be detected and resolved for any feature-based systems.

The feature interactions problem must be considered for any product line based development framework such as our proposed feature-oriented modelling approach for Event-B, discussed in the next chapter. We might have to compose features which are not aware of the overall specification of the system or they do not have knowledge of other features. In this case, there may be chances of feature interaction problems as composing features might alter the behaviour of some of the features being composed. Apel et al. [30] proposed a feature-aware verification approach for detecting feature interactions in product lines where features composed using FeatureHouse [31] are checked for critical feature interactions automatically using their SPLVERIFIER tool suite. They use model checking to verify the existence of feature interactions without generating all the possible products of a product line.

The problem of feature interactions has not been addressed in this thesis. The main reason for not doing so is the time constraint and our primary focus being the introduction of feature-oriented reuse in Event-B. Once there is established feature-oriented reuse approach for Event-B, we could then apply existing feature interaction detection approaches from FOSD and product lines community (e.g., [30, 75]) and from formal methods (e.g., [56]) to our case-studies. Our suggested formal approach using Event-B could be quite useful in applying model checking for this purpose. Once feature interactions are detected, these can possibly be resolved by providing invariants when composing features and the composite model could then be verified by the Event-B tools.

3.5 Conclusion

This chapter completes our literature review. It provides an overview of the model-driven architecture (MDA) and the benefits of using MDA. We have discussed MDA based modelling frameworks (EMF, GMF) and a number of model transformation languages, which have been used in developing our feature modelling tool discussed later. We have also presented an overview of software product lines and how commonality and variability could be explored to introduce reusability. We discussed feature modelling, a well known SPL engineering technique, and its notions and available tool support within feature-oriented software development community. In the next chapter, we discuss our
feature-modelling notations for Event-B and the tool support developed as part of this research. This will then be used to model the case-studies presented in the later chapters and form the basis of our suggested modelling guidelines for feature-oriented reuse in Event-B.
Chapter 4

Feature Modelling in Event-B & Tool Support

4.1 Introduction

This chapter introduces our feature modelling framework for Event-B and the tool support that we have developed for experimental investigation of the feature-oriented reuse in Event-B through case-studies. We present our proposed feature modelling notations as an extension of existing notations commonly used in the feature-oriented software development (FOSD) community. These notations and the developed tooling is a starting point for us to explore how to formally specify software product lines (SPLs) in Event-B. This also enables us to experiment with standard Event-B composition and reuse approaches discussed earlier, possibilities and limitations of these, and help us define the need for tools and techniques relevant to product lines modelling. We then apply this feature modelling framework to real life case-studies in the following chapters to exploit further reuse opportunities, develop modelling patterns and suggest guidelines for future users when modelling feature-oriented systems as expected of SPL development using Event-B and Rodin.

4.2 A Feature Modelling Approach for Event-B

As a first step towards formally specifying product lines using Event-B, we decided to use feature modelling. Feature modelling is a widely known technique for managing commonality and variability within SPLs as discussed in Section 3.4.

A feature model represents a product family (i.e., all the products of an SPL) and different configurations of the feature model result in different members of the family. Therefore, we can split our development into two phases: a feature modelling phase,
where we build a feature model to specify a product line; and a *feature configuration* phase where we select desired features, resolve any conflicts and compose them to build a particular product. The output of the feature modelling phase is a feature model, which generically expresses the variability and the commonality among all the possible products of a product line. The output of configuration phase is a feature model instance, which is a result of selection and composition of features required for a particular product defined in the feature model and satisfying all the constraints expressed in the feature model of the product line. The composition of features is an important part of the configuration phase. We have given an overview of existing composition approaches of Event-B in Chapter 2 and due to their limitations for feature-based modelling through our case-study experiments, we suggest another form of composition, i.e., ‘feature composition’, later in this chapter. This composition – supported by tool – provides syntactic manipulation of Event-B elements when composing models, which also brings up further research questions.

Feature modelling in Event-B can be used in two ways. Firstly, when the Event-B specifications already exist and we want to use feature modelling to increase reusability by expressing the variability and commonality among different parts of the system so that the users can configure these specifications in different ways to model similar systems. Secondly, where we start development of a system by building a feature model of the system and then write Event-B specification afterwards. The later approach can also be used as a design activity during system development and allows users to design the model in a way to be more reuse-oriented.

### 4.2.1 Why to reinvent the wheel?

Since feature modelling is being used for product lines development for a long time now then why can we not use existing feature modelling notations and tools for Event-B modelling? And why do we have to reinvent the wheel by defining our own notation and tool for working with Event-B? The reason for doing so is because it was difficult to use existing feature modelling tools to specify Event-B features which are specified at higher abstraction level using set theory and predicate logic unlike typical high level programming language constructs used in the FOSD community. The stepwise refinement approach of Event-B is another aspect that is not usual of typical feature modelling approaches. Hence, we need a tool specific to Event-B, which would be able to adopt any modifications of the language, since it is continually being improved. Also, this would enable us to make use of all the available plug-ins of the Rodin platform such as editors, provers, model checkers and animators.

Our recent experiment [133] of adding Event-B support to an existing tool (FeatureHouse [31]) did not provide the same flexibility of feature composition as with our proposed feature modelling tool. FeatureHouse is an open-source tool for composing software
artefacts which allows addition of new languages by expressing these in BackusNaur Form (BNF). We found it difficult to integrate independently developed tools (such as FeatureHouse) within Rodin. One reason is an incompatibility of model interchange format between FeatureHouse (plain text) and Rodin (Event-B syntax aware XML). In order to use FeatureHouse, we also need the BNF representation of Event-B language which is currently provided as an EMF metamodel in Rodin. The language parser used in FeatureHouse requires Event-B models as plain text which may be done using a transformation language; but the continuous changing nature of Event-B would make it difficult to modify the transformation rules whenever Event-B language is extended to included further functionalities. Even if we consider this approach, we will need to write rules for round trip transformation. The tool will first convert Event-B models stored in Rodin into plain text to work with FeatureHouse and after manipulation, these must be then converted back to XML for storing into Rodin and to perform Event-B related tasks such as proving and animation. This might affect efficiency of the tool in terms of time consumption, depending on the size of the models and their refinements. The rules for composing models in FeatureHouse are written in Java. We would also require refactoring support while composing features and prefer to use a graphical feature composition tool rather than the script-based composition approach of FeatureHouse. Therefore, our proposed feature modelling approach will be supported by a state of the art formal method by providing extensible feature modelling, configuration and composition functionalities within Rodin tool.

### 4.2.2 Our Feature Modelling Notation

We adapted and extended the ‘cardinality-based feature modelling’ notation [70], so that we can build an Event-B specific feature modelling tool as a Rodin plug-in. We defined a feature to be an Event-B development which consists of a machine with various refinements and their seen context(s). Since an Event-B feature can have a chain of refinements, that makes it more difficult to deal with composing features having various refinement levels. We will discuss this later in our case-study examples.

The graphical notation used in our feature modelling framework is given in Table 4.1. A feature model consists of a tree structured feature diagram whose root feature takes the name of the model. The filled circle on a feature shows that it is a mandatory feature and optional otherwise (empty circle). The features with a triangle attached represent group features which are containers for other features and specify any constraints on the selection of features within that group. One such is the cardinality constraint that indicates how many of the features in the group must be present in a particular instance. An empty triangle means exclusive OR (XOR) or otherwise a group with cardinality (e.g., ‘2..4’) and the filled triangle means OR, i.e., the cardinality ‘1..k’, where k is the number of features in that group. There are three types of connections that can be used
Table 4.1: Our feature modelling graphical notation

<table>
<thead>
<tr>
<th>Notation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Mandatory</td>
</tr>
<tr>
<td></td>
<td>Optional</td>
</tr>
<tr>
<td>△</td>
<td>OR Group</td>
</tr>
<tr>
<td>△</td>
<td>XOR Group</td>
</tr>
<tr>
<td>△</td>
<td>Cardinality of Group</td>
</tr>
<tr>
<td>→</td>
<td>Includes</td>
</tr>
<tr>
<td>→</td>
<td>Excludes</td>
</tr>
<tr>
<td>![Feature]</td>
<td>Feature</td>
</tr>
<tr>
<td>![Feature Model]</td>
<td>Root Feature</td>
</tr>
</tbody>
</table>

to connect various model elements: child features, includes and excludes. The *includes* and *excludes* serve as constraints in a feature model. A feature can include other features, i.e., selecting that feature must also select the included features. Similarly, a feature can exclude other features and it is mutually exclusive, which means you can not have any two features with excludes connection between them in the configured instance. The leaf level features are actually mapped to Event-B developments during configuration.

Figure 4.1 shows an example feature model drawn using our feature modelling notation. This is an arbitrary feature model to show our graphical notation, which is inspired by our production cell (PC) case-study presented in the next chapter and it is only relevant for discussion in this section\(^1\). The root feature PC has a group of five features with cardinality “5..5” which means an instance must select all five of the group features. The features *BeltCrane* and *AdvCrane* of the *CraneType* group are mutually exclusive (i.e., cardinality ‘1..1’ shown by an empty triangle) which means no two of these can be present in a particular variant of PC derived from this feature model. By selecting *BeltCrane* feature, both the *DepositBelt* and *Crane* become mandatory due to group cardinality constraint (i.e., ‘2..2’).

Our main extension to existing notation is that of inclusion of the refinement concept of Event-B. We consider an Event-B development as a feature, where each refinement model can be composed during the configuration for product line instantiation. We also

\(^1\)We show arbitrary components of PC to demonstrate variability, which do not appear in the PC requirement specification as discussed in the next chapter.
provide includes and excludes constraints. Our feature modelling notation slightly differs from [132], as we allow feature refinements and a feature can exclude other features in a feature model.

4.2.3 Metamodelling

We developed a metamodel based on our feature modelling notation as shown in Figure 4.2. This has been developed with a view to possible future extensions to our feature modelling tool that have not been included in the current version, such as annotating the feature models with composition rules and other constraints. The FeatureModel, a named element, is the root feature that can contain both the features and groups as it
inherits from their abstract containers, i.e., FeatureContainer and GroupContainer. The Group and Feature are also named elements and inherit from the Constraint class which means we can specify constraints at both the feature level and at the group level. The Constraint class has min and max attributes for specifying cardinality constraints and expression attribute for additional constraint or description. A group can contain features and vice-versa since both the Group and Feature classes inherit from their respective containers. A feature can include/exclude other features. A feature is not restricted to include/exclude itself in the meta-model as it required the use of a constraint language (e.g., OCL). This was dealt with through the code during implementation.

4.2.4 Feature Model Validation

Model validation is an important issue that must be addressed in any model-driven development. Hence, we need to provide a way to make sure that our models are valid. By valid we mean that they conform to certain defined criteria or rules. In our feature modelling framework, the feature models must conform to the metamodel (which defines our feature modelling notation) and any constraints given in the metamodel must be satisfied. Also, there are other constraints that can not be expressed in a simple metamodel and require the use of a constraints language (this could be considered in the future if more complex constraints need to be expressed based on the case-study experiments). Following are the properties/constraints that must be satisfied in order to build consistent and correct feature models. To be more precise, a correct feature model conforms to its metamodel, does not violate any constraints and can be instantiated. These constraints have been derived from literature and through our experience with existing notations and tools from feature-oriented modelling community.

- A feature model must not have cycles which means a feature must not have any of its ancestors as its child feature, e.g., a feature $x$ having a child feature $y$, which is a parent feature of $x$.
- A feature must not include and exclude the same feature.
- A feature $x$ must not include features $y$ and $z$, which exclude each other. If this scenario is present in the feature model, it is not possible to generate a valid instance of the feature model.
- A feature must not exclude any of its ancestor features.
- A feature must not be unreachable during the configuration (i.e., not be selectable in any valid configuration). An example would be a group of two features with group cardinality ‘1..1’ and one of the features is mandatory. In this case, the user will never be able to have the optional feature selected in a valid configuration as doing that will violate the group cardinality due to the other mandatory feature.
A feature model may not have orphan features. All the features must be connected to the feature tree in order for these features to be configured as part of the feature model.

4.3 Current Feature Modelling Tool Support for Event-B

Based on our feature modelling approach discussed above, we have developed a feature modelling tool (FMT) to specify feature models for product lines and to configure and compose these feature models to generate product line instances. The tool is open source, developed in Eclipse using Java and integrated as a plug-in to the Rodin platform. It consists of two parts, i.e., feature model editor and configuration editor, discussed below. To put it simply, a feature model represents a product line and a feature instance model represents a product line member. A model is an instance of the metamodel and corresponds to an object (model) of a class (metamodel) in object-oriented programming.

Figure 4.3 shows architecture of the tool, which is divided into three phases. In the first phase, we built the feature metamodel in EMF (Ecore) [3] for our feature-based modelling notation, as discussed above. This metamodel was used to build a graphical feature modelling editor for drawing feature models of product lines. The feature models drawn using this editor are then transformed into their equivalent metamodels in the second phase, required for further instantiation (discussed in Section 4.3.2 below). This transformation process is internal and invisible to the user. In the third phase, the feature configurator allows the user to instantiate feature models drawn in the first phase. This is done by selecting the desired features which are then composed by the tool to generate the feature model instance. The feature models, their instances and configuration decisions are serialised in the Event-B features database (Event-B Project).

Each of these modules is discussed in detail below.

![Feature modelling tool architecture](image)

Figure 4.3: Feature modelling tool architecture
4.3.1 Feature Model Editor

Our feature modelling tool includes a graphical feature model editor (FME)\(^2\) developed using GMF [7]. This editor allows feature models to be built in a free form tree structured hierarchy and uses common graphical notations of feature modelling, which are mostly used when drawing feature models by hand. The feature models are serialised as EMF models. At the feature modelling phase, a feature is an atomic entity as it represents the unit of requirements. The Event-B specification of a feature is not shown on the feature model but the user can view it (if it exists already). Event-B development of a feature is shown in the configuration phase because that is where we have to compose various features and resolve any issues that may result in an invalid configuration of the feature model. This approach helps the modeller in focusing on the commonalities and variabilities within a product line during the feature modelling phase and not on the implementation of the features. This also makes our approach generic by allowing any feature models to be formalised as an input to be used in this framework.

4.3.1.1 Implementing Feature Model Validation

There is a three-way validation mechanism provided by the editor to support validation criteria explained above. The division is based on how this is implemented in the tool using Java. Firstly, feature models should conform to the feature metamodel. Secondly, it does not let the user draw a feature model that violates any of the following validation cases which are checked by the tool in real-time.

- A feature model must not have cycles, which means a feature must not have any of its ancestors as its child feature, e.g., a feature \(x\) having a child feature \(y\), which is a parent feature of \(x\).

- A feature must not include and exclude the same feature.

Thirdly, the editor validates a feature model when it is saved based on the following validation cases and warns the user if any of the cases is violated by the model. This differs from the above as it lets the user save an inconsistent/incomplete model to be completed later. If the user then tries to configure this incomplete feature model, the resulting model might not be a valid instance of the feature model.

- A feature \(x\) must not include features \(y\) and \(z\), which exclude each other. If this scenario is present in the feature model, it is not possible to generate a valid instance of the feature model.

\(^2\)Contributed by Nikola Milikic (a University of Southampton Intern)
• A feature must not exclude any of its ancestor features.

• A feature must not be unreachable during the configuration (i.e., not be selectable in any valid configuration). An example would be a group of two features with group cardinality ‘1..1’ and one of the features is mandatory. In this case, the user will never be able to have the optional feature selected in a valid configuration as doing that will violate the group cardinality due to the other mandatory feature.

• A feature model must not have orphan features.

4.3.2 Model To Metamodel Transformation

A metamodel provides a language or notation for building the models. The feature models built using the feature model editor are transformed into their equivalent EMF metamodels at run-time and for each product family. This model to metamodel transformation is needed in order to instantiate the feature models and this is a way to force the instances to conform to their metamodel. Hence, a feature model is an instance of the metamodel defining our feature modelling notation and it also serves as a metamodel for any of its instances.

The transformation is done using Epsilon Transformation Language (ETL) [96] which is part of the Epsilon framework. It provides an execution engine for the transformation rules. By using a transformation language, we can achieve the benefits of Model-Driven Development. It is also easier to use rule-based language compared to a programming language here. We might need to use the Epsilon validation framework later on to validate that the generated instances are valid instances of the feature models. We have looked at other transformation languages such as ATL [92] and RDL [9]. However, ETL is more suitable for Eclipse development because of various plug-ins provided within the Epsilon framework and continuous improvements and extensions. It is also better in terms of documentation, which is provided as an e-book [113]. It gives a complete reference of the syntax and semantics of the languages in Epsilon. There is a repository of examples provided to show the application of Epsilon in various ways and an online support forum which proved very helpful while using ETL in our tool development.

Figure 4.4 shows the metamodel of PC example feature model of Figure 4.1, which was automatically generated based on our transformation rules written in ETL. This metamodel itself is an instance of the metamodel of Figure 4.2, which can be used to generate instances of the PC feature model and any instance of PC should then conform to this metamodel. After transforming feature models into metamodels, for different product lines, these are then used as an input to the feature configurator for instantiation discussed below.
4.3.3 Feature Configurator

The feature configurator (FCON) allows the modeller to select a configuration of features that they want to include in a particular product. The tool then resolves any conflicts automatically/interactively and composes these selected Event-B models into a single composite Event-B model. The FCON is a collapsible tree-structured editor (see Figure 4.5). The FCON enforces some of the constraints provided in the feature model at runtime as discussed earlier. It automatically selects the mandatory features and highlights any violation of the constraints, e.g., cardinality constraint (as highlighted in the figure). Whenever a feature is selected, the tool automatically selects/deselects features specified using includes/excludes constraints. The configurator also shows the associated Event-B machines and contexts for the features.

4.3.3.1 Conflict Resolution

The tool highlights any conflicts that need to be resolved. At the moment it detects naming conflicts within Event-B models (e.g., variables or events having same name). It
provides ways to automatically resolve these name clashes either by making them disjoint through refactoring or by simply deselecting repeating entries in multiple features. It also helps the user in automatically selecting any dependencies, for example, if an event is selected, it can then select the related variables and their invariants to build the correct model.
4.3.3.2 Feature Composition

Once all the desired features are selected and conflicts are resolved, they are composed to generate a composite Event-B feature. Any of the following composition styles can be used to compose the selected features. Note that this can be used as a stand-alone composition tool\textsuperscript{3} developed using GMF [7] for Event-B, though it can also be provided as a separate plug-in [6]. It allows composing Event-B developments even if the user is not interested in using feature modelling.

- **Simple Composition:**
  This composition simply merges all the elements in their respective sections within the feature (without renaming). This adds all the variables from the selected machines into the variable section of the composed machine and similarly adds all the events to the operations section of the composed Event-B machine. The \textit{Initialisation} events from all the machines will be merged into one event in the composed machine because an Event-B machine can only have one such event. A composed feature over here might have repeating elements and these can then be pointed out as errors by the static checker of Rodin tool. Figure 4.6 shows two machines $M_1$ and $M_2$ which are composed using this composition style resulting in machine $M$ shown in Figure 4.7. This composition is useful when composing disjoint machines that do not have any conflicting elements. It may not be very useful otherwise but still this can give the user a better understanding as they will start fixing the errors manually by removing/modifying the problem elements.

- **Disjoint Composition:**
  All the user-selected features are made disjoint through renaming before composition. After making the features disjoint, these are simply concatenated/merged into a single large feature using the simple composition style given above. The advantage of this sort of composition is that the user does not have to do anything and this is an automatic composition without almost any syntactic errors. The disadvantage of this style is that there may be redundant elements with different names and actually doing the same thing, which will have to be manually fixed. The implementation of this style was carried out by prefixing the elements (constants, variables, events) with the name of the machine/context. For example, machines $M_1$ and $M_2$ in Figure 4.6 are made disjoint and composed resulting in machine $M$ in Figure 4.8. The variable $x$ of machine $M_1$ becomes $M_1.x$ and an event \textit{decrement} of machine $M_2$ becomes $M_2.decrement$. Renaming an element such as a variable is replicated at all the occurrences of the variable in the given machine, i.e., invariants and events. The \textit{Initialisation} events of all the composing machines are merged into a single \textit{Initialisation} event. By making the variables disjoint, their initialisation also becomes disjoint automatically.

\footnote{Partly contributed by Chris Franklin (a University of Southampton Intern)}
This style of composition can be very useful when modelling a product of a product line where we need multiple instances of the same feature. For example, we have a model for a push button and we have a requirement for another push button having the same functionality and that might eventually be implemented on a different hardware. Therefore, we can simply duplicate the functionality of the button by making them disjoint and composing along with other selected features of the product. We will show how this style is applied during our case-study experiments when modelling variability in the next chapter.

- **Conflict-aware auto merge:**

  This is an extended form of simple composition. For all the conflicting variables with coinciding names and types, the tool uses only one instance of such variables. The repeating variables with same name and different types are resolved by the tool through renaming such variables accordingly. Similarly, for repeating events with the same name and different actions should be renamed. Only one instance of repeating events with same functionality should be used. The limitation of this style is that it would not be able to differentiate between the repeating elements with the same name and type but used differently in different machines. An advantage of this composition style is that it is more efficient, fast and requires less user interaction.
Conflicts-aware merge with user input:

This is also simple composition but all the conflicting elements are analysed and highlighted for the user to decide which of these to keep and which ones to modify/delete. For example, if an element such as a variable with the same name and different type exists in the selected machines, the user is then asked to resolve this by choosing the option to rename, remove or to change the type of the variable. It would be useful to provide as much information as possible to resolve conflicts or by suggesting any actions that user might need. This style basically provides the user with all the issues that must be addressed before composition and the user manually resolves these to make sure that the composed model is a right composition.

Event Composition:

This composition style allows merging events from different features leaving out unwanted events of those features. When the user selects events from multiple features for composition, these events are analysed for any conflicts and composed using any of the above mentioned composition styles. Typically, this works by concatenating the actions and conjoining the guards of the events being merged to maintain invariant preservation. This composition style helps in increasing the reusability by allowing parts of large machines to be reused.
By using these composition styles, all the machines are merged into a single machine and all the contexts are merged into a single context. This can be called as a structured cut-and-paste composition approach and is quite useful when composing several models, which might be cumbersome to do manually.

4.3.3.3 Limitations of Feature Composition

This composition of Event-B features into a composite feature is required in order for us to reason about the complete model of the generated instance, i.e., through animation or theorem proving etc. One of the limitations of the tool is that we have to reprove the composite model even though the individual features are already proved. There may be some POs already proved for features being composed which are carried forward for the composite model but we do not have a mechanism to detect and reuse these POs. By devising a proof reuse mechanism, this would enable us to automatically discharge most of the POs and newly generated POs could also be discharged using existing ones. This is left as a future work; but our case-study experiments discussed in the following chapters highlight some modelling patterns using existing Event-B (de)composition techniques, which could enable proof reuse or help us avoid reproof by modelling a system in a particular way.

Another limitation of our proposed feature composition is when composing non-conformal refinement chains (i.e., containing different number of refinement steps), which is not straightforward. At the moment, we have to compose models at each refinement level of the features being composed. Other issues would arise when composing two refinement chains of Event-B features having different number refinement steps, discussed in future work Section 8.3. For example, consider a feature $A$ having three refinements and another feature $B$ having two refinements as shown in Figure 4.9. In this case, it might be difficult to automate the composition of these two features since the tool might not be able to judge which levels of refinement from the two features should be composed and how many refinement levels will the composite feature contain? This may be resolved in different ways. For example, composing $B_0$ twice with $A_0$ and $A_1$ in the figure. Another possibility is to introduce an intermediate refinement step between $B_0$ and $B_1$, which would simply refine $B_0$ where all its events would refine $\text{skip}$. Again, our proposed modelling pattern discussed in Chapter 6 addresses this issue.

Now that we have a tool which can be used to model example case-studies, it would help us to figure out common patterns for composition and will be extended in future to include such patterns to further automate the composition process.
4.4 Conclusion

We have presented our feature modelling framework for Event-B, which enables reusability of Event-B specification as expected of feature-based product line modelling. We adapted and extended existing feature modelling notation and developed a feature modelling tool as a Rodin plug-in. We decided to build our own tool rather than using any of the existing feature modelling tools to have full support of Event-B language and its related tools such as animators, model checkers and provers. This tool allows us to explore variability and commonality among different features of a product line and configure the features to model particular products of the product family. The feature models drawn using the tool are checked for validation to ensure that they conform to our feature modelling notation and that any instances generated through these are valid instances of the feature models. The configuration process includes selecting a set of required features, highlighting and resolving any conflicts with the help of the user while maintaining the constraints specified in the feature models. Once the user is confident that all the conflicts are resolved, the selected features are then composed resulting in the desired product of the product family.

At the moment, the tool has some limitations which can be improved based on the case-study experiments. One of these is that the composed features need to be reproved as we have not yet explored the proof reuse mechanism for our proposed feature composition. Though the case-studies presented in the next chapters highlight some modelling patterns which could enable proof reuse. Another limitation is the non-conformality of refinement chains, i.e., composing refinement chains of Event-B features having different number of refinement steps. In the next chapter, we model a well known case-study in Event-B and explore how our feature modelling framework can be applied to achieve benefits of reusing Event-B models, which would form the basis for formal feature-oriented product line reuse in Event-B.
Chapter 5

Case Study - Production Cell

5.1 Introduction

In this chapter, we present production cell (PC) case-study, a real life control system example modelled in Event-B, to be used as an example for our formal featured-oriented reuse framework. The PC has various physical components which must be controlled in real time while maintaining safety properties to avoid any loss to human beings and damage to the equipment. We discuss how we can decompose and compose Event-B models in different ways and at different abstraction levels to explore product line reuse approach in Event-B. By modelling the same example in different ways and using existing decomposition approaches of Event-B, we can provide statistics in term of reusability, modelling and proof effort. We can generalise the different approaches used in this example to suggest patterns of modelling and guidelines for future users. This case-study has been useful in exploring reusability, defining the notion of features for feature-based modelling in Event-B, evaluating existing Event-B (de)composition techniques and to figure out the tooling requirements for product line modelling using Rodin.

5.2 Roadmap

We discuss Event-B development of the PC [4] in three different ways. We start with the physical component-based modelling of PC in Section 5.4, where we model and refine PC system as integral model and then decompose it into various physical components of the plant. We use both types of Event-B decomposition techniques, i.e., shared-variable decomposition (SVD) and shared-event decomposition (SED); and discuss the workload of modelling the system to be decomposed later using the two styles. We then discuss controller-based (functional) modelling of PC in Section 5.5, where we generalise PC requirements to model generic controllers of the plant or functional features that
could then be specialised for modelling various components of the PC. This approach shows more reuse opportunity compared to the components-based PC. We then discuss domain-specific modelling approach based on static variability in Section 5.6, where we can model variants of PC by switching the contexts. This allowed us to use different methods of modelling the same system in Event-B and analysing our approach to feature-based modelling using existing tools and techniques in Event-B. This also enabled us to explore the amount of reuse that can be achieved using different modelling styles.

Refinement is a process of introducing more details in each step from abstract specification to the concrete one, which is closer to implementation. By using refinement, we can model a system in multiple steps and deal with small number of requirements in each step rather than modelling the entire system in a single model. This becomes very difficult to manage and prove when modelling everything in a single refinement step. Hence, step-wise refinement is the approach we have used to model the production cell system.

5.3 Production Cell

The production cell (PC) [104] is an industrial metal processing plant which falls under the category of critical systems. This is an example of a reactive system that has been modelled in more than 30 formalisms [112, 115, 73, 105, 138], including the B formal method, which is a predecessor of the Event-B language [126]. The production cell has also been specified at an abstract level using the RAISE [111] formal method and a stepwise refinement approach has been used to generate the implementation. This was supported by proofs to verify the safety properties of the system [102].

The production cell processes metal blanks that are routed to a press for forging, then routed away from it after processing. Figure 5.1 shows top view of the production cell plant. Metal blanks enter into the system through the feed belt and are dropped on the elevating-rotary table if the table is empty and in the loading position. The table has two positions. The first position is for receiving blanks from the feed belt when the table is neither elevated nor rotated. The table, once loaded, elevates and rotates to the second position so that the first robot arm can pick up blanks as the robot arms cannot move vertically. The robot rotates around its own axis. It has two arms that can extend and retract horizontally and independently of each other. The first arm picks a blank from the table and drops it on the press. The press has three positions as the robot arms are at different planes to avoid collision. The first robot arm drops the blank when the press is in the middle position. The press forges the blank at the high position and moves to the lower position from where the blank is picked up by the second robot arm. The blank is then dropped onto the deposit belt. There is a flaw in the description of the production cell presented in [104], which does not define the
mechanism for finding out the blanks that are not properly forged and fed back into the system for reprocessing. In reality, forged blanks should leave the system after being transported on to the deposit belt. So, we assume that the forged blanks on the deposit belt are removed from the system by some external mechanism, which is not included in our modelling, and the ones that are not forged properly travel towards the end of the deposit belt. A moving crane then picks these unforged blanks from the deposit belt and brings them back to the feed belt for reprocessing. This completes one cycle of the production cell. There are 13 actuators and 14 sensors for controlling and monitoring various components of the plant. The detailed description of the production cell and various requirements are narrated by [104]. The requirements specification subset we used for modelling the production cell is given in Appendix A.

5.4 PC Component-based

5.4.1 Development Structure

In the physical component-based modelling approach, we started with an abstract model of the production cell and refined it in a number of steps. The first four levels are horizontal refinements where we introduced further requirements in each refinement step and the later ones are the vertical refinements after we have decomposed the integral model into various physical components using SVD.
We have also modelled the PC in a way that we could decompose it using SED. This required us to prepare the model so that there are no shared variables and different components communicate via shared-events. This means that we had to perform vertical refinements earlier on, unlike the development that used SVD where we only do horizontal refinement before the decomposition.

Each of these components can then be refined separately while maintaining the constraints of the decomposition technique. We only refined one component (i.e., Press, resulting from SVD of PC) further as the other components can be refined in the similar manner. During the vertical refinements, actuators and sensors were introduced to model the production cell closer to implementation. Figure 5.2 shows Event-B refinement architecture for component-based PC modelling and the detail of each refinement step is discussed below.

### 5.4.2 PC Abstract Model (PC\(_0\))

The basic model of the production cell without much detail is specified in the abstract model. Here we specify the whole production cell cycle described above in one event *Operate*, which models the processing of blanks from *forged* to *unforged* state through the variable *blanks*. The Event-B model is shown in Figure 5.3 where *PC\(_CO\)* is the abstract context (Figure 5.3(a)) seen by the machine (Figure 5.3(b)). The machine has a variable *blanks* and an invariant describing the variable type, i.e., every blank must
Chapter 5 Case Study - Production Cell

5.4.3 PC First Refinement (PC_1)

In the first refinement, we introduce system components such as feed belt, robot arms, press, deposit belt, crane and model position of each blank. So at any time, we know where a particular blank is positioned and its status, i.e., forged or unforged. A part of the Event-B model after first refinement is given in Figure 5.4. The machine on the right refines PC_0 and sees the context PC_CO1 shown on the left that extends the context PC_CO of Figure 5.3(a). The context PC_CO1 defines an enumerated set POS containing the positions (axm1), which blanks can have during the entire cycle of the production cell system. The machine contains a variable for the position, which is defined as a function from BLANKS to POS (inv1), so a blank can be positioned at any one component of the system at any time. The invariant inv2 specifies that the blanks on
the feed belt, table, arm1 and crane are unforged and those on arm2 are forged as stated in the invariant inv3. The deposit belt receives forged blanks from arm2. The properly forged blanks are removed from the system. The status of improperly forged blanks is changed to unforged through some external mechanism so they can be processed again, as stated in the problem description earlier. Several new events were introduced for adding blanks to the feed belt (loadFeedBelt), loading the table (loadTable), dropping blanks on the press (loadPress), loading the robot arms (loadArm1, loadArm2), dropping blanks on the deposit belt (loadDepositBelt), removing forged blanks from the system (removeBlanks) and changing status of improperly forged blanks present on the deposit belt to unforged (unforgeBlanks). The abstract event Operate is refined by opPress event, which models the processing of the blanks. The loadFeedBelt event shown in Figure 5.5, is introduced for adding blanks into the system for processing. The guard grd3 defines that a new blank can be added if it is not already in the plant or it is already in the plant but comes from the crane.

Figure 5.6 shows how the atomicity refinement of Operate event is performed in this level of refinement. The dotted lines show new events introduced in the refinement, whereas the solid line shows refined event. The order in which events can take place is read from left to right, as provided in the notation for atomicity decomposition diagram [53].

This refinement models requirements 2, 6, 13, 21, 28 and 35 of the PC requirement specification.
**Event**  loadFeedBelt ≡

\[
\begin{align*}
\text{any} & \quad \hat{=} \\
\text{where} & \\
grd1 & : b \in \text{BLANKS} \\
grd2 & : \text{blanks}(b) \neq \text{forged} \\
grd3 & : b \notin \text{dom}(\text{position}) \lor \text{position}(b) = \text{cr} \\
\text{then} & \\
act1 & : \text{position} := \text{position} \triangleleft \{b \mapsto \text{fb}\} \\
\text{end}
\end{align*}
\]

Figure 5.5: PC_1 loadFeedBelt event

**Figure 5.6: Atomicity refinement of PC_0 event in PC_1**

### 5.4.4 PC Second Refinement (PC_2)

In the second refinement, we further refined the operations taking place at different components of the production cell and introduced positions/states of the components table, robot and press. The context at this level of refinement defines the enumerated sets: PRESSPOS for press positions (i.e., low, mid, high) and RBTPOS for three robot positions (i.e., pos1, pos2, pos3). Below are some invariants defined at this refinement level:

\[
\begin{align*}
\text{inv5} & : t\text{blElevated} = \text{FALSE} \Rightarrow t\text{blRotated} = \text{FALSE} \\
\text{inv6} & : t\text{blRotated} = \text{TRUE} \Rightarrow t\text{blElevated} = \text{TRUE} \\
\text{inv7} & : \text{position} \triangleright \{\text{fb, db}\} \in \text{BLANKS} \implies \text{POS}
\end{align*}
\]

The invariants (inv5 and inv6) control the correct positions of the table at any time, which means only two of the possible four table positions are valid, as described earlier. The invariant ‘Inv7’ makes sure that the components other than feed belt and deposit
belt should not have more than one blank on them. The operations defined in the previous step are refined further to include more details, e.g., the event \textit{loadTable} of PC\_1 is refined by the events \textit{loadTable}, \textit{moveTableUp}, \textit{moveTableDown}, \textit{rotateTableFwd} and \textit{rotateTableBwd} to complete the functionality of the table as shown in Figure 5.7. Below is the representation of the \textit{loadFeedBelt} event obtained at this refinement level. The guard \textit{grd5} ensures that a blank is only added to the feed belt if the feed belt has not already reached its capacity.

\begin{figure}[h]
\centering
\includegraphics[width=0.8\textwidth]{figure5.7}
\caption{Atomicity refinement of \textit{loadTable} event in PC\_2}
\end{figure}

\begin{tabular}{|c|}
\hline
\textbf{Event} \quad \textit{loadFeedBelt} \equiv \\
\textbf{refines} \quad \textit{loadFeedBelt} \\
\textbf{any} \\
\textbf{b} \\
\textbf{where} \\
grd1 : \quad b \in \text{BLANKS} \\
grd2 : \quad \text{blanks}(b) \neq \text{forged} \\
grd3 : \quad b \notin \text{dom}(\text{position}) \lor \text{position}(b) = \text{cr} \\
grd4 : \quad \text{finite}(\text{dom}(\text{position} \rhd \{fb\})) \\
grd5 : \quad \text{card}(\text{dom}(\text{position} \rhd \{fb\})) < fbMax \\
\textbf{then} \
\textbf{act1} : \quad \text{position} := \text{position} \rhd \{b \mapsto fb\} \\
\textbf{end} \\
\hline
\end{tabular}

This refinement models requirements 5, 7, 8, 9, 10, 11, 16, 22, 23, 24, 25, 26, 27, 29 and 33 of PC requirement specification.

\subsection{5.4.5 PC Third Refinement (PC\_3)}

In third refinement, we introduced control functionality of the robot arms and movement of the belts for the blanks to be delivered to the next stage. We introduced sets \texttt{ARMSTATUS} for recording the status of robot arms (i.e., extended, retracted) and
BELTSTATUS to record whether a belt is running or stopped. Figure 5.8 shows some invariants of the machine at this refinement level. We also introduced some safety requirements for the components here. For example, safety constraints imposed on the movement of robot arms to avoid collision are as follows:

- Arm1 must be retracted when the robot is in position 2 (inv5).
- Similarly, Arm2 must be retracted at all when the robot is in position 1 (inv6).

The invariant ‘inv9’ specifies that the feed belt should only move to make a blank available at its end or if it already has a blank and the table is in a position to receive the blank. The invariant ‘inv10’ ensures that the deposit belt should only move to make a blank available at its end to be picked up by the crane.

The moveFeedBelt event, shown below, models the movement of the belt if it has a blank on it (grd2). The belt can move if there is no blank at the end of the belt to be dropped on to the table or if the table is vacant and in a position to receive a blank (grd2). When the feed belt moves, it makes a blank available on its end to be delivered to the next component.

The loadCrane event below allows the crane to pick a blank from the deposit belt and unloads the belt.
5.4.6 PC Fourth Refinement (PC₄)

This is a small refinement step where we introduced further requirements for the travelling crane, which includes controlling its movement between the two belts and loading/unloading functionality for blanks. A variable cranePos is introduced for the position of the crane. It can be either at the deposit belt or at the feed belt. The new events added in this refinement were moveCraneToFb and moveCraneToDb as shown below:

$$\textbf{Event} \quad \text{moveCraneToFb} \cong$$
when
$$\text{grd1} : \text{cranePos} = \text{crposdb}$$  //crane is at deposit belt
$$\text{grd2} : \text{cr} \in \text{ran(position)}$$  //crane has a blank in it
then
$$\text{act1} : \text{cranePos} := \text{crposfb}$$
end

$$\textbf{Event} \quad \text{moveCraneToDb} \cong$$
when
$$\text{grd1} : \text{cranePos} = \text{crposfb}$$  //crane is at feed belt
$$\text{grd2} : \text{cr} \notin \text{ran(position)}$$  //crane is vacant
then
$$\text{act1} : \text{cranePos} := \text{crposdb}$$
end

The event loadFeedBelt of PC₃ was refined by two events (as shown in Figure 5.9) - loadFeedBelt and unloadCrane. The later models loading of new blanks or improperly forged blanks for reprocessing. The guard grd4 of loadFeedBelt event at the top is split into guard grd4 in both the events at the bottom.

This refinement models requirements 36, 38, 39, 40 and 41 of the PC requirement specification.
There are various other requirements that we have not modelled due to time limitation and can be introduced in further refinement steps in future, as discussed by [104]. For example, the controller must ensure that no blank takes longer than a specified amount of time in the plant and ideally, each blank should take minimum possible time. Also, the controller should make sure that there are always enough blanks in the system to gain maximum throughput. [104]

5.4.7 Decomposition

After four horizontal refinements, at which most of the PC requirements were modelled, we introduced actuators and sensors for different components of the PC and performed vertical refinement. The functionality of sensors and actuators is independent for each of the components. Also, the model became quite big, which was difficult to understand and refine further as a whole. So, we decomposed the model into various physical components (sub-models) of the PC (i.e., feed belt, table, robot, press, deposit belt and crane). Below we present the detail for applying both styles of Event-B decomposition to the component-based production cell development.
Chapter 5 Case Study - Production Cell

5.4.7.1 Shared-Variable Decomposition (SVD)

At first, we used shared-variable decomposition (SVD) since different components were sharing variables. For example, all components shared the variable blanks, which models the status of blanks at any component. The variables tblRotated and tblElevated are shared among feed belt, table and robot components and appear as event parameters in the other components rather than as shared-variables. The same is the case for the variable pressPos, which is shared between the press and robot components. During the decomposition, the events related to a particular physical component became events of that sub-model and any events of the sub-model involving the shared-variable became external events in all other components. For instance, the event loadTable, moved to the table sub-model, became an external event in all the sub-models for other physical components of PC. Figure 5.10 shows loadTable as an internal and external event in the table and press components respectively. Table 5.1 shows the distribution of internal/external events among various components of PC after SVD.

Figure 5.10: loadTable as internal and external event in Table and Press components respectively after SVD

Table 5.1: Event distribution among components of PC based on SVD

<table>
<thead>
<tr>
<th>Event Type</th>
<th>Feed Belt</th>
<th>Table</th>
<th>Robot</th>
<th>Press</th>
<th>Deposit Belt</th>
<th>Crane</th>
</tr>
</thead>
<tbody>
<tr>
<td>Internal Events</td>
<td>3</td>
<td>5</td>
<td>9</td>
<td>5</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>External Events</td>
<td>13</td>
<td>11</td>
<td>14</td>
<td>10</td>
<td>10</td>
<td>7</td>
</tr>
</tbody>
</table>
Table 5.2: Number of local and shared events of PC components based on SED

<table>
<thead>
<tr>
<th>Event Type</th>
<th>Feed Belt</th>
<th>Table</th>
<th>Robot</th>
<th>Press</th>
<th>Deposit Belt</th>
<th>Crane</th>
</tr>
</thead>
<tbody>
<tr>
<td>Local Events</td>
<td>0</td>
<td>4</td>
<td>7</td>
<td>4</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>Shared Events</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

5.4.7.2 Shared-Event Decomposition (SED)

In order to explore whether we can use shared-event decomposition (SED) to decompose the integral model into sub-models, we had to prepare the model to be decomposed using the SED style, right from the abstract model. For this, we had to partition (localise) any variables that are shared between different components, so that there is no more shared variables. For example, the shared variable \( \text{position} \) in \( PC_4 \) (i.e., \( \text{position} \in \text{BLANKS} \rightarrow \text{POS} \)), which maintains the position of a blank on a component within the plant, would be partitioned for each component. So, each component knows the blank(s) present on it, e.g., variable \( \text{blanksOnFb} \) (i.e., \( \text{blanksOnFb} \in \mathbb{P}(\text{BLANKS}) \)) models the blanks present on the feed belt and so on for the rest of the plant components.

During the decomposition, we partitioned the variables into various sub-models along with their related events. Figure 5.11 shows how an event \( \text{loadTable} \) is split into two events for the feed belt and table components. We simply split the guards and actions into two. If a guard or action of an event is complex and cannot be split then it must be simplified during the preparatory step to be split into two. For example, Figure 5.12 shows the event \( \text{moveFeedBelt} \) prepared in a refinement step to be split, as the guard \( \text{grd3} \) of the event on the left could not be partitioned. So, we had to introduce extra parameters (\( t_1, t_2, t_3 \)) and guards (\( \text{grd3, grd4, grd5} \)) to simplify this guard. Hence, the guard \( \text{grd3} \) of event on the left is simplified by guard \( \text{grd6} \) on the right. The decomposition of this event is shown in Figure 5.13 for feed belt and table components. Note that we had to do vertical refinement in order for us to perform SED unlike SVD where we only carried out horizontal refinements before the decomposition. So, it depends on the type of system being modelled and for distributed systems, the SED approach seems more appropriate.

Table 5.2 shows the local and shared events of various components of PC after SED. Table 5.3 shows the distribution of internal/external events among various components of PC if we had performed SVD instead of SED. In this case, there are no globally shared variables and the variables are only shared between connecting components, e.g., the table component shares variables with the feed belt and robot as it interacts with the two in order to receive and deliver blanks.
Table 5.3: Events distribution of PC components when performed SVD on a model prepared for SED

<table>
<thead>
<tr>
<th>Event Type</th>
<th>Feed Belt</th>
<th>Table</th>
<th>Robot</th>
<th>Press</th>
<th>Deposit Belt</th>
<th>Crane</th>
</tr>
</thead>
<tbody>
<tr>
<td>Internal Events</td>
<td>3</td>
<td>5</td>
<td>9</td>
<td>5</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>External Events</td>
<td>9</td>
<td>4</td>
<td>11</td>
<td>5</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

Event `loadTable \equiv`

//Event before decomposition

any

b

where

grd1: blankOnTbl = ∅
grd2: b ∈ blanksOnFb
grd3: tblElevated = FALSE
grd4: tblRotated = FALSE
grd5: blankOnFbEnd = TRUE
grd6: fbStatus = running

then

act1: blankOnTbl := blankOnTbl \cup \{b\}
act2: blanksOnFb := blanksOnFb \{b\}
act3: blankOnFbEnd := FALSE

end

Event `loadTable \equiv`

//Event of Feed Belt Component

any

b

where

typing_b: b ∈ BLANKS
grd2: b ∈ blanksOnFb
grd5: blankOnFbEnd = TRUE
grd6: fbStatus = running

then

act2: blanksOnFb := blanksOnFb \{b\}
act3: blankOnFbEnd := FALSE

end

Event `loadTable \equiv`

//Event of Table Component

any

b

where

typing_b: b ∈ BLANKS
grd1: blankOnTbl = ∅
grd3: tblElevated = FALSE
grd4: tblRotated = FALSE

then

act1: blankOnTbl := blankOnTbl \cup \{b\}

end

Figure 5.11: Event splitting example for SED
Modelling a system using fewer shared variables seems to be a better option here. This is because after SVD we get lots of external events, which must not be refined. This increases the size of the model making it less readable and difficult to manage/understand after some refinements. For example, as shown in Table 5.1, the feed belt component of PC has 3 internal events and 13 external events. Table 5.1 and Table 5.3 show the number of internal/external events of PC components decomposed using SVD after modelling these with and without globally shared variables among the components respectively. This suggests that the integral model having fewer shared variables resulted in small-sized components after the decomposition. The total number of events for a component in Table 5.3 is less than that of the same component in Table 5.1.

On the other hand, sub-models resulting from SED may have some events that do not make much sense on their own due to their other parts being present in another sub-model as a consequence of event sharing. For example, event moveFeedBelt in Table component of Figure 5.13 is only meaningful when presented along with its other half in the feed belt component. The sub-models resulting from SED could be modelled
independently without any restrictions. An extra preparatory refinement step is usually required up front in order to make the state variables disjoint to be decomposed using SED. This vertical refinement step would result in more variables in the model to refine the shared state into disjoint.

After decomposing the model into sub-models, we could then refine each of these sub-models independently. In case of SVD, we had to maintain the restrictions of the SVD style while refining these sub-models, i.e., to ensure that the shared variables and external events were not refined. We further refined the press sub-model vertically by introducing actuators and sensors to model it closer to implementation. This involved another three levels of refinement and was done using the refinement pattern for control systems [54].

We refined both of the press sub-models resulting from the two styles of decomposition, i.e., SVD and SED. The two press sub-models are different in a way that one has external events and shared-variables whereas the other only has local variables and shared events. Their refinement approach was similar since we followed the same refinement pattern for the two developments. We only discuss the press refinement resulting from SVD (i.e., Press_0) below as the other followed the same refinement structure. Figure 5.14 shows variables and internal events of this model (Press_0).
5.4.8 Press First Refinement Model (Press₁)

The press model (Press₀ of Figure 5.14) resulting from SVD of component-based PC (PC₄) was further refined by introducing actuators and sensors for handling the press. Since the press can have three positions (i.e., Low, Mid and High), there were three variables for the position sensors (i.e., pressPosLowSensor, pressPosMidSensor, pressPosHighSensor). A variable for an electric motor of the press (pressMotor) was introduced for its state, i.e., moving the motor forward, backward or stopped. We added events for turning the sensors on and off (i.e., pressLowSensorOn/Off, pressMidSensorOn/Off, pressHighSensorOn/Off). Events were added for starting and stopping the motor at different press positions (e.g., startPressMotorDown, stopPressMotorAtLowPos etc.). Figure 5.15 shows how the event movePressToLow was refined using atomicity decomposition in various refinement steps. So, an event for moving the press from high to low position in Press₀ was decomposed into four events in Press₁. The motor starts from high position that leads to the high position sensor turning off. When the low position sensor turns on, the motor is stopped. These events are shown in Figure 5.16. Other events were introduced in a similar way. Below are some of the invariants of this refinement model.

\[
\begin{align*}
\text{inv4:}& \; \text{pressPosMidSensor} = \text{on} \Rightarrow \text{pressPosLowSensor} = \text{off} \land \text{pressPosHighSensor} = \text{off} \land \text{pressPos} \in \{\text{low, mid}\} \\
& \quad // \text{Turn on mid sensor when the press arrives at mid position from low position} \\
\text{inv5:}& \; \text{pressPosLowSensor} = \text{on} \Rightarrow \text{pressPosMidSensor} = \text{off} \land \text{pressPosHighSensor} = \text{off} \land \text{pressPos} \in \{\text{high, low}\} \\
& \quad // \text{Turn on low sensor when the press arrives at the low position from high position} \\
\text{inv6:}& \; \text{pressPosHighSensor} = \text{on} \Rightarrow \text{pressPosMidSensor} = \text{off} \land \text{pressPosLowSensor} = \text{off} \land \text{pressPos} \in \{\text{mid, high}\} \\
& \quad // \text{Turn on high sensor when the press arrives at the high position from mid position} \\
\text{inv8:}& \; \text{pressMotor} = \text{bwd} \Rightarrow (\exists b \in \text{BLANKS} \land b \mapsto pr \in \text{position} \land \text{blanks}(b) = \text{forged}) \land \text{pressPos} = \text{high} \\
\text{inv9:}& \; \text{pressMotor} = \text{fwd} \land \text{pressPos} = \text{mid} \Rightarrow (\exists b \in \text{BLANKS} \land b \mapsto pr \in \text{position} \land \text{blanks}(b) = \text{unforged}) \\
\text{inv10:}& \; \text{pressMotor} = \text{fwd} \land \text{pressPos} = \text{low} \Rightarrow pr \notin \text{ran(position)}
\end{align*}
\]

Some gluing invariants were required to relate the abstract state to the new variables. For example, invariant ‘inv8’ shown above makes sure that the press will only move down if it is at the high position and has a blank on it, which has been processed. The invariant ‘inv9’ specifies that the press should only move up when it is at the mid position and has received a blank to be processed. Similarly, the invariant ‘inv10’ ensures that the press only moves up from the low position when the blank on it has been removed by the robot. We managed to discharge all the proof obligations. There were 83 POs generated by the Rodin, only 18 were discharged interactively and the rest were discharged automatically by the Rodin provers.
Figure 5.15: Event refinement for press component

Event \( \text{startMotorFromHigh} \) \( \triangleq \)
\[
\text{any } b \\
\text{where} \quad \\
\text{grd1}: \text{pressMotor} = \text{stop} \\
\text{grd2}: \text{pressPosHighSensor} = \text{on} \\
\text{grd5}: \text{pressPos} = \text{high} \\
\text{grd6}: b \in \text{dom}(\text{position}) \\
\text{grd7}: \text{position}(b) = \text{pr} \\
\text{grd8}: \text{blanks}(b) = \text{forged} \\
\text{then} \\
\text{act2}: \text{pressMotor} := \text{bwd} \\
\text{end}
\]

Event \( \text{highPosSensorOff} \) \( \triangleq \)
\[
\text{when} \\
\text{grd1}: \text{pressPosHighSensor} = \text{on} \\
\text{grd2}: \text{pressPos} = \text{high} \\
\text{grd3}: \text{pressMotor} = \text{bwd} \\
\text{then} \\
\text{act1}: \text{pressPosHighSensor} := \text{off} \\
\text{end}
\]

Event \( \text{lowPosSensorOn} \) \( \triangleq \)
\[
\text{when} \\
\text{grd1}: \text{pressPosHighSensor} = \text{off} \\
\text{grd2}: \text{pressPosMidSensor} = \text{off} \\
\text{grd3}: \text{pressPosLowSensor} = \text{off} \\
\text{grd4}: \text{pressMotor} = \text{bwd} \\
\text{grd5}: \text{pressPos} = \text{high} \\
\text{grd6}: \text{pr} \in \text{ran}(\text{position}) \\
\text{then} \\
\text{act1}: \text{pressPosLowSensor} := \text{on} \\
\text{end}
\]

Event \( \text{stopMotorAtLowPos} \) \( \triangleq \)
\[
\text{refines} \text{movePressToLow} \\
\text{when} \\
\text{grd1}: \text{pr} \in \text{ran}(\text{position}) \\
\text{grd2}: \text{pressPos} = \text{high} \\
\text{grd3}: \text{pressMotor} = \text{bwd} \\
\text{grd4}: \text{pressPosLowSensor} = \text{on} \\
\text{then} \\
\text{act1}: \text{pressPos} := \text{low} \\
\text{act2}: \text{pressMotor} := \text{stop} \\
\text{end}
\]

Figure 5.16: Events of Press.1 resulting from atomicity decomposition of \( \text{movePressToLow} \) event of Press.0
Event `stopPressMotorAtLowPos` refines `stopPressMotorAtLowPos`

<table>
<thead>
<tr>
<th>when</th>
<th><code>pr ∈ ran(position)</code></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>pressPos</code></td>
<td><code>= high</code></td>
</tr>
<tr>
<td><code>pressMotor</code></td>
<td><code>= bwd</code></td>
</tr>
<tr>
<td><code>prLowSensedFlag</code></td>
<td><code>= TRUE</code></td>
</tr>
<tr>
<td><code>pressLowSensed</code></td>
<td><code>= on</code></td>
</tr>
<tr>
<td>then</td>
<td><code>pressPos := low</code></td>
</tr>
<tr>
<td><code>pressMotor := stop</code></td>
<td></td>
</tr>
<tr>
<td><code>prLowSensedFlag := FALSE</code></td>
<td></td>
</tr>
</tbody>
</table>

Event `sensePressLowSensorVal` when

| `prLowSensedFlag` | `= FALSE` |
| `pressLowSensed` | `= pressPosLowSensor` |
| `prLowSensedFlag := TRUE` |

Figure 5.17: Events of Press_2 resulting from atomicity decomposition of `stopMotorAtLowPos` event of Press_1

5.4.9 Press Second Refinement Model (Press_2)

In the second refinement, using the sensing pattern from the cookbook [54], we refined the model so the controller uses the sensed values of the sensors. This is done to model the system closer to reality as the value of the sensors at some point in time will be different from the sensed values. We have not yet introduced the notion of time, which can be introduced later on. We added variables for sensed values of the sensors and a flag for each sensor to monitor whether a sensor has been sensed or not. Gluing invariants were added, e.g., a sensor flag will be true if the sensed value is same as the actual value of the sensor and false otherwise, as shown below:

\[
\text{inv7}: \text{prMidSensedFlag = TRUE } \Rightarrow \text{pressMidSensed = pressPosMidSensor}
\]
\[
\text{inv8}: \text{prLowSensedFlag = TRUE } \Rightarrow \text{pressLowSensed = pressPosLowSensor}
\]
\[
\text{inv9}: \text{prHighSensedFlag = TRUE } \Rightarrow \text{pressHighSensed = pressPosHighSensor}
\]

The event decomposition diagram of Figure 5.15 shows how `stopMotorAtLowPos` event of Press_1 was decomposed into two events in Press_2 for sensing the sensor value (new event: `SenseLowPosSensor`) before stopping the motor (refined event:`StopMotorAtLowPos`), as shown in Figure 5.17. Similarly, other events were introduced for different sensors. This was a smaller refinement step as compared to Press_1. There were 21 POs and all were discharged automatically.

5.4.10 Press Third Refinement Model (Press_3)

At this level of refinement, the actuation was refined where a controller sets the actuation of a motor before the motor is actuated. Again, this brings the model closer to implementation as in reality there will be a delay in setting the actuator and its actual movement based on the actuation set by the controller. An abstract actuation event is decomposed into two events for a controller and actuator. For example, in Figure 5.15,
Table 5.4: Number of events at different refinement levels of press component decomposed using SVD and SED

<table>
<thead>
<tr>
<th>Model</th>
<th>SVD-based Events</th>
<th>SED-based Events</th>
</tr>
</thead>
<tbody>
<tr>
<td>Press_0</td>
<td>16</td>
<td>6</td>
</tr>
<tr>
<td>Press_1</td>
<td>25</td>
<td>15</td>
</tr>
<tr>
<td>Press_2</td>
<td>28</td>
<td>18</td>
</tr>
<tr>
<td>Press_3</td>
<td>34</td>
<td>22</td>
</tr>
</tbody>
</table>

Table 5.5: Proofs statistics of component-based PC modelled for SED/SVD

<table>
<thead>
<tr>
<th>Model</th>
<th>SVD-based</th>
<th>SED-based</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC_0</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>PC_1</td>
<td>43</td>
<td>29</td>
</tr>
<tr>
<td>PC_2</td>
<td>25</td>
<td>21</td>
</tr>
<tr>
<td>PC_3</td>
<td>43</td>
<td>30</td>
</tr>
<tr>
<td>PC_4</td>
<td>10</td>
<td>18</td>
</tr>
<tr>
<td>Press_1</td>
<td>83</td>
<td>57</td>
</tr>
<tr>
<td>Press_2</td>
<td>21</td>
<td>20</td>
</tr>
<tr>
<td>Press_3</td>
<td>107</td>
<td>89</td>
</tr>
</tbody>
</table>

the event \textit{stopMotorAtLowPos} of \textit{Press_2} is split into \textit{stopMotorAtLowPosCtrl} (new event) and \textit{stopMotorAtLowPosAct} (refined event) in \textit{Press_3}. These two events are shown in Figure 5.18. There were 107 POs generated at this refinement level where most of these POs were discharged automatically with a few discharged interactively. This model can be further refined vertically by introducing the notion of time.

Table 5.4 shows the number of events at each refinement level of the \textit{press} component decomposed using SVD and SED. Table 5.5 shows the number proof obligations at each refinement level for modelling component-based PC from the view of SVD/SED. Other components (e.g., \textit{table}, \textit{robot}, \textit{feed belt} etc.) can also be modelled in the same way as the \textit{press}, using the refinement pattern for control systems. Their refinement will be similar to the press as we will have to model the actuators and sensors for these components in the same way.

5.4.11 Building Variants of PC

The component-based modelling approach discussed above gives us a product of PC that models a particular topology. This topology is presented in Figure 5.19, which shows the component-based PC model at fourth refinement level, before decomposition into various components. The figure shows how the physical components are connected with
Each different topology is an instance of PC product line and we can build more variants of PC by selecting a different configuration (or topology) of these physical components. For example, consider a production cell with two press components for processing blanks twice and using two robots. We can call this ‘topology 2’ where robot1 picks a blank from the table and drops onto press1 and robot2 takes the blank from robot1, which picks it from press1 and drops on press2 (see Figure 5.20). Here we are interested in exploring to what extent we can reuse the models of topology1 while modelling topology2 and
hence the proof effort. In terms of the requirements specification listed in Appendix A, we reused most of the requirements of topology1 with the requirements for the robot and press components were duplicated. The additional requirement is the collection of blanks by robot2 from robot1 for processing in the second press.

We had to do instantiation and refactoring to simply duplicate the functionality of existing components for modelling topology2. This means that we would not have to reprove the models, which have already been proved for topology1. This is because renaming of elements would not affect the proof obligations (POs) and is currently supported by the refactoring plug-in [127] in Rodin. We only had to prove the POs generated for any additional information modelled in the second topology. For example, we specify that arm1 of robot2 collects the blank from arm2 of robot1 unlike picking it from the table in topology1.

For topology2 development, we started with the abstract model of topology1 and duplicated the events Operate for processing blanks twice, i.e. events Operate1 and Operate2. Both events could be achieved through refactoring. We added an additional guard in Operate2, which ensures that it happens after Operate1 event, as shown below:
In the first refinement, we duplicated the functionality for press and robot, which means we had to replicate all the events related to both of these components. Figure 5.21 shows events of topology1 and that of topology2 at first refinement level. We also had to duplicate variables and invariants while duplicating a component. For example, a variable \texttt{arm1State} in Topology1 became \texttt{arm1r1State} and \texttt{arm1r2State} for robot1 and robot2 respectivey in Topology2 along with duplication of their typing and other invariants, as shown below:

**INVARIANTS**

//Topology 1

\begin{align*}
\text{inv1: } & \text{arm1State} \in \text{ARMSTATE} \\
\text{inv5: } & \text{arm1State} = \text{extended} \Rightarrow (\text{robotPos} = \text{pos1} \lor \text{robotPos} = \text{pos3})
\end{align*}

**INVARIANTS**

//Topology 2

\begin{align*}
\text{inv1: } & \text{arm1r1State} \in \text{ARMSTATE} \\
\text{inv2: } & \text{arm1r2State} \in \text{ARMSTATE} \\
\text{inv7: } & \text{arm1r1State} = \text{extended} \Rightarrow (\text{robot1Pos} = \text{pos1} \lor \text{robot1Pos} = \text{pos3}) \\
\text{inv9: } & \text{arm1r2State} = \text{extended} \Rightarrow (\text{robot2Pos} = \text{pos1} \lor \text{robot2Pos} = \text{pos3})
\end{align*}

Figure 5.22 show events of \textit{topology1} and that of \textit{topology2} as a result of refactoring at second, third and fourth level of refinement.

Figure 5.23 shows the refinement architecture for modelling the two topologies and their components as achieved after decomposition. An example of event instantiation while modelling \textit{topology2} after \textit{topology1} is shown in Figure 5.24 where event \texttt{loadPress} is
Chapter 5 Case Study - Production Cell

Figure 5.21: Events refactoring for modelling topology2 after topology1 - 1st refinement

Figure 5.22: Events refactoring for modelling topology2 after topology1 - 2nd, 3rd and 4th refinement
duplicated for the two presses (events: loadPress1 & loadPress2). This type of instantiation and refactoring has no proof burden. Figure 5.26 shows the proof obligations for the events that were refactored when modelling the second topology along with the additional POs. Figure 5.25 shows another example of event instantiation for loading robot arm1 in topology1 (i.e., event loadArm1). Here, the instantiation is not trivial. We refactor the instantiated events for arm1 of robot1 (loadArm1R1) and robot2 (loadArm1R2), add and modify some guards and actions accordingly. This would require us to prove that the refactored events preserve invariants.

The POs for topology2 were discharged in the same way as topology1. Table 5.6 shows the number of POs for both topology developments at different refinement levels and how these were discharged, i.e., automatically or interactively. It also shows the amount of newly generated POs, reused (i.e., appear as these were in topology1) when modelling the second topology and those generated as a result of refactoring. It shows the percentage of reused POs as well. This means that refactoring utility could save the effort of reproving these refactored POs. Table 5.7 shows the number of events and invariants for the two topology developments along with the number of events reused from topology1 (i.e., appear without any modification) and those refactored (i.e., duplicated along with some renaming). This experiment shows that we can reuse existing models and their proofs if we have tools to automate the instantiation and refactoring processes; hence, generating additional tooling requirements, which are discussed later.

5.4.12 Evaluation

We have discussed component-based modelling approach for PC. We refined PC both horizontally and vertically and decomposed the integral model into various components using both styles of Event-B decomposition (SVD/SED). After decomposition, we refined the press component further to model actuators and sensors in various refinement steps,
Figure 5.24: Event instantiation example for PC topology2

Figure 5.25: Event instantiation example for PC topology2
Figure 5.26: Proof obligations refactoring for an event of topology 2

Table 5.6: Proof obligations statistics for two topology developments
using the pattern for modelling control systems in Event-B [54]. Other components
could be modelled in the same way and hence not discussed here. In order to explore
how we could reuse models of this development (which we call topology1), we modelled
another variant of PC that has more components than in the first one. Here we wanted
to model a PC that has two presses and two robots to process a blank twice. This
was done by simply duplicating the functionality of these two components, which means
we had to duplicate the events related to these components through refactoring. We
can define variation points as the connecting/shared events (events connecting boxes
in Figure 5.19). These variation points may vary for a different topology. This means
that these variation points would behave differently unlike event duplication (which is
simple refactoring to make them disjoint). We have shown the reuse statistics when
modelling topology2 after topology1, which clearly shows the amount of reuse achieved.
This includes parts of Event-B specification (events) as well as proofs so that the proof
effort could also be avoided.

This topology development approach seems feasible only when we need to duplicate a
couple of components or to model a couple of topologies. Imagine a case where we
have to model a topology having many instances of a component (e.g., 10 presses and
10 robots). This would become very hard to model, refine and manage the overall
development with the current state of tool support. One way to deal with the issue of
modelling any number of topologies with multiple replication of components is to raise
the abstraction level of our base topology model. We can specify generic components
and generic events in our model that could be specialised as and when required to model
various components of a particular topology through reuse and generic instantiation. For
example, we can have an event that models the transfer of blanks from one component
to another (i.e., passBlankBtwCpt1Cpt2) as shown below:

Table 5.7: Events and invariants statistics for topology 1 and 2

<table>
<thead>
<tr>
<th>PC Topology 1</th>
<th>PC Topology 2</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Ref. level</strong></td>
<td><strong>Invariants</strong></td>
</tr>
<tr>
<td>PC-0</td>
<td>1</td>
</tr>
<tr>
<td>PC-1</td>
<td>3</td>
</tr>
<tr>
<td>PC-2</td>
<td>7</td>
</tr>
<tr>
<td>PC-3</td>
<td>10</td>
</tr>
<tr>
<td>PC-4</td>
<td>3</td>
</tr>
<tr>
<td>PC-5</td>
<td>10</td>
</tr>
<tr>
<td>PC-6</td>
<td>9</td>
</tr>
<tr>
<td>PC-7</td>
<td>14</td>
</tr>
</tbody>
</table>
Chapter 5 Case Study - Production Cell

Event passBlankBtwCpt1Cpt2 =

any
b

where
grd1 : b ∈ blanks
grd2 : \text{position}(b) = Cpt1
grd1 : Cpt2 ∉ \text{ran}(\text{position})

then
act1 : \text{position}(b) := Cpt2

end

This event could be specialised for passing blanks between any two components of the plant where we could refactor \textit{Cpt1} and \textit{Cpt2} with feed belt and table and so on for the rest of the components. This generic base topology could be refined to model internal mechanise of a component. For example, in a refinement model, a generic event for moving a component between two positions could be introduced. This event could then be specialised for table or crane components, etc.

This experiment generated further tool requirements and also helped us suggest guidelines for users (discussed later) who would like to benefit from reuse when modelling control systems in Event-B.

5.5 PC Controller-based

In the controller-based PC modelling, functional requirements for modelling the behaviour of each controller of the production cell were grouped together as a feature. Therefore, the requirements specification (given in Appendix A) was decomposed into various controller features. We also generalised the requirements for each of the controller so that we could model generic controllers; which could then be specialised and reused for modelling various controllers of different physical components of the PC. Hence, the controller-based modelling of PC was a result of decomposition plus generalisation. Figure 5.27 shows Event-B refinement architecture for controller-based PC modelling.

Table 5.8 shows part of the requirements specification for the \textit{table} feature of component-based PC (column 1) and the \textit{movement} feature for controller-based PC (column 2). This shows how we can define the feature in terms of requirements for two styles of modelling the PC, while making the features more reusable. The compositional requirements are implemented while actually composing various components; this may include topological information and how components are connected together. Similarly, Table 5.9 shows part of the requirements specification for the \textit{crane} feature of component-based PC (column 1), which can be modelled by specialising the requirements of the generic \textit{magnet} feature for controller-based PC (column 2). The controller-based PC models consisted of \textit{loader},
Table 5.8: Requirements description for Table and Movement features of PC

<table>
<thead>
<tr>
<th>Table Component</th>
<th>Generic Movement Controller</th>
</tr>
</thead>
<tbody>
<tr>
<td>Req#6 - Move table upwards/downwards</td>
<td>- Move a component from position A to position B and vice-versa</td>
</tr>
<tr>
<td>Req#6 - Rotate table clockwise/anti-clockwise</td>
<td>Instantiation Requirements</td>
</tr>
<tr>
<td>Req#9 - The table must not rotate clockwise if it is in a position to deliver</td>
<td>- Extend/Retract Arm1</td>
</tr>
<tr>
<td>blanks (unloading position)</td>
<td>- Extend/Retract Arm2</td>
</tr>
<tr>
<td>Req#10 - Table must not rotate when its at low position</td>
<td>- Move Feed Belt/Deposit Belt</td>
</tr>
<tr>
<td>Req#11 - Table must not move down if it is rotated (rotate backward first and</td>
<td>- Move the Table upwards/ downwards</td>
</tr>
<tr>
<td>then move down) or if it is already not elevated</td>
<td>- Move Crane To and from Feed Belt/Deposit Belt</td>
</tr>
<tr>
<td>Compositional Requirements</td>
<td>Compositional Requirements</td>
</tr>
<tr>
<td>Req#7 - Drop blank on table from feed belt when it is in the loading position</td>
<td>- Extend/Retract Arm1 if robot is facing table or facing press while press is in middle</td>
</tr>
<tr>
<td>(not elevated and not rotated)</td>
<td>position</td>
</tr>
<tr>
<td>Req#8 - Robot picks blank from table when it is in unloading position (elevated</td>
<td>- Extend/Retract Arm2 if robot is facing deposit or facing press while press is in lower</td>
</tr>
<tr>
<td>and rotated)</td>
<td>position</td>
</tr>
<tr>
<td></td>
<td>Table must not move down if its rotated (rotate backward first and then move down) or if</td>
</tr>
<tr>
<td></td>
<td>it is already not elevated</td>
</tr>
<tr>
<td></td>
<td>- Crane should only move towards feed belt if it is positioned on deposit belt and</td>
</tr>
<tr>
<td></td>
<td>vice-versa</td>
</tr>
</tbody>
</table>

movement, rotation and magnet controllers. A member of PC product line could be modelled by instantiating and composing these controller-based reusable features. These features were then refined independently. We only discuss the refinement of magnet and movement features below where we introduced sensors and actuators in various refinement steps using the pattern for refining control systems as suggested in [54].

Figure 5.27: Controller-based PC modelling
Table 5.9: Requirements description for Crane and Magnet features of PC

<table>
<thead>
<tr>
<th>Crane Component</th>
<th>Generic Magnet Controller</th>
</tr>
</thead>
<tbody>
<tr>
<td>Req#35 - A travelling crane is used to bring the unforged blanks back from the</td>
<td>- Pick and drop a blank</td>
</tr>
<tr>
<td>deposit belt to the feed belt</td>
<td>Instantiation Requirements</td>
</tr>
<tr>
<td>Req#36 - The crane has an electromagnet gripper to pick and drop the blank</td>
<td>- Pick and drop functionality for Arm1</td>
</tr>
<tr>
<td>Compositional Requirements</td>
<td>- Pick and drop functionality for Arm2</td>
</tr>
<tr>
<td>Req#39 - The crane picks a blank from deposit belt if there is one at the end of</td>
<td>- Pick and drop functionality for Crane</td>
</tr>
<tr>
<td>the belt</td>
<td></td>
</tr>
<tr>
<td>Req#40 - The crane drops a blank on to the feed belt if it is not already full</td>
<td></td>
</tr>
</tbody>
</table>

5.5.1 Magnet Controller

At the abstract level, we have events for picking and dropping of blanks by a component. A component that has not already picked a blank can do so and a component that has picked a blank can drop it as shown below:

\[
\text{Event } X\text{compXPickBlank} \triangleq \\
\text{any } b \\
\text{where } \\
\text{grd}_1: X\text{compX} \notin \text{ran}(\text{position}) \\
\text{grd}_2: b \in \text{dom}(\text{position}) \\
\text{then } \\
\text{act}_1: \text{position}(b) := X\text{compX} \\
\text{end}
\]

\[
\text{Event } X\text{compXDropBlank} \triangleq \\
\text{any } b \\
\text{where } \\
\text{grd}_1: X\text{compX} \in \text{ran}(\text{position}) \\
\text{grd}_2: b \in \text{dom}(\text{position}) \\
\text{then } \\
\text{act}_1: \text{position} := \text{position} \setminus \\
\{ b \mapsto X\text{compX} \} \\
\text{end}
\]

The notation ‘X\text{compX}’ means a generic placeholder for a component, which must be filled with appropriate component during specialisation. So, this feature will be instantiated to a specific component such as a crane or a robot arm. The model is quite abstract and the details were added later in the refinements and during specialisation. Figure 5.28 shows how the two events of this abstract model are refined along with the introduction of new events in three refinements steps.

In the first refinement, we added sensor functionality for magnet that informs the controller whether a blank has been picked up or dropped off. An electromagnet switch acts as an actuator for the magnet, which performs the pick and drop of blanks. Two variables were introduced as following:

\[
\text{inv}_1: X\text{compXMagnetSensor} \in \text{SENSOR} \\
\text{inv}_2: X\text{compXMagnet} \in \text{ELECMAGNET}
\]
We also added events for starting and stopping the magnet and switching the sensor on and off as shown below:

\[
\text{Event } \text{XcompXstartMagnet} \triangleq \text{refines } \text{XcompXPickBlank} \\
\text{any } b \\
\text{where} \\
\quad \text{grd1: } \text{XcompX} \notin \text{ran(position)} \\
\quad \text{grd2: } b \in \text{dom(position)} \\
\quad \text{grd3: } \text{XcompXMagnet} = \text{drop} \\
\quad \text{grd4: } \text{XcompXMagnetSensor} = \text{off} \\
\text{then} \\
\quad \text{act1: position}(b) := \text{XcompX} \\
\quad \text{act2: } \text{XcompXMagnet} := \text{pick} \\
\text{end} \\
\text{Event } \text{XcompXsensorOn} \triangleq \\
\text{when} \\
\quad \text{grd1: } \text{XcompXMagnetSensor} = \text{off} \\
\quad \text{grd2: } \text{XcompXMagnet} = \text{pick} \\
\text{then} \\
\quad \text{act1: } \text{XcompXMagnetSensor} := \text{on} \\
\text{end} \\
\text{Event } \text{XcompXsensorOff} \triangleq \\
\text{when} \\
\quad \text{grd1: } \text{XcompXMagnetSensor} = \text{on} \\
\quad \text{grd2: } \text{XcompXMagnet} = \text{pick} \\
\text{then} \\
\quad \text{act1: } \text{XcompXMagnetSensor} := \text{off} \\
\text{end} \\
\]

In the second refinement, we differentiate between the actual and sensed values of the sensors. So, a boolean variable \text{XcompXMagnetSensorFlag} (shown below) was introduced to
ensure that the magnet’s start event takes place when both the sensed and actual values are same.

\[ \text{inv4} : \ X\text{compXMagSensorFlag} = \text{TRUE} \Rightarrow X\text{compXMagSensedVal} = X\text{compXMagnetSensor} \]

Similarly, in the third refinement, we refined the actuation mechanism where controller sets the actuation of the motor before the motor actually starts moving. Here we split the actuation events into two, i.e., a new event for setting the actuation of magnet by the controller and a refined event for magnet to actuate accordingly. Figure 5.29 shows invariants and events for the third refinement model of magnet controller. For example, \( X\text{compXStartMagnetCtrl} \) is a new event which tells the plant to start magnet for picking a blank by turning on its flag and is then followed by the event \( X\text{compXStartMagnetAct} \) which actually picks the blanks. Similarly, \( X\text{compXStopMagnetCtrl} \) is a new event to stop magnet for dropping a blank by turning on its flag which then leads to the event \( X\text{compXStopMagnetAct} \) for actually dropping the blank. The sensor events occur after a start or stop magnet event takes place. The refinement style is similar to that of the press refinement discussed earlier, since both of these use the same refinement pattern of [54].

### 5.5.2 Movement Controller

The development structure of movement controller is similar to that of the magnet controller. At the abstract level, we have events for moving a physical component forward and backward between two positions (i.e., \( posA, posB \)), as shown below:

\[
\begin{align*}
\text{Event} & \quad X\text{compXMoveToXposBX} \\ & \quad \text{when} \quad \text{grd1} : X\text{PosVarX} = X\text{posAX} \\ & \quad \text{then} \quad \text{act1} : X\text{PosVarX} := X\text{posBX} \\ \end{align*}
\]

\[
\begin{align*}
\text{Event} & \quad X\text{compXMoveToXposAX} \\ & \quad \text{when} \quad \text{grd1} : X\text{PosVarX} = X\text{posBX} \\ & \quad \text{then} \quad \text{act1} : X\text{PosVarX} := X\text{posAX} \\ \end{align*}
\]

This feature will be instantiated to a specific component such as a crane or a table. In case of crane feature, \( posA \) can become deposit belt and \( posB \) can be instantiated as feed belt. During the first refinement, we added sensors for the two positions and a motor for moving the component backward and forward. Events were added for starting and stopping the motor at different positions and switching the sensors on and off. Figure 5.30 shows events and invariant of the first refinement level. Here, we have events for starting and stopping the motor in forward and backward direction and turning the sensors on and off at positions \( A \) and \( B \). In the second refinement, we differentiate between the actual and sensed values of the sensors as discussed earlier. Using the
while providing topological information. For example, if we want to model the crane component, we have to specialise one instance of the crane component (e.g., phases. In the first phase, we specialise and compose generic models to build a composition between setting the motor’s actuation by the controller from its actual movement. The magnet controllers provide us refinement chains of generic Event-B models for the two features. In order to model any component of the PC, we need to instantiate and compose these chains of models. The composition is done in two phases. In the first phase, we specialise and compose generic models to build a component (e.g., crane or robot) and in the second phase, we compose all the components while providing topological information. For example, if we want to model the crane component, we have to specialise one instance of the magnet controller to pick and drop

Figure 5.29: Magnet controller third refinement model

same refinement style of magnet controller, at third level of refinement, we differentiate between setting the motor’s actuation by the controller from its actual movement.

5.5.3 Instantiation & Composition

The magnet and movement controllers provide us refinement chains of generic Event-B models for the two features. In order to model any component of the PC, we need to instantiate and compose these chains of models. The composition is done in two phases. In the first phase, we specialise and compose generic models to build a component (e.g., crane or robot) and in the second phase, we compose all the components while providing topological information. For example, if we want to model the crane component, we have to specialise one instance of the magnet controller to pick and drop.
### INVARIANTS

inv1: XcompXmotor ∈ MOTOR
inv2: XcompXSensorPosA ∈ SENSOR
inv3: XcompXSensorPosB ∈ SENSOR
inv4: XcompXSensorPosA = on ⇒ XcompXSensorPosB = off
inv5: XcompXSensorPosB = on ⇒ XcompXSensorPosA = off
inv6: XcompXmotor = stop ∧ XPosVarX = XposAX ⇒ XcompXSensorPosA = on
inv7: XcompXmotor = stop ∧ XPosVarX = XposBX ⇒ XcompXSensorPosB = on

### EVENTS

**Event** XcompXStartMotorFwd ≡

when
grd1: XcompXSensorPosA = on
grd2: XcompXmotor = stop
then
act1: XcompXmotor := fwd
end

**Event** XcompXStopMotorFwd ≡

extends XcompXMoveToXpos2X
when
grd1: XPosVarX = XposAX
grd2: XcompXmotor = fwd
grd3: XcompXSensorPosB = on
then
act1: XPosVarX := XposBX
act2: XcompXmotor := stop
end

**Event** XcompXStartMotorBwd ≡

when
grd1: XcompXSensorPosB = on
grd2: XcompXmotor = stop
then
act1: XcompXmotor := bwd
end

**Event** XcompXStopMotorBwd ≡

extends XcompXMoveToXpos1X
when
grd1: XPosVarX = XposBX
grd2: XcompXmotor = bwd
grd3: XcompXSensorPosA = on
then
act1: XPosVarX := XposAX
act2: XcompXmotor := stop
end

**Event** XcompXSensorPosBOn ≡

when
grd1: XcompXSensorPosB = off
grd2: XcompXSensorPosA = off
then
act1: XcompXSensorPosB := on
end

**Event** XcompXSensorPosBOff ≡

when
grd1: XcompXSensorPosB = on
grd2: XcompXSensorPosA = off
grd3: XcompXmotor = bwd
then
act1: XcompXSensorPosB := off
end

**Event** XcompXSensorPosAOn ≡

when
grd1: XcompXSensorPosA = off
grd2: XcompXSensorPosB = off
then
act1: XcompXSensorPosA := on
end

**Event** XcompXSensorPosAOff ≡

when
grd1: XcompXSensorPosA = on
grd2: XcompXSensorPosB = off
grd3: XcompXmotor = fwd
then
act1: XcompXSensorPosA := off
end

END

---

Figure 5.30: Movement controller events and invariants for first refinement

blanks and two instances of the movement controllers for moving the crane horizontally and vertically, as shown in Figure 5.31. In this example, we have three refinements in each development which align well during the composition. This alignment issue (also discussed in previous chapter) needs to be explored further to address the composition of Event-B developments having different number of refinements. Figure 5.32 shows a simple example where event PickBlank of magnet controller is specialised for the crane component. Here the generic model parameter XcompX is replaced by crane, provided both of these are of the same type. For now, we use X...X as a syntactic convention to model a generic parameter, given that the current Rodin tool does not support generics. This instantiation has no proof burden and the instantiated model will be correct by construction. Figure 5.33 shows another example where event XcompXMoveToXposBX of movement controller is specialised as craneMoveToPosFB event of the crane component, for moving it from deposit belt to feed belt.
The composition of abstract level models from each refinement chain would give us an abstract specification for the crane. We also had to do some guard strengthening and add some invariants during the composition. This would generate addition POs and some of the exiting POs may also change that would require reproving. The composition of implementation level models for each refinement would provide us with the implementation of the crane. Again extra guards for events and invariants were needed. Figure 5.34 shows the events cranePickBlank and craneMoveToPosFB (of Figures 5.32 & 5.33) with extra guards added during the composition. For example, \( \text{grd3} \) of cranePickBlank event
specifies the topological information that the crane can only pick a blank when it is positioned on the deposit belt. Similarly, \( \text{grd2} \) of \( \text{craneMoveToPosFB} \) event in Figure 5.34 specifies that the crane can only move towards the feed belt if it has picked up a blank. The guard \( \text{grd2} \) of \( \text{cranePickBlank} \) event means it can pick any blank in the system. Table 5.10 shows P0s for three crane components and the crane model resulting from their composition.

When we finally compose all the components to model the entire PC in the second phase of composition, we will need to strengthen this guard to say that the crane can only pick a blank from the deposit belt. Here we would need to give topological information of PC in terms of how different components are connected to each other as discussed in the components-based PC example earlier. Again by adding extra guards and invariants in this second phase of composition may generate additional P0s to be reproved.
We call this style of composition ‘feature composition’ where additional information can be added during the composition. As of yet, this style of composition does not guarantee refinement preservation between the composed abstract and implementation models (see ‘refines?’ in Figure 5.31). In order to deal with this kind of composition, we need the support for proof reuse. By this we mean to find a way of automatically discharging composite POs with the help of already discharged POs of the components being composed. This requires further work. Although, we will discuss an alternative approach in the next chapter where we can avoid reproof by following a particular modelling pattern, that approach would not be feasible for modelling generic components of PC due to its blank passing topological nature.

We had to use feature composition because the SED could not be applied here due to the shared variables between the components being composed. Similarly, the SVD approach is too constraining and could only be used here if we start with an abstract model containing the functionality of both the magnet and movement features. We could then decompose these into two developments (both having external events), refine each of these, instantiate for the crane and compose to build the crane model. This would require all the additional components to be modelled in the same manner, which would limit the reuse and benefits of genericity. The ATM case-study discussed in the next chapter further explores these issues and suggests the modelling style through which we could use existing techniques of Event-B to achieve partial reuse of existing specifications, when modelling variants of a product line.

5.5.4 Evaluation

The controller-based PC modelling discussed above showed how we can improve reusability by modelling a system as a set of generic features. These feature could then be specialised for modelling various components of the system. These components when composed together would result in a particular product of a product line. In comparison to the component-based approach discussed earlier, this style of modelling SPLs in Event-B seems more appropriate for product line modelling because it provides more reuse opportunities. This approach is modelled at a different abstraction level compared to the component-based approach where reuse only lies in terms of components and their connectivity. Here, we model and reuse fine-grained features as compared to coarse-grained features of component-based PC.

For this approach, we had to generalise requirements and group these together to model as features. We showed how generic features for movement and magnet controllers could be instantiated and composed to model the crane component of PC. These generic features along with others could be instantiated and composed to model a robot and similarly rest of the PC components. These components could then be composed to build integral model of PC where topological information is provided for connecting these
components. Once we have an integral model of the PC, this could again be decomposed using either of the two decomposition techniques of Event-B (i.e., SVD/SED), depending on how the system is to be implemented. This experiment requires us to reprove the instantiated product since this middle-in loose-structured composition - which we call feature composition - does not guarantee refinement preservation. We will address this issue in the next chapter by suggesting a different approach when modelling another case-study. We will also discuss how this approach could be generalised in our suggested guidelines for future users in Chapter 7.

5.6 PC – Domain-specific SPL modelling through contexts

We also modelled a generic component-based PC which supports the two topologies mentioned earlier. The variability is provided through the context which means the machine for both the topologies will remain same and we could have a different topology by just switching the contexts. Figure 5.36 shows this development architecture where a generic product is specialised by switching the contexts to build specialised products. We modelled the topology of PC in the context, i.e., we specify the physical components and how these could be connected to each other. The machine is modelled in a generic way as shown in Figure 5.37. The event passBlankBtwCpts models how a blank is passed from one component to another and the event moveCpt models the movement
of a component itself as the components of PC can have different positions (e.g., press moves between three positions and table moves between two positions). The invariant ‘inv1’ defines position of blank which can be at only one component at a time. The invariant ‘inv2’ specifies the position of a component which can be any one of its possible positions, e.g., press can be in low, mid or high position and table can be in up or down position.

The topological information is modelled in the contextual part at the bottom of the figure where \textit{cptGraph} defines the components graph, i.e., valid ways of connecting different components. This is then used in ‘\textit{grd4}’ of the \textit{passBlankBtwCpts} event to make sure that the blank is passed between connectable components. If we like to model a different topology (e.g., \textit{topology2} as discussed earlier) with two robots and two presses, we would simply need to modify the context, i.e., include these extra components to the set CPT and update the component connectivity information accordingly in the \textit{cptGraph}. We can refine this model further to introduce sub-component functionality of a component. For example, a robot has two sub-components for its arms. This could then allow us to model a robot having three arms by modifying the context and using the same machine.

5.6.1 Evaluation

The advantage of modelling in this way is that we will not need to reprove a variant of PC resulting from static variability through context switching. The disadvantage is that the modeller may not visualise various events of the machine for a particular topology by looking at the model unless the machine is animated. For example, in component-based approach, we have a separate event for each component to pass a blank to another component, whereas in this approach, we have only one event representing the same functionality. This could be a useful domain modelling activity for exploring variability of a product line in a distributed environment. This development could also be considered as a base for a product line and all of its products must be derived from this base development and then refined. Further work in this direction is required to explore this concept with different case-studies.

Another possibility is to refine this generic model down to a level where we introduce all the sensors and actuators; and in order to model a particular topology of PC, we could then instantiate events as required and switch contexts accordingly. This again would not need reproof effort unless additional information is required when instantiating various events of the development.
INVARIANTS
inv1 : blankPosition ∈ BLANKS ↦ → CPT
inv2 : cptPosition ∈ CPT → POS

EVENTS
Event passBlankBtwCpts ≡
any b, cpt1, cpt2
where
  grd1 : b ∈ BLANKS
  grd2 : cpt1 ∈ CPT
  grd3 : cpt2 ∈ CPT
  grd4 : cpt1 ↦→ cpt2 ∈ cptGraph
  grd5 : b ∈ dom(blankPosition)
  grd6 : blankPosition(b) = cpt1
  grd7 : cpt2 ∉ ran(blankPosition)
then
  act1 : blankPosition(b) := cpt2
end

Event moveCpt ≡
any pos1, pos2, cpt
where
  grd1 : pos1 ∈ POS
  grd2 : pos2 ∈ POS
  grd3 : cptPosition(cpt) = pos1
then
  act1 : cptPosition(cpt) := pos2
end

CONSTANTS
fb, tbl, arm1, arm2, pr, db, cr
cptGraph

AXIOMS
axm1 : partition(CPT, {fb}, {tbl}, {arm1}, {arm2}, {pr}, {db}, {cr})
axm2 : cptGraph ∈ CPT ↦→ CPT
axm3 : cptGraph = {fb ↦→ tbl, tbl ↦→ arm1, arm1 ↦→ pr, pr ↦→ arm2,
        arm2 ↦→ db, db ↦→ cr}
axm4 : partition(POS, {up}, {down}, {low}, {mid}, {high})

Figure 5.37: Partial Event-B model of domain-specific modelling through context switching

5.7 Conclusion

By modelling production cell in three different ways, we have explored to what extent we can use existing Event-B tools and techniques for feature-based product line modelling. This also enabled us to figure out the requirements for future tooling and techniques (discussed in Chapter 7) that can further facilitate such development approach to benefit from reuse of existing models and their proofs.

The first style of modelling – component-based – is a natural approach of modelling in Event-B which used both types of decomposition techniques to reduce the complexity
of modelling and proving by decomposing large models into smaller sub-models. We started with an integral model of PC and refined this up to four refinements. We then decomposed this model into various physical components of PC. Each of these components could then be refined separately. We only refined the press component of PC where we introduced actuators and sensors to model it closer to implementation. In order to explore variability, we modelled a variant of PC having two of the press and robot components each. This provided us with another topology of PC. So, we could specify the commonality and variability in terms of the components and the topology that can be used to connect these components respectively. Further variants of PC could be modelled by reusing existing models and altering the topology as required. We have shown reuse statistics in terms of specification and proofs when modelling two products of PC having different topologies. This clearly shows that we can significantly reduce modelling and proof effort through reuse. The downside of this approach is that the variability only exists in the number of components and their connections, i.e., a variation of PC could be modelled through reuse by replicating the functionality of existing components and how these would be connected to each other. In order to add a new component, we will have to model the PC again from abstract specification and doing the proof effort again. This approach is only feasible when we duplicate small number of components, otherwise it becomes very difficult to manage the development with the current tool support. This experiment helped us in exploring requirements for instantiation and refactoring tool support that could be useful in automating this approach.

The second approach of controller-based modelling is more feature-oriented as we have modelled generic reusable features that could be instantiated and composed in different ways to model different PC components and hence benefit from their reuse. We generalised the requirements of PC and grouped these into several features. These generic features could then be modelled and refined independently. We could then specialise and compose these generic features to model a product of PC product line. We call the composition required in this modelling style as feature composition. This is a loose-structured cut-and-paste composition type which suits the feature-based modelling. Specialisation of generic features does not require reproof effort but this composition does not guarantee refinement preservation as we have to provide additional information during composition. This approach could be very useful if we can figure out proof reuse mechanism, i.e., how to discharge POs of composite model by analysing and reusing POs of the models being composed? However, we can avoid this proof reuse problem by using a modelling pattern suggested in the next chapter. Note that the existing composition techniques of Event-B could not be applied in this style of modelling.

The third approach of modelling static variability through context switching allows us to evaluate the scope of a product line and without doing the proof effort upfront. This is another way of modelling component-based PC and suits the product line development
approach as we can figure out the common base for different variants (topologies) of the PC, and the configuration or the variability can be embedded in the context. This could be useful to foresee how a product line would evolve for a particular domain and later on could be modelled in one of the two styles mentioned to build a database of reusable features.

The case-study presented in this chapter suggests that we can use existing tools and techniques (to some extent) for feature-based development using Event-B. But there are certain restrictions of the existing (de)composition techniques, which must be followed and that restrict the feature-based development in terms of reuse. We have highlighted another form of composition – feature composition – which provides a less restrictive and more suitable form of composition for feature-based development. It does not support proof reuse as of yet and that requires further research work in future. This problem of proof reuse during composition can be overcome by following a particular modelling pattern, which is explained in the next chapter using the ATM case-study. The suggested modelling pattern uses the existing Event-B (de)composition techniques. By following this pattern, new features of a product line could be added to model a variant of the product line without the need to reprove the existing ones; provided the new features are designed in the same way and follow the same architectural style. We also show that the suggested pattern can overcome the ‘non-conformal’ refinement chains problem highlighted in the previous chapter, which means the number of refinement steps in each feature does not have to be same across all the features. We can then generalise our case-study experiments to suggest modelling patterns as guidelines for SPL modelling in Event-B for future users.
Chapter 6

Modelling ATM Case-study

6.1 Introduction

In this chapter, we present another case-study modelled in Event-B to further investigate our research findings of production cell (PC) case-study. This could also help in evaluating the application of existing standard Event-B composition and decomposition techniques for our feature-based reuse approach. We decided to model automated teller machine (ATM) because it provides significant variability as compared to PC. The ATM example is more reuse-oriented in terms of requirements features and suits the traditional feature modelling approach. We can model a product line of ATM systems having different features which can be configured to build variants of ATM. We explore further patterns for composition and to what extent we can reuse existing specifications and their associated proofs to build more products of a product line. This enables us to generalise our feature-oriented reuse framework in the form of a modelling pattern and its application on a different system to that of PC. Based on this case-study, we can then suggest modelling guidelines for future users of Event-B to model distributed systems while improving reusability of specification and their associated proofs. This case-study also generated requirements for future tools and techniques to further enhance reusability of Event-B developments.

6.2 ATM

An ATM provides various services to a bank’s customers using their ATM cards issued by the bank. There are some basic services provided by an ATM such as cash withdrawal, viewing account balance and card pin related services. Other services can also be provided by ATMs which vary for different banks and ATM locations, e.g., mobile top up and cash deposit etc. We can build a product line of ATMs to manage variability
and commonality and benefit from reuse while building several ATM products having different features. A set of available features configured and composed together result in a variant of an ATM product line. Figure 6.1 shows a feature model for the ATM product line and the requirements specification for the ATM case-study is given in Appendix B. Although some ATM requirements have been previously modelled in Event-B [123], we have used a different set of requirements and modelled these in a different way to experiment with our feature-oriented reuse approach.

6.3 Roadmap

In the following section, we show the Event-B development of some ATM features [5] and explore the amount of reuse that can be achieved by using existing (de)composition techniques of Event-B. We also investigate whether existing tools and techniques are capable enough for our proposed feature-oriented modelling in Event-B. Based on this, we can then suggest any requirements for the tools and techniques to be developed in the future to compliment our feature-based reuse framework. By generalising the modelling style used in this case-study, we can also provide a set of guidelines for Event-B users to model feature-oriented systems as expected in product line development.

At first, we modelled two features of ATM product line and by using the two existing decomposition techniques of Event-B, we show how we can avoid reproof efforts through reuse by following a pattern of modelling. Sections 6.4.1 and 6.4.2 discuss the modelling, refinement and decomposition of transfer and deposit features respectively. The composition of various sub-components of the two features (resulting from SED) is presented in Section 6.4.3 to model an ATM product. After modelling and refining these two features to build one ATM product, we modelled another ATM feature - withdraw - to build a

![Figure 6.1: ATM feature model](image-url)
second ATM product having three features, discussed in Section 6.4.4. We show how we can reuse existing features to build the second product and save user time and effort. Our suggested modelling pattern preserves refinement, which ensures that there are no inconsistencies and ambiguities introduced during the application of the pattern. In Section 6.4.5, we evaluate and generalise the suggested modelling pattern. This pattern also supports team-based development where different features of a product line could be modelled in parallel by different teams so long as the restrictions of the pattern are maintained.

6.4 Event-B Modelling of ATM Features

We started with an abstract model of ATM that models its requirements for a set of features. These include: withdraw, deposit, check balance, balance transfer, change PIN and validate card. So, in the abstract model, we have events for each of these features. Some of these features are mandatory while others are optional, as shown in the ATM feature model (Figure 6.1). Any instance of the ATM feature model must contain the mandatory features. For example, the card validation feature is required for any ATM product. So, we have modelled this as an independent reusable feature to be included in

Figure 6.2: Integral ATM abstract model for two features
any configuration of the ATM product line and this also helps in avoiding redundancy when modelling a subset of features from the feature model.

In order to save time and to use a smaller example, we only consider a subset of these features. So, we have an integral model of the ATM that allows cash deposit and balance transfer between two accounts. This abstract model is shown in Figure 6.2. In terms of user actions, we assume that an ATM card is validated in the card validation feature before any of these two features could be used by the user. We then decomposed this model into deposit and transfer features (Figure 6.3) using shared-variable decomposition (SVD). This acts as a problem decomposition step. This pattern remains valid even if we model and refine the deposit and transfer features separately, as far as the two features could later be shown as a result of shared-variable decomposition of a model. This would require a tool to generate external events in the developments of both features and to validate that the shared-variables are not refined.

As a result of decomposition, the event Deposit goes into the deposit feature and the Transfer event goes into the transfer feature - see Figure 6.4. Both features now have shared variables (i.e., bal, validCard and cardAccount) and external events (i.e., deposit feature has Transfer as an external event and the transfer feature has Deposit as an external event). All external events and shared-variables must not be refined as a consequence of SVD. Note that the external events shown in the figure are exact copies of their internal counterparts. This is because there are no local variables present for each of the feature and all the parameters are being used by the shared variables. Normally, external events are abstracted away leaving out details of their local variables which do not appear at this abstract level. Following is the detail for each of these features and their stepwise refinement, both horizontally and vertically.
Figure 6.4: ATM integral model decomposed using SVD into transfer and deposit features

### 6.4.1 Refinement of Transfer Feature

The first refinement model of transfer feature refines the Transfer event for a successful transfer of money (TransferOk) and another event (TransferFails) is introduced when the transfer fails due to the account balance being less than the transfer amount as shown in Figure 6.5. Other reasons for failure of balance transfer can be introduced in subsequent refinements as required (e.g., failing due to hardware or communication problems, though not considered in the scope of this case-study). The event FinishTransfer completes the transfer and resets the ATM involved in the transfer for a new transaction. As we can not refine the external event Despoit due to restriction of SVD, it is present at all refinement levels of transfer feature but not shown in the figures due to space limitations. Invariants show the typing of new variables introduced in this refinement, i.e., transferOkA, transferFailA and cardInAtm.

Figure 6.6 shows the events (both new and refined) of transfer feature during all the
refinements. The dotted lines show new events which refine _skip_ whereas the solid lines show refined events. The event _TransferFails_ of first refinement has similar structure to _TransferOk_ event in terms of event refinement and hence not elaborated in the figure.

The second refinement introduces request and response mechanism between ATM and the Bank. Here, ATM sends a balance transfer request to the bank (i.e., _ReqTransfer_ event), which responds after a successful or failed transfer event takes place (i.e., event _TransferOk_ or _TransferFails_ through event _RespTransferOk_ or _RespTransferFails_ respectively). The ATM then displays the transfer status (i.e., event _TransferOkAtm_ or _TransferFailsAtm_). The Event-B specification of all the events at this refinement level is shown in Figure 6.7. We also introduce a transaction variable _trans_ (typing: _trans ∈ P(ATM)_) for transferring balance which restricts the transfer request event only if the ATM is not currently in a transaction. An ATM currently in a transfer transaction could be in any of its sub states, as shown in the invariant below. Several new variables are also introduced at this refinement level.

\[
\text{inv18 : partition}(\text{trans, reqTransfer, transferOk, transferFail, respTransferOk, respTransferFail})
\]

The third level of refinement further refines the request and response mechanisms by decomposing the request event for sending and receiving the request and similarly for the response event. For example, the abstract event _ReqTransfer_ is decomposed into the refined event _SendReqTransfer_ (which sends the balance transfer request from the ATM) and the new event _RecvReqTransfer_ (which receives the balance transfer request at the

| INVARIANTS |
|----------------|----------------|
| inv9 : transferOK ∈ P(ATM) |
| inv10 : cardInAtm ∈ ATM → CARD |
| inv11 : transferOKA ∩ transferFailA = ∅ |

<table>
<thead>
<tr>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>TransferOK</td>
</tr>
<tr>
<td>Transfer</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>TransferFails</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>FinishTransfer</td>
</tr>
</tbody>
</table>

Figure 6.5: Transfer model’s invariants and events of first refinement
Figure 6.6: Event refinement of transfer feature

Table 6.1: Proof statistics for transfer refinements

<table>
<thead>
<tr>
<th>Model</th>
<th>Auto</th>
<th>Manual</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transfer_0</td>
<td>7</td>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td>Transfer_1</td>
<td>6</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>Transfer_2</td>
<td>53</td>
<td>5</td>
<td>58</td>
</tr>
<tr>
<td>Transfer_3</td>
<td>58</td>
<td>2</td>
<td>60</td>
</tr>
<tr>
<td>Transfer_4</td>
<td>115</td>
<td>8</td>
<td>124</td>
</tr>
</tbody>
</table>

Bank). Similarly, the response abstract event RespTransferOk is decomposed into the refined event SendRespTransferOk (which sends the response for a successful transfer from the Bank) and the new event RecevRespTransferOk (which receives the successful transfer response at the ATM) and so on for the transfer failure events. The abstract request and response variables reqTransfer, respTransferOk and respTransferFail are also refined as shown below:

\[ \text{inv7} : \text{partition}(\text{reqTransfer}, \text{sendReqTransfer}, \text{recvReqTransfer}) \]
\[ \text{inv8} : \text{partition}(\text{respTransferOk}, \text{sendRespTransferOk}, \text{recvRespTransferOk}) \]
\[ \text{inv9} : \text{partition}(\text{respTransferFail}, \text{sendRespTransferFail}, \text{recvRespTransferFail}) \]

The fourth refinement introduces middleware (MW) between the ATM and the Bank. This refinement also prepares the model to be decomposed using SED into three architectural components of the transfer feature, i.e., ATM, MW and the Bank, where MW is used for communicating between the two. The recomposition of these (ATM, MW, Bank) would refine the feature being decomposed (fourth refinement). Here we do not introduce any further requirements rather the focus is on how the model could be
designed towards implementation. Since we would like all features to have the same architectural design, this would enable us to later compose all components of a particular type (i.e., MW or Bank) resulting from the decomposition and refine these further.

In this refinement step, we added new variables to make sure that there are no more shared-variables when we decompose the model into three components and only have shared events. For example, in third refinement, we only had one variable for transfer amount \( \text{trAmount} \) whereas we introduced two extra variables \( \text{trAmountB}, \text{trAmountM} \) to hold transfer amount for each of the three components. This would be used to pass value between the components synchronised through shared events after SED as discussed below. Several new invariants were also added as a result of this. Some of the
invariants at this refinement level are shown below. These invariants allow us to make the state of three components disjoint by introducing new variables.

\[\text{inv12} : \forall a \cdot a \in \text{dom}(\text{trAmountM}) \Rightarrow a \in \text{dom}(\text{trAmount}) \land \text{trAmountM}(a) = \text{trAmount}(a)\]

\[\text{inv13} : \forall a \cdot a \in \text{recvReqTransfer} \land a \in \text{dom}(\text{trAmountB}) \Rightarrow a \in \text{dom}(\text{trAmountM}) \land \text{trAmountB}(a) = \text{trAmount}(a)\]

\[\text{inv14} : \forall a \cdot a \in \text{dom}(\text{srcAccM}) \Rightarrow a \in \text{dom}(\text{srcAcc}) \land \text{srcAccM}(a) = \text{srcAcc}(a)\]

\[\text{inv15} : \forall a \cdot a \in \text{recvReqTransfer} \land a \in \text{dom}(\text{srcAccB}) \Rightarrow a \in \text{dom}(\text{srcAccM}) \land \text{srcAccB}(a) = \text{srcAcc}(a)\]

\[\text{inv16} : \forall a \cdot a \in \text{dom}(\text{destAccM}) \Rightarrow a \in \text{dom}(\text{destAcc}) \land \text{destAccM}(a) = \text{destAcc}(a)\]

\[\text{inv17} : \forall a \cdot a \in \text{recvReqTransfer} \land a \in \text{dom}(\text{destAccB}) \Rightarrow a \in \text{dom}(\text{destAccM}) \land \text{destAccB}(a) = \text{destAcc}(a)\]

\[\text{inv18} : \forall a \cdot a \in \text{dom}(\text{cardInAtmM}) \Rightarrow a \in \text{dom}(\text{cardInAtm}) \land \text{cardInAtmM}(a) = \text{cardInAtm}(a)\]

\[\text{inv19} : \forall a \cdot a \in \text{recvReqTransfer} \land a \in \text{dom}(\text{cardInAtmB}) \Rightarrow a \in \text{dom}(\text{cardInAtm}) \land \text{cardInAtmB}(a) = \text{cardInAtm}(a)\]

\[\text{inv20} : \forall a \cdot a \in \text{dom}(\text{cardInAtmB}) \Rightarrow a \in \text{dom}(\text{cardInAtm})\]

Figure 6.8 shows the architecture of fourth refinement model including all the events and the component to which these would belong to after decomposition. The events shared between any two components are split into two. The sequence of events is shown by numbering the events in ascending order. Only one of the events from 3a or 3b would take place for a particular transaction, followed by their corresponding events only (a or b).
Chapter 6 Modelling ATM Case-study

The architectural decomposition is achieved using shared-event decomposition (SED). During the decomposition, the shared variable `validCard` is moved to the ATM component and the other shared variables `bal` and `cardAcct` into the Bank component. The external events were also partitioned accordingly, e.g., part of the external event `Deposit` containing variable `validCard` moved to ATM and the rest to the Bank component as shown in Figure 6.9. This splitting of external events is possible since we can leave out such external events (resulting from SVD) when composing models using SVC later. The only restriction of SVD is to not refine these events.

During architectural decomposition, we partitioned variables into different components whereas events were split between any two components. Hence, the components synchronise through these shared-events following the message passing mechanism of SED. For example, in Figure 6.8, the event `SendReqTransfer` is shared between ATM and the MW whereas the event `RecvReqTransfer` is shared between MW and the Bank. Figure 6.10 shows Event-B specification of `SendReqTransfer` event decomposed into two events for ATM and MW components. So, a balance transfer transaction starts when an ATM machine sends a transfer request (e.g., variable `trAmount` holds the transfer amount here) through the MW (e.g., `trAmountM` is passed the transfer amount) which is received by the Bank (e.g., `trAmountB` eventually contains the transfer amount). After processing the transfer request, the Bank then sends a response for a successful or failed transfer through the MW. The ATM finally displays the transfer status accordingly.

This decomposition was performed using the SED tool available as a plug-in for Rodin. We specify names of the components and the variables that each component should contain after the decomposition. This is why there must not be any shared variables among the components resulting from the decomposition. The tool then partitions the events based on the variables. Each of these three components can be further refined but the shared variables and external events (resulting from SVD) must not be refined. It is interesting to note that we have used SED on models that were decomposed using SVD having external events. Table 6.1 shows the proof obligations statistics for each of the transfer refinements and whether the POs were discharged automatically by the Rodin provers or interactively.

6.4.2 Refinement of Deposit Feature

Similar to the transfer feature, we refined and decomposed the deposit feature resulting in three components, i.e., ATM, MW and the Bank. In the first refinement, the abstract event `Deposit` was refined further to introduce the cash available in an ATM and the card inserted in it as shown below. This also increments the cash in the ATM along with incrementing the account balance with the deposited amount.
Event \textit{Deposit} \triangleq
\begin{verbatim}
// External event, DO NOT REFINE
extends Deposit
any
acc, am, c
\end{verbatim}
where
\begin{verbatim}
grd1 : acc ∈ ACCOUNT
grd2 : acc ∈ dom(bal)
grd3 : am ∈ \mathbb{N}_1
grd4 : c ∈ validCard
grd5 : c ↦→ acc ∈ cardAcct
\end{verbatim}
then
\begin{verbatim}
act1 : bal(acc) := bal(acc) + am
\end{verbatim}
end

Event \textit{Deposit} \triangleq
\begin{verbatim}
// ATM part
// External event, DO NOT REFINE
any
acc, am, c
\end{verbatim}
where
\begin{verbatim}
typing_c : c ∈ CARD
typing_am : am ∈ \mathbb{Z}
grd1 : acc ∈ ACCOUNT
grd2 : am ∈ \mathbb{N}_1
grd4 : c ∈ validCard
\end{verbatim}
then
\begin{verbatim}
skip
\end{verbatim}
end

Event \textit{Deposit} \triangleq
\begin{verbatim}
// BANK part
// External event, DO NOT REFINE
any
acc, am, c
\end{verbatim}
where
\begin{verbatim}
typing_c : c ∈ CARD
typing_am : am ∈ \mathbb{Z}
grd1 : acc ∈ ACCOUNT
grd2 : acc ∈ dom(bal)
grd3 : am ∈ \mathbb{N}_1
grd5 : c ↦→ acc ∈ cardAcct
\end{verbatim}
then
\begin{verbatim}
act1 : bal(acc) := bal(acc) + am
\end{verbatim}
end

In the second refinement, we introduced the request and response mechanism between the ATM and the Bank. Here, an ATM sends a deposit request to the bank (i.e., \textit{ReqDeposit} event), which responds (i.e., \textit{RespDeposit} event) after the deposit event takes place at the Bank (i.e., \textit{DepositB} event), where the account balance associated with the card is incremented by the deposited amount. Upon successful deposit, the
Event \( \text{SendReqTransfer} \) \[\refines\] \( \text{SendReqTransfer} \)

any \( \text{src\_ac, dest\_ac, am, a, c} \)

where

\begin{align*}
\text{grd1:} & \text{ src\_ac } \in \text{ACCOUNT} \\
\text{grd2:} & \text{ dest\_ac } \in \text{ACCOUNT} \\
\text{grd3:} & \text{ am } \in \mathbb{N}_1 \\
\text{grd4:} & \text{ src\_ac } \neq \text{ dest\_ac} \\
\text{grd5:} & \text{ c } \in \text{validCard} \\
\text{grd7:} & \text{ a } \mapsto c \in \text{cardInAtm} \\
\text{grd8:} & a \in \text{ATM} \\
\text{grd11:} & a \notin \text{trans}
\end{align*}

then

\begin{align*}
\text{act1:} & \text{sendReqTransfer} := \text{sendReqTransfer} \cup \{a\} \\
\text{act2:} & \text{trAmount}(a) := \text{am} \\
\text{act3:} & \text{srcAcc}(a) := \text{src\_ac} \\
\text{act4:} & \text{destAcc}(a) := \text{dest\_ac} \\
\text{act6:} & \text{trAmountM}(a) := \text{am} \\
\text{act7:} & \text{srcAccM}(a) := \text{src\_ac} \\
\text{act8:} & \text{destAccM}(a) := \text{dest\_ac} \\
\text{act10:} & \text{cardInAtmM}(a) := c \\
\text{act11:} & \text{trans} := \text{trans} \cup \{a\}
\end{align*}

end

Figure 6.10: An event of transfer feature’s fourth refinement model split into two events using SED

ATM displays deposit successful message and also increments the cash available in the ATM (i.e., \( \text{DepositATM} \) event). We also introduced transaction mechanism through the variable \( \text{trans} \) as we did in the transfer feature’s refinement. Figure 6.11 shows the events at this refinement level.

Again, following from the transfer feature refinement, the third level of refinement further refined the request and response mechanisms by partitioning the request event for sending and receiving the request and so on for the response event. Similarly, the fourth refinement introduced middleware between ATM and the Bank. The proof obligations statistics for this development is given in Table 6.2 and Figure 6.12 shows the events (both new and refined) of deposit feature during all the refinements.

We then decomposed this deposit feature into ATM, MW and the Bank using SED. The shared variables and external events were partitioned in the same way as done in the
Event $\text{ReqDeposit} \equiv$
\begin{align*}
\text{any} & \quad c, am, a \\
\text{where} & \\
\text{grd2} : & \quad am \in N_1 \\
\text{grd3} : & \quad c \in \text{ran(cardInAtm)} \\
\text{grd4} : & \quad a \in \text{dom(cardInAtm)} \\
\text{grd5} : & \quad \text{cardInAtm}(a) = c \\
\text{grd6} : & \quad c \in \text{validCard} \\
\text{grd9} : & \quad a \notin \text{trans} \\
\text{then} & \\
\text{act1} : & \quad \text{reqDep} := \text{reqDep} \cup \{a\} \\
\text{act2} : & \quad \text{trans} := \text{trans} \cup \{a\} \\
\text{act3} : & \quad \text{atmDepAm}(a) := am \\
\text{end} \\
\end{align*}

Event $\text{RespDeposit} \equiv$
\begin{align*}
\text{any} & \quad a \\
\text{where} & \\
\text{grd1} : & \quad a \in \text{processedDep} \\
\text{grd2} : & \quad a \in \text{reqDep} \\
\text{then} & \\
\text{act1} : & \quad \text{respDep} := \text{respDep} \cup \{a\} \\
\text{act3} : & \quad \text{processedDep} := \text{processedDep} \setminus \{a\} \\
\text{act4} : & \quad \text{reqDep} := \text{reqDep} \setminus \{a\} \\
\text{end} \\
\end{align*}

Event $\text{DepositATM} \equiv$
\begin{align*}
\text{any} & \quad am, a \\
\text{where} & \\
\text{grd8} : & \quad a \in \text{respDep} \\
\text{grd9} : & \quad a \in \text{dom(atmDepAm)} \\
\text{grd10} : & \quad \text{atmDepAm}(a) = am \\
\text{then} & \\
\text{act2} : & \quad \text{atmCashA}(a) := \text{atmCashA}(a) + am \\
\text{act4} : & \quad \text{respDep} := \text{respDep} \setminus \{a\} \\
\text{act5} : & \quad \text{trans} := \text{trans} \setminus \{a\} \\
\text{end} \\
\end{align*}

Figure 6.11: Deposit feature’s events of second refinement

Table 6.2: Proof statistics for deposit refinements

<table>
<thead>
<tr>
<th>Model</th>
<th>Auto</th>
<th>Manual</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Deposit_0</td>
<td>7</td>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td>Deposit_1</td>
<td>7</td>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td>Deposit_2</td>
<td>27</td>
<td>0</td>
<td>27</td>
</tr>
<tr>
<td>Deposit_3</td>
<td>37</td>
<td>0</td>
<td>37</td>
</tr>
<tr>
<td>Deposit_4</td>
<td>46</td>
<td>3</td>
<td>49</td>
</tr>
</tbody>
</table>

transfer feature. So, the shared variable $\text{validCard}$ moved to the ATM component and the rest of the shared variables (i.e., $\text{bal}$ and $\text{cardAcct}$) moved to the Bank component. The external events were also partitioned accordingly. The vertical refinement architecture of deposit feature is same as that of the balance transfer feature discussed earlier.

Figure 6.13 shows the architecture of fourth refinement model including all the events and which component these would belong to after decomposition. It also shows the sequence of events. So, for the deposit feature, an ATM sends a deposit request through the MW which is received by the bank. The bank then sends a response after incrementing the account balance through the middleware. The ATM finally displays the deposit successful message and also increments the amount of cash it contains by the deposited
amount. These decomposed components synchronise using the shared-events and can be further refined independently.

### 6.4.3 Composing Sub-components of Two Features

Figure 6.14 shows the development and (de)composition structure for the deposit and transfer features of the ATM. In the figure, asterisk (*) denotes a model with external events, and bal, validCard and cardAcct are the model’s shared variables. We needed external events in order to later compose these features using SVC after several refinement steps, to make sure the composition was correct by construction and hence
not need reproving. This means that we had to make sure that the two features were a result of SVD of an integral abstract model, as mentioned earlier. This generated a tooling requirement to automatically generate external events for the features that we would like to compose using SVC and hence saving time to manually do so. Note that in this case study, the shared variables \texttt{bal} and \texttt{cardAcct} along with their corresponding part of the external event are localised in the Bank component; whereas the shared variable \texttt{validCard} and its corresponding part of the external event is localised in the ATM component.

Now that we have the same architectural decomposition (ATM, MW, Bank) for both of the features, we would like to compose these models pairwise (i.e., Bank\_T+D = Bank\_T + Bank\_D, etc.) for implementation purposes. In general, the task would of course be more complex, involving more than two features. In our case, where the shared variables \texttt{bal} and \texttt{cardAcct} are localised into the two architectural Bank components, these can be composed, with the composite Bank refining each Bank component. This is because each Bank’s external events are ‘cancelled out’, or implemented, by the other Bank’s
actual events. For example, components Bank0\_T and ATM0\_T of transfer feature have parts of the external event $\text{Deposit}$ as shown below:

\[
\text{Event } \text{Deposit} \triangleq \\
// \text{ ATM0\_T} \\
// \text{ External event, DO NOT REFINE} \\
\text{any} \\
\text{acc, am, c} \\
\text{where} \\
\text{typing}_c : c \in \text{CARD} \\
\text{typing}_am : am \in \mathbb{Z} \\
\text{grd1} : \text{acc} \in \text{ACCOUNT} \\
\text{grd3} : am \in \mathbb{N}_1 \\
\text{grd4} : c \in \text{validCard} \\
\text{then} \\
\text{skip} \\
\text{end}
\]

These external events have been refined by events $\text{sendReqDeposit}$ and $\text{depositB}$ respectively, present in ATM0\_D and Bank0\_D of deposit feature as shown below:

\[
\text{Event } \text{sendReqDeposit} \triangleq \\
\text{any} \\
\text{c, am, a} \\
\text{where} \\
\text{typing}_c : c \in \text{CARD} \\
\text{typing}_am : am \in \mathbb{Z} \\
\text{grd2} : a \in \text{ATM} \\
\text{grd3} : am \in \mathbb{N}_1 \\
\text{grd4} : c \in \text{ran(cardInAtm)} \\
\text{grd5} : a \in \text{dom(cardInAtm)} \\
\text{grd6} : c \in \text{validCard} \\
\text{grd9} : a \notin \text{trans} \\
\text{then} \\
\text{act2} : \text{trans} := \text{trans} \cup \{a\} \\
\text{act3} : \text{atmDepAm}(a) := am \\
\text{end}
\]

\[
\text{Event } \text{depositB} \triangleq \\
\text{any} \\
\text{acc, am, a, c} \\
\text{where} \\
\text{typing}_c : c \in \text{CARD} \\
\text{typing}_am : am \in \mathbb{Z} \\
\text{grd1} : \text{acc} \in \text{ACCOUNT} \\
\text{grd2} : \text{acc} \in \text{dom(bal)} \\
\text{grd3} : am \in \mathbb{N}_1 \\
\text{grd5} : c \mapsto \text{acc} \in \text{cardAcct} \\
\text{grd6} : a \in \text{ATM} \\
\text{grd7} : a \in \text{dom(atmCash)} \\
\text{grd8} : a \in \text{dom(cardInAtmB)} \\
\text{grd9} : \text{cardInAtmB}(a) = c \\
\text{grd11} : a \in \text{dom(atmDepAmB)} \\
\text{grd12} : \text{atmDepAmB}(a) = am \\
\text{grd13} : a \in \text{recvReqDep} \\
\text{then} \\
\text{act1} : \text{bal}(acc) := \text{bal}(acc) + am \\
\text{act2} : \text{atmCash}(a) := \text{atmCash}(a) + am \\
\text{act5} : \text{processedDep} := \text{processedDep} \cup \{a\} \\
\text{end}
\]

Figure 6.15 shows the events of Bank\_T and Bank\_D and the resulting composition Bank\_T+D, where external events disappear as a consequence of shared-variable composition. The composition of different components is achieved using our feature composition tool, as there is no tool support for shared-variable composition. Since we had same architecture for both the components, there were conflicting variables and invariants which were resolved by the tool automatically. Table 6.3 shows the number of proof
Chapter 6 Modelling ATM Case-study

125

Figure 6.15: Composing bank components of transfer and deposit features

Table 6.3: Proof statistics for deposit and transfer components and their composition

<table>
<thead>
<tr>
<th>Component</th>
<th>Transfer T</th>
<th>Deposit D</th>
<th>Composite T+D</th>
</tr>
</thead>
<tbody>
<tr>
<td>ATM</td>
<td>11</td>
<td>10</td>
<td>20</td>
</tr>
<tr>
<td>MW</td>
<td>17</td>
<td>7</td>
<td>25</td>
</tr>
<tr>
<td>Bank</td>
<td>32</td>
<td>18</td>
<td>47</td>
</tr>
<tr>
<td>Total</td>
<td>60</td>
<td>35</td>
<td>92</td>
</tr>
</tbody>
</table>

obligations for different components of transfer and deposit features and their compositions. This shows that by following the suggested modelling pattern, we can avoid the proof effort for the composite models (i.e., fourth column of the table).

The suggested pattern is correct by construction due to the application of shared-variable and shared-event (de)composition techniques and following their restrictions. In order to prove that the pattern works for this example, we composed the three components (i.e., ATM_T+D, MW_T+D, Bank_T+D) using shared-event composition, though this is not required in practice. The composition resulted in the concrete integral model of ATM (box C in Figure 6.14) which refines the abstract integral model (box A in the figure). This refinement relationship was also proved by refining the integral ATM abstract model up to four refinements and the composition (i.e., ATM concrete model) was proved to refine the fourth refinement of integral ATM model, as shown in Figure 6.16. In order to prove that box C refines box A, we had to perform refinements of box B although this is not required in actual development.
6.4.4 Modelling Second ATM Product By Reusing Existing Features

After modelling an ATM product with two features, we modelled another ATM product having a cash withdraw feature as well (as shown by the dotted box on the right in Figure 6.17). This enabled us to explore how much of the existing models and their proofs could be reused or to what extent we could reduce the overall modelling effort for the second ATM product through reuse after modelling the first one. So, we elaborated the top-level integral model to include the cash withdrawal functionality and decomposed it into three features (i.e., deposit, transfer and withdraw). Figure 6.18 shows the abstract integral Event-B model for the second ATM product.
6.4.4.1 Refinement of Withdraw Feature

We refined the withdraw feature in the same way as the other two features discussed earlier. The first refinement differentiates between the successful and failed withdrawal. The next two refinements introduce the request and response mechanism between Bank and the ATM. The fourth refinement introduces MW so that we can decompose this model into three components (i.e., ATM, MW and Bank), as we did for the other two features. Figure 6.19 shows the events (both new and refined) of withdraw feature during all the refinements. Note that the number of refinement steps for any of the features could be different. This solves the problem of composing non-conformal refinements as discussed in the production cell case-study.

6.4.4.2 Discussion

Provided the new feature is ‘non-interfering’ - in the sense that in the SVD refinement, the other two features remain unchanged - then all we have to do is refine the withdraw feature only. An interfering feature would require making changes to the existing features which would lead to feature interaction problem mentioned in Chapter 3. If the additional feature only interferes with a subset of existing features, then we would only need to re-engineer that subset. This could also be very useful in saving modelling and proof effort when we can reuse a large set of existing features by modifying a small subset of these due to feature interaction of additional feature to model a new product.
MACHINE IntegralATM2_0
SEES IntegralATM2_CO
VARIABLES bal, cardAcct, validCard
INVARIANTS
  inv1: bal ∈ ACCOUNT → N
  inv2: cardAcct ∈ CARD → ACCOUNT
  inv3: validCard ⊆ CARD
EVENTS

Event Deposit $\triangleq$
  any
  acc, am, c
where
  grd1: acc ∈ ACCOUNT
  grd2: acc ∈ dom(bal)
  grd3: am ∈ N
  grd4: c ∈ validCard
  grd5: c ↦→ acc ∈ cardAcct
then
  act1: bal(acc) := bal(acc) + am
end

Event Withdraw $\triangleq$
  any
  acc, am, c
where
  grd1: acc ∈ ACCOUNT
  grd2: am ∈ N
  grd3: acc ∈ dom(bal)
  grd4: am ≤ bal(acc)
  grd5: c ∈ validCard
  grd6: c ↦→ acc ∈ cardAcct
then
  act1: bal(acc) := bal(acc) − am
end

Figure 6.18: Integral ATM abstract model for 2nd ATM product of Figure 6.17

Figure 6.19: Event refinement of withdraw feature
Table 6.4: Proof obligations for withdraw component and the composite models

<table>
<thead>
<tr>
<th>Component</th>
<th>Withdraw</th>
<th>Composite</th>
</tr>
</thead>
<tbody>
<tr>
<td>ATM</td>
<td>13</td>
<td>29</td>
</tr>
<tr>
<td>MW</td>
<td>9</td>
<td>33</td>
</tr>
<tr>
<td>Bank</td>
<td>28</td>
<td>61</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>50</strong></td>
<td><strong>123</strong></td>
</tr>
</tbody>
</table>

The additional feature must also preserve any invariants related to the shared-variables in the existing features.

For the second ATM product, the existing deposit and transfer features would now contain external event Withdraw of the withdraw feature. Since the deposit and transfer components have already been proved, new POs were only generated for the newly added external event acting on the shared variables (i.e., bal, cardAcct and validCard). Also, these new POs were only generated in the abstract models of the deposit and transfer features, no matter how many refinements exist because of the restriction of SVD that external events and shared variables should not be refined. Hence, we only have to discharge these small number of POs when reusing existing models. In this particular example, only three new POs were generated for each of the deposit and transfer abstract models and these were automatically discharged by the Rodin provers. Table 6.4 shows the number of POs for different components of withdraw feature before and after composition with the components of other two features. This shows that we managed to avoid reproving POs for whole developments of deposit and transfer features by following this pattern of reuse (Tables 6.1 and 6.2 show total savings of 375 POs for second ATM product). With the help of proper tool support, we could even avoid discharging POs of abstract models for the components being reused.

### 6.4.5 Evaluation

We have examined a specific pattern of mixed decomposition-recomposition - SVD followed by SED and then SVC/SEC in a single development. It is possible to do this provided shared variables and their associated external events are not refined and the shared variables are localised in exactly the same component resulting from architectural decomposition of various features. Our case-study example supports this claim where we proved that this reuse pattern preserves the refinement relation provided the restrictions of the pattern stated above are followed.

We can generalise this pattern where the shared variables and their associated external events must be localised in exactly the same component in each of the feature developments. For example, consider two features P and Q with shared-variables x and y and
external event(s); which appear to result from SVD of model $M$ as shown in Figure 6.20. After several refinement steps of $P$, we get $Pn$ where new events and variables (e.g., variable $w$) are introduced. This model is further decomposed using SED into three components $C1...C3$ where variables $w$ and $x$ are localised into $C1$ and $y$ into $C3$. This pattern requires that the variables $x$ and $y$ must be localised into components $C1$ and $C3$ respectively for the feature $Qm$ (which results from various refinements of $Q$). This could then scale up to any number of features (as required in product line modelling) without the need for reproving already proved features, and their composition - as suggested earlier - would be correct by construction. These components could be further refined, provided that the restrictions of the two styles of decomposition are observed.

As shown in the figure, we have to use SVC while composing $PnC1(w, x)$ and $QmC1(x)$ as these two components have external events and shared-variable $x$, and same is the case when composing $PnC3(y)$ and $QmC3(y, z)$. Here, we can leave out the external events during SVC since these are cancelled-out by their counter-parts, e.g., external event of $PnC1$ is cancelled out by an event of $QmC1$. We can compose $PnC2$ and
Chapter 6 Modelling ATM Case-study

QmC2 using SEC since both components are disjoint, and so on for the rest of the components.

6.5 Conclusion

We have modelled and refined features of ATM product line using Event-B with a view to reusing these features when modelling a second ATM product after the first one. We explored the use of both types of decomposition/composition techniques of Event-B in a single development (i.e., shared-event (SED/SEC) and shared variable (SVD/SVC) (de)composition). We used SVD for problem decomposition earlier in the development where we decomposed the problem into various requirements features. These features, after refining separately, were decomposed again into architectural components using SED. This served as a solution decomposition. These architectural components could then be refined independently as required and could be composed using SVC/SEC for implementation purposes.

This resulted in a modelling pattern which preserves refinement, provided that the restrictions of both styles of decomposition and that of the suggested pattern are observed. These include that the shared-variables and their associated external events are not refined as a consequence of SVD; and are localised in exactly the same type of architectural component during solution decomposition of all the features. It has also been shown that the external events could be decomposed when preparing the model for SED. This does not violate the restriction of SVD on external events. Although the pattern is correct by construction due to the application of the two styles of (de)composition (i.e., SVD/SED), this refinement preservation was also verified by manually refining the integral development and proving that the concrete model resulting from this decomposition/recomposition pattern refines the abstract model, which was decomposed using SVD in the beginning.

The first ATM product modelled contained two features, i.e., deposit and transfer. We then modelled another ATM product which contained withdraw feature as well. The second product reused already developed features and hence saved modelling and proof effort. The amount of proof obligations to be discharged for existing features was minimal which could be completely avoided with the help of proper tool support. Since this pattern can be scaled for modelling a product having large number of features, this could significantly improve productivity and reduce modelling time and effort. So, this case-study has proved quite useful in terms of suggesting a modelling pattern which employs existing techniques of Event-B. This pattern helps in reusing Event-B developments and their associated proofs as required in product line modelling. Based on our two case-studies, we suggest guidelines for Event-B users in the next chapter, to use
feature-oriented reuse approach in Event-B for modelling product lines. We also highlight the requirements for future tool support that needs to be implemented to make full use of the suggested feature-oriented reuse approach.
Chapter 7

Guidelines for Feature Modelling in Event-B & Tooling

Requirements

7.1 Introduction

In this chapter, we present a set of guidelines for feature-oriented modelling in Event-B for future modellers who would like to model software product lines (SPLs) using Event-B, to gain benefits of reusing formal models and their associated proofs. These guidelines are based on our case-study experiments with existing Event-B techniques, our proposed feature modelling framework and the developed tooling. We also discuss future tooling requirements that have not been implemented as part of our work due to the time limitations. By developing the suggested tools, our feature-modelling framework would become more efficient and productive for modellers as this involves automating the tasks that have been carried out manually during this research.

7.2 Guidelines for Feature Modelling in Event-B

Our case-study experiments reveal some patterns of modelling that could be generalised and presented as guidelines for specifying SPLs in Event-B. These patterns make use of Event-B’s existing tools and techniques and also point out the shortfalls in terms of tooling. By developing the tool support, discussed later, this whole process could be automated, saving lot of user time and effort. At the moment, the lack of proper tool support means that we have to do several tasks manually, e.g., trivial refactoring and instantiation of a generic model into two disjoint models.
The decision to use top-down usual modelling approach of Event-B, bottom-up approach of SPL engineering or the middle-in approach of the ATM example would vary for different domains. The domain expert or system modeller would be in a better position to judge which approach would suit more for a particular system. We have used both top-down and bottom-up approaches in our PC case-study and the middle-in approach in our ATM case-study.

Following are the two styles of modelling a system in Event-B that generalise the approaches we used in our case-study examples to improve reusability of specifications as needed for modelling SPLs.

### 7.2.1 Modelling a System as a Set of Generic Features

In order to apply this style of modelling, the modellers need to explore if there are any common requirements which could be specified in a way that these could be used to model other, more complex, requirements. This means modelling the requirements that exhibit common functionality of various features of the system as generic features in Event-B. This style of modelling would be useful if the system being modelled has a set of requirements that can be generalised and decomposed, modelled as a set of generic features, specialised and composed later on, to model various features needed to model a complete system. An application of this modelling pattern can be seen in our controller-based modelling of production cell (Section 5.5).

Figure 7.1 shows architecture of this modelling pattern. We can categorise this process into three phases as shown in the figure, discussed below.

#### 7.2.1.1 Requirements Manipulation

In the first phase, we start by generalising requirements and functionally decomposing these requirements into various features of the requirement specification. For example, instead of grouping all the requirements of a robot controller in a single feature, we may decompose its requirements into sub-controllers. This would include requirements features for movement, rotation and gripper (for pick and drop functionality). By generalisation, we mean to leave out some of the requirements that are specific for robot controller since we would like to keep these sub-controllers as generic as possible to be reused in various settings and not just for specifying the robot controller. This also gives us the robot specific requirements needed during specialisation and composition in the third phase. So, we could specify requirements for a robot controller by composing these three requirements features. If we need to specify requirements for another controller, e.g., a crane, we could reuse requirements for movement and gripper feature along with some additional crane specific requirements.
In the second phase, each of these requirement features are then modelled in Event-B and refined independently, resulting in a repository of reusable generic Event-B models having various refinement levels (e.g., features $F_1, F_2, ..., F_N$ in the figure). The emphasis on refinement is due to the fact that composing refinement chains in the next phase is not trivial and becomes even more complex when composing refinement chains having different number of refinement steps. By generic features, we mean to use special keywords or placeholders that must be filled during the specialisation of these features. A possible approach is to specify these generic placeholders as a special construct in Event-B, i.e., ‘$X...X$’, and the tool should detect and force/guide the user to fill these placeholders. At the moment, there is no such facility available in the Rodin tool which would require enhancements in the tool support for refactoring, as discussed in Section 7.3.1 later.
7.2.1.3 Instantiation and Composition

In the third phase, we instantiate, specialise and compose these generic refinement chains to build the desired product. For example, as the figure shows, we may instantiate feature $F_1$ to model features $P_a$ and $P_b$ and feature $F_2$ to model feature $P_c$. These three product features (i.e., $P_a, P_b$ and $P_c$) could be composed to model a product $P_1$. The composition information could be usually available through the first phase of this process when we generalise and decompose the requirements specification. We could model another product, e.g., $P_2$, by reusing a different set of generic features through instantiation and composition.

Here, we are only interested in the abstract and concrete models of the final composed product (i.e., $P_1, P_2$), which provide us the abstract specification of a controller and its implementation respectively. We could only apply our suggested ‘feature composition’ technique here, since both SVC and SEC could not be applied due to their restrictions (i.e., models with shared variables must have external events or models must not have shared variables, respectively). This means that the refinement relationship between the abstract and concrete models of a composed product (e.g., $P_1$) needs to be reproved since we add compositional information that may generate additional POs to be discharged; and the refinement preservation of the composition is not guaranteed as shown by the ‘?’ in the figure. One possible solution to deal with this proofs issue is to provide a proof reuse mechanism as discussed in the future work chapter. Although this proof effort could be saved by using the second modelling style suggested below, this may not be applicable in all situations as is the case with the PC case-study.

7.2.2 Modelling a System as a Set of Separable Features

This style of modelling is more suitable for feature-oriented modelling approach commonly practiced within SPL community. This is particularly applicable to product lines which can be represented as feature models having well defined commonality and variability expressed in terms of features. This means there will be mandatory features present in all products of a product line and optional features could be specified as and when required. This style allows a modeller to reuse existing features with minimum (and possibly without any) proof effort when modelling variants of a product line. Figure 7.2 shows an arbitrary feature model of a product line ‘SystemX’ with mandatory feature $F_1$ and other optional features. We will use this example feature model below to build two variants of the product line through reuse by following this modelling style. We can divide this modelling style into three steps, i.e., problem decomposition, architectural decomposition and architectural composition, discussed below.
7.2.2.1 Problem Decomposition

In this pattern of modelling, we start with an abstract model of the system being modelled where we introduce the requirements so that this integral model can then be functionally decomposed into a number of features using SVD style. This means the features would contain shared variable(s) and external event(s). It is also possible to start with abstract models of the features for a product line specified in Event-B and represented as an SVD of an integral model. In principle, this would require composing these features and then decomposing using SVD so that each feature contains external events. An external event in a feature mimics the behaviour of an internal event present in another feature acting on the shared variables. The aim is to ensure that we could use SVC later on, which is only possible if we apply SVD earlier on. This process could be automated with proper tool support. From SPL point of view, we could start with the base model of the system which only contains the mandatory features. Optional features can then be added to model further products of a product line. This pattern also supports team-based development where different teams can work on additional features independently once the architecture for the base model of a product line is agreed.

Figure 7.3 shows architecture of this modelling pattern where an integral model of a product $P$ with events $e1$ and $e2$ (acting on shared variables $x$ and $y$) is decomposed into two features $F1$ and $F2$. This is based on the configuration of our feature model shown in Figure 7.4, where we have selected features $F1$ and $F2$ for modelling the product. Event $e2$ becomes external event in feature $F1$ and event $e1$ becomes external in feature $F2$. We can then refine these features, in a number of steps, by ensuring that the shared-variables ($x, y$) and external events (marked by * in the figure) are not refined as a constraint of SVD. During the refinement, new variables (i.e., $w$ and $z$ in features $F1$ and $F2$ respectively) and events (i.e., $e11, e12$ in $F1$ and $e21, e22, e23$ in $F2$) can be introduced as usual. The number of refinement steps for each of the features does not have to be the same.
Figure 7.3: Modelling a product line as a set of separable features

Figure 7.4: Product 1 configurations of our example feature model of Figure 7.2
7.2.2.2 Architectural Decomposition

After refining the features independently, these can then be architecturally decomposed using SED technique into various components (i.e., feature $F_1 n$ decomposed into $F_{1n_A}$, $F_{1n_B}$, $F_{1n_C}$ and so on for feature $F_2$). The decomposition must be done in a way that the shared variables and the corresponding external events are localised in the same type of components across different features of $P$ (e.g., component ‘B’ and ‘C’ of $F_1$ and $F_2$ in the figure). This restriction is in place to make sure that the recomposition of these decomposed components can be achieved without reproof as a result of SVC.

Note that we might also need to decompose external events into various components during this architectural decomposition. This is due to variable splitting during SED. For example, part of external event $e_2$ in $F_1 n$ using shared variable $x$ (i.e., $e_2 a$) moves to component $F_{1n_B}$, whereas the rest of the event acting on the shared variable $y$ (i.e., $e_2 b$) moves to the component $F_{1n_C}$. This is possible to decompose external events as the restriction is avoid refining these because we will be leaving out any external events when recomposing these components, as explained in the next step. This splitting of external events might require us to simplify some guard predicates in preparation for SED. For example, if we have a guard predicate in event $e_2$ which contains both $x$ and $y$, this must be simplified into two predicates before we split this event; otherwise the decomposition would not be possible and will be highlighted as a problem by the tool. This is a usual practice when applying SED and is performed in a preparatory refinement step before the decomposition. These architecturally decomposed components can then be further refined as required (e.g., components $F_{1n_A}$ and $F_{2n_A}$ as shown in the figure).

7.2.2.3 Architectural Composition

In the final step, we can compose these components using SVC/SEC and the resulting concrete model of product $P$ will be a refinement of the abstract model of $P$. This composition is possible only if we have the same architectural design among all the components being composed. This will be correct by construction due to the application of the two (de)composition styles. The components having shared variables are composed using SVC and the disjoint ones are composed using SEC. For example, we compose components $F_{1n_C}$ and $F_{2n_C}$ using SVC as these have a shared variable $y$. The external events $e_{1b}$ and $e_{2b}$ are thrown away because of their internal counterparts (or their refinements) being present in the components being composed, i.e., event $e_{1b}(y)$ is refined by event $e_{13}(y)$ and $e_{2b}(y)$ is refined by event $e_{23}(y)$. This is how the recomposition is suggested by the SVD technique [25]. Similarly, we compose components ‘B’ of $F_1 n$ and $F_2 n$ using SVC by leaving out the external events $e_{1a}(x)$ (refined by $e_{12}(w,x)$) and $e_{2a}$ (refined by $e_{22}(x)$). The components $F_{1n_A}$ and $F_{2n_A}$, after any refinements, are composed using SEC as these do not have any shared state. Note that we will only
need to compose $A$, $B$ and $C$ components of each of the features ($F_1, F_2, ..., F_n$) having common architecture, which can again be refined further and translated to executable code.

### 7.2.2.4 Modelling Second Product Through Reuse

After modelling one product of $P$, we can model another product having an additional feature (i.e., $F_3$, as shown in Figure 7.5) by reusing already modelled and proved features (i.e., $F_1$ and $F_2$). Figure 7.6 shows the configuration of our example feature model for the second product. The new feature to be added must be non-interfering and should not lead to the feature interaction problem, i.e., the additional feature must not require the existing features to be re-engineered. If this new feature introduces more shared variables then that may require us to modify events of the existing features and hence reproofing those at various refinement levels. It is also possible that the new feature only interacts/interferes with a subset of existing features, i.e., by introducing variables that are only shared with some of the existing features. In this case, only the affected subset of existing features would require re-engineering. Note that this may generate ripple affect where modifying this subset of features may affect rest of the existing features.

For modelling the second product, provided that this new feature is non-interfering, we will only need to make sure that $F_3$ is a result of SVD of $P$ and will need to include the external events of $F_3$ in $F_1$ and $F_2$ and vice-versa as shown in the Figure 7.5 (this process can also be automated). This may generate additional POs for abstract models of $F_1$ and $F_2$, which should be easily discharged by the tool. There will be no effect of this additional feature on the refinements of $F_1$ and $F_2$ (i.e., $F_1, ..., F_{1n}$ and so on for $F_2$). This is because the external event of $F_3$ in existing features must not be refined and hence there will be no further POs to be discharged; irrespective of the number of refinement steps present in the existing features. This additional feature $F_3$ must also be designed in the same way as $F_1$ and $F_2$. Following the same rules of architectural decomposition used for existing features, $F_3$ must be decomposed in three components (i.e., $A, B$ and $C$). The shared variable $x$ along with its associated external events (i.e., $e_1a$ and $e_2a$) moves to the component $B$ whereas the shared variable $y$ along with its associated external events (i.e., $e_1b$ and $e_2b$) moves to the component $C$. We then compose components $B$ and $C$ with existing composites $F_1nF_2n_B$ and $F_1nF_2n_C$ respectively, resulting in $F_1nF_2nF_3n_B$ and $F_1nF_2nF_3n_C$. The same type of composition would apply here, i.e., SVC for components $B$ and $C$ and SEC for components $A$. So, we get another product of $P$ by modelling and proving additional component and reusing previously modelled features.

This modelling style allows reusing existing Event-B models as well as their proofs. In terms of proof effort, this modelling style is more useful as compared to the one discussed earlier since this style provides proof reuse as well. We have shown the amount of proof
Figure 7.5: Modelling a product line variant by reusing existing features

Figure 7.6: Product 2 configuration of our example feature model of Figure 7.2
reuse provided by this style in our ATM case study in previous chapter. If this pattern is not followed and features are modelled as generic reusable models independently and later on composed using additional composition time information (discussed as Feature Composition earlier), then the user needs to reprove the composite model. This could only be helpful if we have a mechanism for proof reuse that would reduce the reproof effort, as discussed in future work Section 8.3.

7.3 Future Tool Support Requirements

In order to make use of the feature modelling approach suggested, the following tool support is needed to facilitate the modeller and improve productivity.

7.3.1 Refactoring Support

During the composition of features, allow the user to add new variables, invariants and guards. These additional variables, guards and invariants should be static-checked for errors and instantly reported to the user. The user should be able to guide the refactoring, i.e., when refactoring a machine, allow the user to suggest prefixes. This should also support instantiation of generic refinement chains by refactoring. The refactored elements should also be static-checked for any errors. The tool should provide support for generic placeholders. For example, an element name (i.e., variable, event etc.) must be provided by the user at the time of composition or instantiation when a model contains any placeholder (e.g., ‘X...X’). This will allow a modeller to use generic names in the models which can be specialised with meaningful names and to model duplicate functionality in case of multiple instantiation of the same model. For example, as discussed in Chapter 5, we modelled the ‘press’ component of PC, which could be specialised twice when we need a variant of PC having two presses. The refactoring plug-in [127] available in Rodin was quite helpful during our case-study modelling but we also need refactoring support during the instantiation and composition process. It does not provide a systematic way of refactoring different elements of the model as highlighted in previous chapters. It only allows renaming one element of a machine or a context at a time, which is cumbersome when instantiating large models.

7.3.2 Feature Modelling and Topology Tool

The tool should allow the user to specify cardinality for feature instantiation (cloning), so that a feature could be replicated a number of times in a particular instance. For example, in PC feature model, the feature ‘press’ could have a cardinality ‘1..n’, which means any instance of the PC could have at least one and up to ‘n’ presses. This must be
then supported in the configuration editor and would also need user guided refactoring support as discussed above.

The users should be able to annotate the feature model, i.e., to specify composition rules, more complex constraints and other information that they think might be helpful during the configuration process. For example, see the composition rule and rationale provided in the example feature model of Figure 3.7. This would help the modeller during the selection of a particular feature while instantiating a product line, e.g., if a user has the option of selecting one of the two optional features then this could guide the user in selection the feature more suitable in a particular situation. Another example is that of an invariant which must be added to the composite model when a particular set of features are selected during configuration of the feature model.

After drawing a feature model, allow the user to draw a topology graph based on that feature model to visualise the feature model instance and how different features will be connected to each other in the graph, as discussed in PC. The features could be drawn as nodes and the user should be able to select the transitions between these nodes by selecting events from Event-B model of the available features. These events will act as communicating events for passing values between the component. This may also be achieved by adding extra constraints to the feature model that specifies connection between components. So, the user specifies which component is connected to which other components and this information could be used to draw transitions between different nodes. At the moment, the feature modelling tool only supports two constraints, i.e., *includes* and *excludes*.

### 7.3.3 SVC Tool

There should be a tool for composing models using the SVC style. This may also include the support for automatically generating external events in the models to be composed using SVC and later on removing these external events during the composition. Also, it would be helpful for a modeller to have a single tool that could be used to compose models using any of the composition styles, i.e., SEC, SVC, fusion and feature composition.

### 7.3.4 Composition Replay Support

All the composition decisions must be saved so that the user can come back later to modify or redo the composition without losing additions or refactoring etc. It would be useful to have a script of all the configuration decisions so that the user can replay the configuration by editing the script (may be doing some refactoring etc.). This will be like an audit trail of all the actions performed during the configuration. This may also include graphical representation of the actions performed during the composition to visualise more complex multi-step compositions.
7.4 Conclusion

We have presented two patterns of modelling, that emerged as a consequence of modelling the two case-studies discussed in previous chapters, as guidelines for future users who would like to model SPLs in Event-B. This will improve reusability of formal models and save time and effort in modelling similar systems. These patterns make use of existing Event-B (de)composition techniques. The two modelling patterns suggest the modelling of a product line into a set of generic features and as a set of separable features. These features could be reused when modelling further members of the product line by minimising the modelling and proof effort through reuse. Since these guidelines have been applied to real life case-studies which are available online, future users can also benefit from looking at the application of these modelling styles. We have also highlighted a number of tooling requirements which would further facilitate the modellers in applying these patterns and achieving the benefits of reusability in Event-B.
Chapter 8

Conclusion & Future Work

8.1 Summary

In this thesis, we have given an overview of the work we have done to support our research contribution. Our objective was to suggest methodological and tool support for feature-oriented reuse in Event-B for modelling software product lines using case-study experiments. Event-B, a recent formal specification language, provides composition/decomposition support but our aim was to investigate whether the existing techniques are enough for modelling SPLs, and if not, what else could be done to support this? We used a well-known product line modelling technique called “Feature Modelling” for product line development in Event-B. This also supports the concepts of model-driven architecture (MDA) and hence can be used to gain the benefits of both the product lines and MDA. We extended existing feature modelling notations for our domain-specific feature modelling in Event-B.

We started by modelling the production cell (PC) case-study, in order to generate requirements for the tools while seeking patterns for composition and exploring reuse for modelling SPLs in Event-B. The PC case-study specifies a real-life metal processing plant and proved quite useful. This was modelled in three different ways, i.e., physical component-based, controller-based and variability provided through context. The component-based PC modelling resulted in a product of PC product line which models a particular topology of physical components. The variability can be seen in the number of components and how these are connected. In order to build a variant of PC, we modelled another topology having more components (i.e., two presses and robots). This topology was modelled by reusing models and their proofs through refactoring. The reuse statistics clearly showed significant reduction in modelling and proof effort when modelling a variant of PC after the first one. The controller-based modelling involves generalising and decomposing the requirements specification of PC based on the controllers of the plant, i.e., movement, rotation etc. This also makes these controllers
generic which must be specialised and composed to build a particular component of the plant. This approach was more reuse oriented as the controllers were fine-grained models which could be reused while building various components of the PC. We used generic instantiation to instantiate the movement and magnet controllers for modelling the crane. This shows the amount of possible reuse by using this style of modelling PC. Further tooling requirements were also revealed during this experiment. In the third style of modelling PC, a generic model of a product line (PL) is specified in an Event-B machine and the variability is provided through the context. Different variants of a PL could be modelled by switching contexts and hence without any proof burden. This could be a useful activity while defining the scope of a product line and avoiding proof effort upfront.

After modelling the PC in three ways, we explored how we can use feature-oriented approach to increase reusability of Event-B specifications using existing Event-B techniques and suggesting new composition patterns to build variants of a PL by composing different features. The composition approach that we suggested is less restrictive which allows the addition of further information to glue various components together and we called it ‘feature composition’. We have developed a tool as a Rodin plug-in that allows us to compose Event-B features. At the moment, this tool does not provide any proof reuse which was then explored using other techniques such as generic instantiation. This exercise generated tooling requirements for discharging already discharged proof obligations (POs) when composing features as discussed in the future work section.

We have also developed a feature modelling tool, as part of this research, that allows us to build feature models of a product line along with any constraints between various features that must be maintained during the configuration. A feature model represents a product line and it can be configured by selecting various features required in a particular instance of the product line. Any conflict that arises during the selection of features must be resolved before these features can be composed to build a product. This conflict resolution is part of the feature composition utility which enables us to detect and resolve conflicts such as elements with same name etc. The tool also enforces all the constraints specified such as includes and excludes. The tool provides validation at two stages, i.e., firstly while building the feature models and secondly during the configuration. The tool has been useful in modelling PL using the two case-study examples and configuring different set of features to model different products.

After modelling the production cell, we modelled another case-study (ATM) to further explore our feature-oriented reuse framework for Event-B and see if existing Event-B tools and techniques could be used for such development. We started with a feature model of ATM and modelled some features in Event-B. An interesting scenario came up where we mixed both types of decomposition (shared-variable (SVD) and shared-event (SED)) in a single development for feature and architectural decomposition. An
ATM model was first decomposed using SVD and then the decomposed models (components) were further refined independently while maintaining the restrictions of SVD style. These components were further decomposed using SED to be refined separately. This modelling style provided a few interesting future work directions and tooling requirements such as generating external events in existing features and automatically discharging their corresponding POs when adding new features to model a variant of a product line. This pattern of modelling will work if the decomposition architecture is same across different features being modelled and that the restrictions of both the decomposition techniques are maintained. So, our experiment with this case-study suggested that we can use existing techniques for feature-based development in Event-B. But the restrictions of (de)composition techniques of Event-B limit us from achieving full benefits of software product lines engineering. Following is the summary of our research contribution.

8.2 Research Contribution

Our main contribution is to provide methodology and tool support for feature-oriented reuse in Event-B formal method. Although this is a well known technique within product line community, this has not been explored to this extent within Event-B domain. We have applied existing techniques of Event-B to our developed case-studies in different ways which include: horizontal and vertical refinement, decomposition/composition, generic instantiation and pattern for modelling control systems. This enabled us to suggest generic patterns of reuse that could be used to efficiently apply feature modelling in Event-B. This also required us to develop tool support for feature modelling in Event-B. Based on this, we have provided guidelines for applying these patterns for future users and highlighted limitations of various Event-B approaches and requirements for future tool support.

We have evaluated existing Event-B (de)composition approaches (i.e., SVD/SED) by applying to our case-study examples and provided statistics in terms of reusability, modelling and proof effort. This includes the advantages and disadvantages of applying these approaches in a particular situation that could help a modeller in deciding which of these would be more efficient without trying all of these. For example, while modelling PC, we modelled and refined the same set of requirements so that we could apply both SVD and SED. Modelling the system to be used for SVD, which seemed natural way of modelling, suggests that the decomposed components had large number of external events and less number of variables (i.e., large models; less readable and difficult to manage); whereas in case of SED the resulting components have smaller number of events and more variables (i.e., small model; easy to read and manage). The use of SVD also places a restriction on further refinement of decomposed components which is to avoid refining external events and shared variables. Although this type of restriction
is not there when using SED, this requires us to perform vertical refinement before the decomposition in order to prepare the model to be decomposed. More variables are introduced to make the state of decomposed components disjoint. We have used both horizontal and vertical refinement during our case-study work. It seemed feasible to apply horizontal refinement earlier for introducing functional requirements before decomposing a problem into various features and then vertically refine these features to introduce design mechanism.

The two generic patterns of reuse are suggested for feature-oriented modelling in Event-B as a result of this research. These include modelling a system as a set of generic features ($P_G$) and as a set of separable features ($P_S$).

The pattern $P_G$ requires the modeller to generalise and group the requirements for common functionality among a number of components of a system as a generic requirements feature. For example, a generic controller feature that exhibits a common functionality and can be reused to model several components of a system. This means leaving out the requirements that are specific to a particular component of a system. After modelling these generic requirements features in Event-B, these could then be instantiated and composed using the suggested feature composition technique to model various components of a system. Since the existing composition techniques of Event-B could not be used due to their restrictions, this pattern has some limitations in terms of proof effort which are highlighted as future research direction for proof reuse.

The second pattern $P_S$ follows the natural way of modelling product lines as practiced within the SPL community. Initially a set of features for a base product are developed and further features could be added later on to model variants of a product line. Once the design of base features has been agreed, different teams can work independently to develop additional features. This pattern suggests the use of SVD for problem decomposition where a product line model is decomposed into a number of features. After refining (both horizontally and vertically) these features separately, these can then be architecturally decomposed using SED for implementation purposes. The restriction of this pattern, apart from those of the SVD and SED, is to ensure that all the features of a product line follow the same architectural design. These architecturally decomposed components could be further refined (if needed) and then composed (i.e., for subsystem consolidation) using existing composition techniques of Event-B (SVC/SEC) as applicable, i.e., components having shared variables are composed using SVC whereas the rest of these are composed using SEC. These consolidated subsystems could again be refined to concrete implementation. Additional features for modelling a different product of a product line must be modelled in the same manner following the same architectural design up to the point of architectural decomposition using SED. By doing so, the existing features could be reused without any modelling and proof effort. These additional features may lead to the feature interaction problem where adding a feature could require
the existing feature(s) to be modified. This is left as a future work as discussed in the net section.

We have also provided guidelines for feature-oriented modelling in Event-B based on these modelling patterns and their application on real-life examples. These guidelines provide details of various steps that a modeller needs to take when applying these patterns to benefit from reusability in Event-B. These also highlight any restrictions of the patterns along with their advantages and disadvantages.

8.3 Future Work

Based on the work presented in this thesis, we have highlighted a number of future directions below which could lead to further contribution to this area of research.

- **Reusing proofs associated with Event-B features during composition**
  The aim of a composition technique is to make sure that the composite model is correct by construction and does not need to be reproved. This is only possible when we use the existing composition techniques of Event-B (i.e., SVC and SEC). These two techniques have constraints that restrict the user to benefit from full potential of feature modelling in Event-B. We suggested another style of composition which is less restrictive and more suitable for feature modelling but that does require the composite model to be reproved. Hence, reusing proofs associated with the Event-B features during composition is an interesting area for future work. At the moment, our feature-oriented product line approach in Even-B lacks this unless we follow the modelling pattern of ATM example. Some work has been done by Sorge et al. [132] which deals with invariant proof obligations for composing features. This does not support feature refinement and event fusion which is required to complement our feature modelling framework.
  We can investigate how to reuse the proofs by comparing POs for the models being composed and the composite model and see if theorem provers can be guided to automatically prove the composite models using POs of the models being composed. This work could be done by extending the Rodin PO generator and theorem provers which will require considerable amount of time and effort. But this can certainly improve productivity and will constitute a good contribution towards this research area.

- **Well defined composition patterns**
  We would like to explore and develop algebraic properties for the composition patterns as discussed in [32] for feature composition. By this we mean to specify the properties for the composition operators and how these will behave in different situations. We could also represent our feature models as an equation and
vice versa. This will enable us to reason about the composed models using these properties. We will look at the difference in behaviours of the composed models when the composition patterns are applied in different orders and to see whether such ordering plays any role in the composition. We might be able to suggest the possible compositions that can take place. With the help of algebraic properties, we might be able to figure out whether composed models in a different way are semantically equivalent. At the moment we do not have any criteria for semantic equivalence of our composite models and we can only see whether the composed models are syntactically equal and correct.

- **Composing asymmetrical refinement chains**
  
  It is not always possible to have same number of refinement steps in a refinement chain. So, when composing two or more models having different number of refinement levels that do not align with each other, what would be the structure of the composed model? If we have same number of refinements steps in the models being composed, then we can compose models at each refinement level. There is also a question of the level of abstraction being modelled at a particular refinement level and how well that suits the composition of various models at that level resulting in conflicts which may be resolved during the composition if we have appropriate tool support.

- **Feature modelling tool improvements**
  
  We have highlighted a number of requirements based on our work, for our feature modelling tool in Section 7.3. These include: refactoring support, topology modelling, shared-variable style composition and composition replay support. It would also be quite useful to have versioning support for feature models so that a feature model instance could be traced back to its base feature model. This would help in making changes to a feature model instance as expected of any model-driven development. By providing these facilities as part of our feature modelling tool, we could save lot of user time and effort in modelling product lines using feature-oriented modelling in Event-B.

- **Feature modelling support for UML-B**
  
  UML-B provides a UML-like front end for modelling in Event-B which makes it more user-friendly. Hence, it would be interesting to provide our feature-oriented reuse approach for product line modelling in UML-B. For this, we will have to extend our feature modelling tool to build feature models that can be mapped to UML-B models. We will have to lift our composition mechanism so that these can be used for composing UML-B models, although, UML-B models use Event-B language at the backend.
• Mapping and traceability tool support – From requirements specifications to feature model to Event-B refinements

There has always been the need to transform user requirements in narrative form to an executable artefact and this is one of the complex and important tasks in the software development life cycle (SDLC). This is because the rest of the process depends on this activity and any mistakes made at this point is multiplied at each of the following phases of the SDLC [121]. It would be very useful if we can develop a mechanism to transform user requirements well established in narrative form to the corresponding feature models. Abrial suggested a way for presenting the requirements into various categories in his Event-B book [22]. This also helps in prioritising the requirements and can be dealt with accordingly. The ProR tool [89] is provided as a Rodin plug-in which can be used to link narrative requirements to Event-B models and also provides traceability support. This approach could be adopted and extended for linking requirements to feature models and the associated Event-B specification.

The requirements specifications in narrative form could be used to build a feature model. The feature model is then mapped on to Event-B features. There may be a need to trace the requirements specification down to concrete implementation and vice versa. Any tool support for feature-oriented modelling should have such traceability functionality that could enable the users to trace back to the requirements specification if a change is required in a concrete model. For example, if a PO could not be discharged at a lower refinement level, which means an ambiguity in the requirements specification must be fixed or if a conflict could not be resolved during feature composition which requires altering the constraints in the feature model. So, we should be able to track the origin of features and the requirements and any changes made to them should be recorded for future needs. This will enable the complete development of formally verified software systems while utilising the strengths of formal methods, model-driven development, product-lines and generative development.

• Detecting and resolving feature interactions

Feature interactions [58] is an important issue that must be addressed for a feature-based product line approach. When additional features are added to build a product line member, the behaviour of the overall system or existing features might be affected resulting in feature interactions. This issue must be dealt with by detecting and resolving critical feature interactions that may violate the overall specification of the system leading to unwanted behaviour. Our proposed feature-oriented reuse approach must also address this feature interaction problem. We could possibly apply existing feature interaction detection approaches from FOSD and product lines community (such as [30]) and from formal methods (such as [56]) to our case-studies and appropriate tooling could then be developed. Our formal
approach using Event-B could be quite useful in applying model checking for this purpose. Once feature interactions are detected, these can possibly be resolved by providing invariants in the composite model which could then be verified by Event-B tools.
Appendix A

Production Cell Requirements
Specification

Below is the list of requirements for Production Cell case-study. Each requirement falls under a particular category, i.e., SAF: Safety Requirements, FUN: Functional Requirements, EQP: Equipment Requirements, PERF: Performance Requirements.

1. Production cell plant processes metal blanks (FUN)
2. Blanks are loaded into the system through a conveyor feed belt (EQP)
3. The feed belt has a sensor towards the end which informs whether a blank is present at the end to be delivered (EQP)
4. A motor is used to run the conveyor feed belt (EQP)
5. Blanks are loaded onto the feed belt only if it is not already full (SAF)
6. An elevating rotary table receives blanks from the feed belt (FUN)
7. Only deliver a blank to the table if it is neither elevated nor rotated and does not already have a blank on it (SAF)
8. The table elevates and rotates to position the blank for the robot to pick it up using two motors (FUN)
9. The table must not rotate clockwise if it is in a position to deliver a blank (SAF)
10. The table must not rotate if it is at the position to receive blanks from the feed belt (SAF)
11. The table must not move down if it is rotated (rotate backward first and then move down) or if it is already not elevated (SAF)
12. Sensors are used to detect the position of the table (EQP)
13. The robot has two arms (EQP)
14. The robot arms can move horizontally using motors (EQP)
15. Each robot arm has an electromagnet gripper at the end to pick and drop a blank (EQP)
16. The robot rotates forward and backward using a motor to position itself towards the table, press and the deposit belt (FUN)
17. The robot should rotate only if the press is in a safe position to avoid collision (SAF)
18. The robot must not rotate clockwise if arm1 is pointing towards the table and it must not rotate anti-clockwise if the arm1 is facing the press (SAF)
19. Extend/Retract Arm1 if the robot is facing table or facing press while the press is in middle position (SAF)
20. Extend/Retract Arm2 if the robot is facing deposit belt or facing the press while the press is in lower position (SAF)
21. A press stamps (forges) the blanks (EQP)
22. Press can have three positions to receive, forge and deliver the blanks as the robot arms are in a different planes (FUN)
23. Arm1 drops the blank in the press when it is in the middle position (FUN)
24. The press moves to high position to forge the blank (FUN)
25. The press moves to low position after forging the blank (FUN)
26. The press moves to middle position after the processed blanks is removed from it (FUN)
27. The press must not move downward if it is in low position and must not move upward if it is in high position (SAF)
28. A blank is only loaded in the press if it is empty (SAF)
29. Arm2 of the robot removes the blank from the press when it is in the low position and drops onto the deposit belt (FUN)
30. A deposit belt is used to remove the processed blanks out of the system (EQP)
31. The Deposit belt has a sensor towards the end which informs the controller whether a blank is present at the end to be delivered (EQP)
32. A motor is used to run the deposit belt (EQP)

33. Blanks are loaded onto the deposit belt only if it is not full already (SAF)

34. Forged blanks are removed from the deposit belt by some external mechanism which also changes the status of improperly forged blanks to unfoged state for reprocessing (FUN)

35. A travelling crane is used to bring the unforged blanks back from the deposit belt to the feed belt (EQP)

36. The crane has an electromagnet gripper to pick and drop the blank (EQP)

37. The crane gripper can move vertically to pick and drop a blank using a motor (FUN)

38. The crane gripper can move horizontally between the deposit belt and the crane using a motor (FUN)

39. The crane picks the blank from deposit belt if there is one at the end of the belt (FUN)

40. The crane drops a blank on to the feed belt if it is not already full (FUN)

41. The crane should only move towards feed belt if it is positioned on the deposit belt and vice-versa (FUN)

42. The sensors at both ends of the crane inform the controller whether it is positioned at the deposit belt or the feed belt (EQP)

43. The crane should be moved back to the deposit belt after dropping a blank on the feed belt for better efficiency (PERF)

44. Blanks should not be dropped outside the designated places (SAF)

45. Various machines in the plant should not collide with each other and should operate in a safe way (SAF)
Appendix B

ATM Requirements Specification

Following is the list of requirements features for ATM product line.

1. **Cash-withdrawal & Check balance (done by Mar Yah [123])**
   
   **Description:**
   
   These two features have already been modelled in UML-B and the generated Event-B models are available for us to use.

2. **Pin service**
   
   **Description:**
   
   The user can change their card pin.

   **Pre-condition:**
   
   Valid card is inserted in the ATM
   
   A new pin is entered
   
   New pin should be 4 digits long

   **Post-condition:**
   
   Pin for the card is changed

3. **Cash deposit**
   
   **Description:**
   
   The user can deposit cash into their account using the ATM.

   **Pre-condition:**
   
   Valid card is inserted in the ATM
   
   Deposit amount provided by the user
   
   Cash counted by the machine
Both the user and machine amount are matched

**Post-condition:**
- User account is credited with the amount
- ATM cash is topped up with the amount

4. **Manage accounts**

**Description:**
The user can add/remove an account at the card

**Pre-condition:**
- Valid card is inserted in the ATM
- User provides the account information

**Post-condition:**
- New account is added/removed

5. **Balance transfer**

**Description:**
The user can transfer money between their added accounts. This requires that the user has more than one accounts associated with this card. This feature is also used for making payments to credit card.

**Pre-condition:**
- Valid card is inserted in the ATM
- User selected crediting account
- Credit Amount

**Post-condition:**
- User selected account is credited with the amount and the main account is debited at the same time

6. **Cash-withdrawals (fees charged for withdrawals)**

**Description:**
The user is charged a fee for cash-withdrawals using the ATM machine. This requires the user must be told of the charges before the transaction and should be able to cancel the transaction.

**Pre-condition:**
- Valid card is inserted in the ATM
- User account balance is greater than withdrawal amount including the fee
- User wants to continue after told about the fee being charged
Appendix B ATM Requirements Specification

Post-condition:
User account is debited by the amount plus the fee
User is given the cash
ATM cash stock is deducted by the amount given to the user

7. Print mini-statements

Description:
The user should be able to print mini-statement for a particular bank account.

Pre-condition:
Valid card is inserted in the ATM

Post-condition:
Provides the user with a mini-statement

8. Activate Card

Description:
The user can activate their new card. The user is prompted to change their password before making any transaction.

Pre-condition:
A new card is inserted in the ATM
Old and new pin is provided by the user

Post-condition:
Card is activated and a new pin is set

9. Order New Card

Description:
The user can order a new card.

Pre-condition:
Valid card is inserted in the ATM

Post-condition:
A new card request is sent to the bank

10. Order Statements by post

Description:
The user can order bank statement for a particular account and time period by post.

Pre-condition:
Valid card is inserted in the ATM
User provides the time period for the statement

Post-condition:
A postal statement request is sent to the bank

11. **Mobile Top Up**

   **Description:**
   The user can top up their mobile phone using the ATM. This requires the user has registered a mobile phone number with this card. There is a limit of minimum and maximum top up amount in one transaction

   **Pre-condition:**
   Valid card is inserted in the ATM
   The user provides the mobile number and network

   **Post-condition:**
   A top up request is sent to the bank with the mobile number, network and the top up

12. **Cash-withdrawal using foreign card**

   **Description:**
   The user can withdraw using a foreign card, this will perform additional task at the bank site for currency conversion, fees and inter-bank communication

   **Pre-condition:**
   Valid card is inserted in the ATM
   User account balance is greater than withdrawal amount including the fee
   User wants to continue after told about the fee and conversion charges

   **Post-condition:**
   User account is debited by the amount after conversion including any fees
   User is given the cash
   ATM cash stock is deducted by the amount given to the user
References


REFERENCES


REFERENCES


REFERENCES


