RULE ENHANCED BUSINESS PROCESS MODELING OF SERVICE ORIENTED ARCHITECTURES

BY

LUIS ROCHA

A project submitted in partial fulfillment
Of the requirements for the degree of

MASTER OF SCIENCE in INFORMATION SYSTEMS

Athabasca, Alberta
May, 2011

© Luis Rocha, 2011
DEDICATION

To my parents Domingos and Sidel, who have always supported and motivated me in pursuing education.
ABSTRACT

Business process modeling is the most promising research direction in modeling service orchestrations and choreographies. Existing approaches do not fully support best practices for orchestration and choreography modeling (workflow, service interaction and message exchange patterns); have limited support for representing business rules and vocabulary; and cannot generate complete service composition descriptions from business process models.

The Rule-enhanced Business Process Modeling (rBPMN) language is created to overcome these issues by integrating Business Process Modeling Notation (BPMN) with REWERSE Rule Markup Language (R2ML), a rule modeling and interchange language, and providing rule and choreography support. As a new language, rBPMN is conceptually defined but it lacks of an implementation, thus limiting its use and the evaluation of its feasibility and effectiveness. The goal of this master’s project is to provide tooling support for rBPMN by designing and implementing an rBPMN modeling editor using a model driven engineering (MDE) approach. The rBPMN Editor is implemented as an Eclipse plug-in using the Graphical Modeling Framework (GMF) and the Eclipse Modeling Framework (EMF), where a set of models is defined (domain, graphical, tooling, and mapping), and model transformations generate the editor source code in Java. Our solution consists of a multi-diagram editor, where a main diagram represents the business process using a BPMN extended notation, and sub-diagrams represent the details of business rules and vocabulary using the graphical concrete syntax of R2ML, which is also known as UML-Based Rules Modeling Language (URML) notation. Sub-diagrams are connected to the main process diagram through rule set elements. This multi-diagram approach avoids that business rules details interfere with the overall business process visualization. The rBPMN Editor supports
modeling service compositions using both interaction and interconnected behavior models, and all types of rules: derivation, integrity, production and reaction rules. The project has shown the feasibility of the practical implementation of the rBPMN language, while effectiveness of the language is investigated through four groups of business process patterns (workflow, service interaction, message exchange, and business rules patterns for agile business processes), which are well-known in the business process modeling community. Moreover, the effectiveness of rBPMN and its implementation has also been evaluated on three different real-world case studies.
ACKNOWLEDGMENTS

I sincerely appreciate all support from Dr. Dragan Gašević, Dr. Ebrahim Bagheri, and Dr. Milan Milanović. Dr. Gašević, my project supervisor, introduced me to the Model Driven Engineering concepts during COMP 610 Software Engineering, helped me defining this project topic, and provided me guidance and motivation through the whole project. Dr. Bagheri, my project co-supervisor, helped me reviewing this report, and provided guidance for its structure and presentation. Dr. Milanović worked with me on the rBPMN Editor, helping me with the rBPMN language and the technologies used, and contributing with the business process editor, while I contributed with the rules editors. I also would like to thank Dr. Kinshuk for allowing me to extend my MScIS program in order to finish this project, and Bardia Mohabbati from Simon Frasier University for supporting us on the development infrastructure.
TABLE OF CONTENTS

Chapter I .......................................................................................................................... 1

Introduction .................................................................................................................. 1

Chapter II ..................................................................................................................... 5

Review of Related Literature ...................................................................................... 5

Model Driven Engineering ............................................................................................ 5

Model Transformation ................................................................................................. 6

Model Driven Architecture ......................................................................................... 10

Eclipse Modeling Project ............................................................................................. 12

Model Integrated Computing ....................................................................................... 16

Service Oriented Architecture ...................................................................................... 17

MDE Approaches for Service Oriented Architectures .................................................. 21

Business Process Modeling .......................................................................................... 25

The Business Process Modeling Notation .................................................................. 26

Business Process Based Modeling of SOAs ................................................................. 29

Business Rules ............................................................................................................ 33

R2ML ........................................................................................................................... 34

URML .......................................................................................................................... 37

Mappings Between URML and R2ML XML ............................................................... 39

Mappings Between R2ML and R2ML XML ............................................................... 40

Business Rules and Business Process Models Integration .......................................... 41

The Rule-enhanced Business Process Modeling Language ......................................... 48

Message Exchange Patterns in rBPMN ...................................................................... 51
| Service Interaction Patterns in rBPMN ................................................................. | 52 |
| Workflow Patterns in rBPMN .................................................................................. | 55 |
| Business Rules Patterns for Agile Business Process in rBPMN ............................ | 57 |
| Methodology for Developing Rule-enabled SOA .................................................. | 58 |
| Chapter III ............................................................................................................. | 61 |
| Methodology and Design ....................................................................................... | 61 |
| Requirements Analysis ......................................................................................... | 62 |
| Architecture .......................................................................................................... | 65 |
| MDE Framework ...................................................................................................... | 65 |
| Diagram Partitioning ............................................................................................. | 66 |
| Component View ..................................................................................................... | 68 |
| rBPMN Editor Models ............................................................................................. | 71 |
| Domain Model ......................................................................................................... | 72 |
| Events and Activities ............................................................................................ | 73 |
| Rules Activities and Process Activities .............................................................. | 74 |
| Rules Events and Process Messages ..................................................................... | 74 |
| RuleSet Vocabulary Entries and Events .................................................................. | 74 |
| Abstract Classes for Rules .................................................................................... | 75 |
| Bidirectional Association between Rules and Atoms ........................................... | 76 |
| Derivation Rule Conclusions ............................................................................... | 77 |
| Condition Filters .................................................................................................... | 78 |
| Class Variables for Conditions and Conclusions ............................................... | 78 |
| Graphical Definition Model .................................................................................. | 79 |
Evaluation and Analysis ................................................................. 154
Development Approach .................................................................. 154
rBPMN Editor Implementation.......................................................... 155
Business Process Patterns Support...................................................... 157
Comparison between Strelka and rBPMN Editor................................. 165
Chapter VII ..................................................................................... 171
Conclusion ....................................................................................... 171
References ....................................................................................... 175
Appendix A – Workflow Patterns.......................................................... 188
Appendix B – Message Exchange Patterns .............................................. 198
Appendix C – Service Interaction Patterns ............................................ 206
Appendix D – Business Rules for Agile Business Process Patterns............. 222
Appendix E – rBPMN Editor Source Code ............................................ 230
Appendix F – rBPMN Editor Screencasts ............................................. 231
LIST OF TABLES

Table 1. Classification of Model Transformations .......................................................... 7
Table 2. Model Transformation Languages ................................................................... 9
Table 3. MDE Implementations .................................................................................. 17
Table 4. MDE Approaches for SOA ........................................................................... 21
Table 5. URML and R2ML XML Mappings ................................................................. 40
Table 6. Business Rules and Business Process Integration Approaches ..................... 47
Table 7. Service interaction patterns support of business process modeling languages .... 54
Table 8. Workflow patterns support of business process modeling languages ............. 56
Table 9. Business Rules Patterns for Agile Business Process in rBPMN ....................... 57
Table 10. rBPMN Editors ......................................................................................... 68
Table 11. rBPMN Editor Components ....................................................................... 70
Table 12. Attributes for Conditions and Conclusions .................................................... 78
Table 13. R2ML Graphical Concrete Syntax (URML Extended) ................................. 83
Table 14. Creation Tools for Rules Editor Tool Palette ................................................ 86
Table 15. Mapping definitions for connecting objects of rules editors ......................... 90
Table 16. Constraints for rules editor link mappings ................................................... 92
Table 17. rBPMN Editor New File Wizards ................................................................ 106
Table 18. Patterns used for the English Auction Case Study ........................................ 121
Table 19. Patterns used for the Dutch Auction Case Study .......................................... 131
Table 20. Patterns used for the UServ Product Derby Case Study ................................. 142
Table 21. Strelka and rBPMN Editor Comparison ...................................................... 166
Table 22. Strelka and rBPMN Editor Comparison - Rules and Vocabulary Support ........ 168
LIST OF FIGURES

Figure 1. The four-layered MDA architecture ................................................................. 11
Figure 2. Overview of GMF-based development. Adapted from Eclipse Foundation (2011m). ........................................................................................................................................................................................................................................... 15
Figure 3. Core modeling elements in BPMN ........................................................................ 27
Figure 4. Sales and Shipping processes (BPMN). .................................................................. 28
Figure 5. RuleBase and RuleSet in the R2ML metamodel. Adapted from R2ML (2006). .... 35
Figure 6. Atoms in the R2ML metamodel. Adapted from R2ML (2006). ......................... 36
Figure 7. Events in the R2ML metamodel (R2ML, 2006). .................................................. 37
Figure 8. Rule elements in URML ....................................................................................... 37
Figure 9. Derivation rule in URML ..................................................................................... 38
Figure 10. Production rule in URML ................................................................................... 39
Figure 11. rBPMN Rule Gateway element ......................................................................... 49
Figure 12. rBPMN One-to-Many Send pattern ................................................................... 53
Figure 13. Passing a participant reference over the message flow in rBPMN. .................. 53
Figure 14. Methodology for developing rule-enabled SOAs (Milanović, 2010). ............. 58
Figure 15. Sample of a business process diagram in rBPMN (Milanović & Gašević, 2009). 63
Figure 16. Sample of a business rule diagram in rBPMN (Milanović & Gašević, 2009). .... 64
Figure 17. Core Package in the rBPMN metamodel (Milanović & Gašević, 2009). ......... 67
Figure 18. rBPMN Editor component view. For the sake of illustration, only the reaction rule component is displayed, but similar components for other types of rules are also provided. 69
Figure 19. rBPMN Editor models ....................................................................................... 71
Figure 20. Reaction rule with events and activity .............................................................. 73
Figure 21. RuleSet vocabulary entries................................................................. 75
Figure 22. Abstract classes for rules ..................................................................... 76
Figure 23. Extending AndOrNafNegFormula for a bidirectional association with a Rule. ... 77
Figure 24. Derivation rule conclusions.................................................................. 77
Figure 25. R2ML ObjectClassificationAtom (R2ML, 2006). .................................... 78
Figure 26. Graphical notation for R2ML complex events....................................... 80
Figure 27. Example of nested complex events. .................................................... 81
Figure 28. Graphical representation for integrity rule. ......................................... 82
Figure 29. Visualization of mapping definition of rBPMN events.............................. 88
Figure 30. R2ML transformations. ....................................................................... 96
Figure 31. Creating rBPMN diagram wizard....................................................... 99
Figure 32. rBPMN Editor with an empty diagram. ............................................ 100
Figure 33. Tool palette of the rBPMN Process Editor............................................ 101
Figure 34. Process design with rBPMN Editor for the Patient Info Request sample as per case study from (Milanović & Gašević, 2009)......................................................... 102
Figure 35. Interaction model for the Patient Info Request sample.......................... 104
Figure 36. rBPMN Reaction Rules Editor with an empty diagram.......................... 105
Figure 37. Creating rules diagram wizard............................................................ 106
Figure 38. Tool palettes of the rBPMN rules editors: Derivation Rule, Integrity Rule, Production Rule, and Reaction Rule................................................................. 108
Figure 39. Data and rule design with the rBPMN Editor for Patient Info Request sample as per the case study from (Milanović & Gašević, 2009)......................................................... 110
Figure 40. rBPMN Editor – Connecting rules activities to process tasks............. 111
Figure 41. rBPMN Editor – Connecting process messages to rules events ............... 112
Figure 42. Rules modeling in the rBPMN process editor ........................................ 113
Figure 43. Model file in the rBPMN Editor ............................................................... 114
Figure 44. English Auction high-level business process in rBPMN language .......... 118
Figure 45. English Auction business process in rBPMN language .......................... 122
Figure 46. The BidRequest message and related data objects ................................ 124
Figure 47. English Auction Rule Set 1 .......................................................... 125
Figure 48. English Auction Rule Set 2 .......................................................... 126
Figure 49. English Auction Rule Set 3 .......................................................... 128
Figure 50. English Auction interaction model ....................................................... 130
Figure 51. Dutch Auction business process in rBPMN language ....................... 133
Figure 52. Dutch Auction Rule Set 1 .......................................................... 135
Figure 53. Dutch Auction Rule Set 2 .......................................................... 136
Figure 54. Dutch Auction Rule Set 3 .......................................................... 137
Figure 55. UServ Product Derby high-level business process in rBPMN language .... 140
Figure 56. UServ Product Derby business process in rBPMN language ............... 144
Figure 57. UServ Product Derby vocabulary ...................................................... 146
Figure 58. UServ Product Derby Rule Set 1 ...................................................... 147
Figure 59. UServ Product Derby Rule Set 2 ...................................................... 149
Figure 60. UServ Product Derby Rule Set 3 ...................................................... 150
Figure 61. UServ Product Derby Rule Set 4 ...................................................... 151
Figure 62. UServ Product Derby Rule Set 5 ...................................................... 152
Figure 63. UServ Product Derby interaction model ............................................... 153
Figure 64. Creating more than one graphical element for the same model element. .......... 156

Figure 65. The Send/Receive pattern interaction model: (a) proposed by Milanović(2010); (b) implemented using the rBPMN Editor. ............................................................. 160

Figure 66. The Dynamic Routing pattern (interaction model) proposed by Milanović (2010). .............................................................................................................. 161

Figure 67. The Dynamic Routing pattern (Interaction Model) – rBPMN Editor. ............ 162

Figure 68. The Sequence pattern using the rBPMN Editor. ........................................ 188

Figure 69. The Parallel Split pattern using the rBPMN Editor.................................... 188

Figure 70. The Synchronization pattern using the rBPMN Editor. ............................... 188

Figure 71. The Exclusive Choice pattern using the rBPMN Editor. ............................... 188

Figure 72. The Simple Merge pattern using the rBPMN Editor.................................. 189

Figure 73. The Multiple Choice pattern using the rBPMN Editor. ............................... 189

Figure 74. The Structured Synchronizing Merge pattern using the rBPMN Editor........ 190

Figure 75. The Multiple Merge pattern using the rBPMN Editor. ............................... 190

Figure 76. The Discriminator pattern using the rBPMN Editor. ................................. 191

Figure 77. The Arbitrary Cycles pattern using the rBPMN Editor............................. 191

Figure 78. The Implicit Termination pattern using the rBPMN Editor. ....................... 192

Figure 79. The Multiple Instances without Synchronization pattern using the rBPMN Editor. .............................................................................................................. 192

Figure 80. The Multiple Instances with a Priori Known Design Time Knowledge pattern using the rBPMN Editor (process diagram).............................................. 193

Figure 81. The Multiple Instances with a Priori Known Design Time Knowledge pattern using the rBPMN Editor (rule set diagram).............................................. 193
Figure 82. The Multiple Instances with a Priori Known Runtime Knowledge pattern using the rBPMN Editor (process diagram). ................................................................. 194

Figure 83. The Multiple Instances with a Priori Known Runtime Knowledge pattern using the rBPMN Editor (rule set diagram). ................................................................. 194

Figure 84. The Multiple Instances with no a Priori Runtime Knowledge pattern using the rBPMN Editor (process diagram). ................................................................. 195

Figure 85. The Multiple Instances with no a Priori Runtime Knowledge pattern using the rBPMN Editor (rule set diagram). ................................................................. 195

Figure 86. The Deferred Choice pattern using the rBPMN Editor ............................................. 196

Figure 87. The Milestone pattern using the rBPMN Editor ....................................................... 196

Figure 88. The Cancel Activity pattern using the rBPMN Editor ................................................. 196

Figure 89. The Cancel Case pattern using the rBPMN Editor ..................................................... 197

Figure 90. The In-Only pattern using the rBPMN Editor (process diagram). ......................... 198

Figure 91. The In-Only pattern using the rBPMN Editor (rule set 1 diagram). ....................... 198

Figure 92. The Robust In-Only pattern using the rBPMN Editor (process diagram) .......... 199

Figure 93. The Robust In-Only pattern using the rBPMN Editor (rule set 1 diagram) ....... 199

Figure 94. The In-Out pattern (with fault) using the rBPMN Editor (process diagram) ..... 200

Figure 95. The In-Out pattern (with fault) using the rBPMN Editor (rule set 1 diagram). . . 200

Figure 96. The In-Optional-Out pattern using the rBPMN Editor (process diagram)............ 201

Figure 97. The In-Optional-Out pattern using the rBPMN Editor (rule set 1 diagram) ...... 201

Figure 98. The Out-Only pattern using the rBPMN Editor (process diagram) ..................... 202

Figure 99. The Out-Only pattern using the rBPMN Editor (rule set 1 diagram) ............... 202

Figure 100. The Out-Only pattern using the rBPMN Editor (rule set 2 diagram) ............. 202
Figure 101. The Robust Out-Only pattern using the rBPMN Editor (process diagram). ..... 203
Figure 102. The Robust Out-Only pattern using the rBPMN Editor (rule set 1 diagram). .... 203
Figure 103. The Robust Out-Only pattern using the rBPMN Editor (rule set 2 diagram). .... 203
Figure 104. The Out-In pattern using the rBPMN Editor (process diagram). .................... 204
Figure 105. The Out-In pattern using the rBPMN Editor (rule set 1 diagram). .................... 204
Figure 106. The Out-In pattern using the rBPMN Editor (rule set 2 diagram). .................... 204
Figure 107. The Out-Optional-In pattern using the rBPMN Editor (process diagram). ...... 205
Figure 108. The Out-Optional-In pattern using the rBPMN Editor (rule set 1 diagram). .... 205
Figure 109. The Out-Optional-In pattern using the rBPMN Editor (rule set 2 diagram). .... 205
Figure 110. The Send pattern using the rBPMN Editor.................................................. 206
Figure 111. The Send pattern using the rBPMN Editor (interaction model). ..................... 206
Figure 112. The Receive pattern using the rBPMN Editor............................................. 207
Figure 113. The Receive pattern using the rBPMN Editor (interaction model). ..................... 207
Figure 114. The Send/Receive pattern using the rBPMN Editor..................................... 208
Figure 115. The Send/Receive pattern using the rBPMN Editor (interaction model). .......... 208
Figure 116. The Racing Incoming Messages pattern using the rBPMN Editor. ......... 209
Figure 117. The Racing Incoming Messages pattern using the rBPMN Editor (interaction model). ........................................................................................................ 209
Figure 118. The One to Many Send pattern using the rBPMN Editor. ............................ 210
Figure 119. The One to Many Send pattern using the rBPMN Editor (interaction model). .. 210
Figure 120. The One from Many Receive pattern using the rBPMN Editor ..................... 211
Figure 121. The One from Many Receive pattern using the rBPMN Editor (interaction model). ........................................................................................................ 211
Figure 122. The One to Many Send/Receive pattern using the rBPMN Editor. ............... 212
Figure 123. The One to Many Send/Receive pattern using the rBPMN Editor (interaction model). ................................................................. 213
Figure 124. The Multi-Responses pattern using the rBPMN Editor. ......................... 214
Figure 125. The Multi-Responses pattern using the rBPMN Editor (interaction model). 214
Figure 126. The Contingent Requests pattern using the rBPMN Editor. ................. 215
Figure 127. The Contingent Requests pattern using the rBPMN Editor (interaction model). ................................................................................................................................. 215
Figure 128. The Request with Referral pattern using the rBPMN Editor. ............... 216
Figure 129. The Request with Referral pattern using the rBPMN Editor (interaction model). ................................................................................................................................. 217
Figure 130. The Relayed Request pattern using the rBPMN Editor....................... 218
Figure 131. The Relayed Request pattern using the rBPMN Editor (interaction model). 219
Figure 132. The Dynamic Routing pattern using the rBPMN Editor. ................. 220
Figure 133. The Dynamic Routing pattern using the rBPMN Editor (interaction model). 221
Figure 134. The Decision Logic Abstraction pattern using the rBPMN Editor (process diagram). ................................................................................................................................. 222
Figure 135. The Decision Logic Abstraction pattern using the rBPMN Editor (derivation rule set diagram)................................................................................................................................. 222
Figure 136. The Decision Logic Abstraction pattern using the rBPMN Editor (production rule set diagram). ................................................................................................................................. 222
Figure 137. The Decision Node to Business Rule Binding pattern using the rBPMN Editor. ................................................................................................................................. 223
Figure 138. The Decision with Flexible Input Data pattern using the rBPMN Editor. 223
Figure 139. The Decision Flexible Output pattern using the rBPMN Editor (process
diagram). 224
Figure 140. The Decision Flexible Output pattern using the rBPMN Editor (production rule
set diagram). 224
Figure 141. The Constraints at Predefined Checkpoint pattern using the rBPMN Editor
(process diagram). 225
Figure 142. The Constraints at Predefined Checkpoint pattern using the rBPMN Editor
(integrity rule set 1 diagram). 225
Figure 143. The Constraints at Predefined Checkpoint pattern using the rBPMN Editor
(integrity rule set 2 diagram). 225
Figure 144. The Constraints at Multiple Checkpoints pattern using the rBPMN Editor
(process diagram). 226
Figure 145. The Constraints at Multiple Checkpoints pattern using the rBPMN Editor
(integrity rule set 1 diagram). 226
Figure 146. The Constraints at Multiple Checkpoints pattern using the rBPMN Editor
(integrity rule set 2 diagram). 226
Figure 147. The Constraints Enforced by External Data pattern using the rBPMN Editor
(process diagram). 227
Figure 148. The Constraints Enforced by External Data pattern using the rBPMN Editor
(reaction rule set diagram). 227
Figure 149. The Constraints Enforced by External Data pattern using the rBPMN Editor
(integrity rule set diagram). 227
Figure 150. The Business Rule-Based Subprocess Selection pattern using the rBPMN Editor (process diagram)........................................................................................................................................... 228

Figure 151. The Business Rule-Based Subprocess Selection pattern using the rBPMN Editor (production rule set diagram)........................................................................................................................................... 228

Figure 152. The Business Rule-Based Process Composition pattern using the rBPMN Editor (process diagram)........................................................................................................................................... 229

Figure 153. The Business Rule-Based Process Composition pattern using the rBPMN Editor (derivation rule set diagram)............................................................................................................................................ 229

Figure 154. The Business Rule-Based Process Composition pattern using the rBPMN Editor (production rule set diagram)........................................................................................................................................... 229
CHAPTER I

INTRODUCTION

In the past decade, Service Oriented Architecture (SOA) has become a promising approach to develop systems integrating business processes of multiple parties (e.g. customers, suppliers, etc.) into collaborative information systems. SOA is based on the idea of composing an application by discovering and invoking network available services that are interoperable, i.e. non programming language or operating system specific, loose-coupled, stateless, and reusable (Papazoglou et al., 2007). Multiple services can be aggregated into a single composite service in a process known as service composition. Service composition has two perspective views: single participant view (service orchestration) and multiple participant view (service choreography). Service orchestration focuses on the control flow of a single participant (Peltz, 2003), and service choreography focuses on the interaction aspects of multiple participants (Papazoglou et al., 2007). One of the initial steps for designing SOAs is to model service compositions (Papazoglou et al., 2007).

The use of business process modeling (BPM) languages is the most promising research direction in modeling service compositions (Pant and Juric, 2008). BPM approaches include UML 2.0 activity diagrams, Business Process Execution Language (BPEL), and Business Process Modeling Notation (BPMN) which is the OMG standard (Graml et al., 2007). Although BPM languages can be used to model service compositions, there are several limitations. For service orchestration modeling, BPM languages do not fully cover control flow best practices (i.e. workflow patterns), they have limited support for representing logical
expressions, business rules and vocabulary, and they cannot generate completely service compositions from process models (Milanović et al., 2009). Service choreography modeling has attracted less attention from the research community compared to orchestration modeling and it presents the following challenges: service choreography models are not well connected to underlying vocabulary and domain models, limited support for decoupling business logic from complete choreography models, redundant elements of shared business logic in service choreography models (Milanović et al., 2009b).

In order to address the abovementioned limitations, Milanović and Gašević (2009) conceptually defined a new modeling language, Rule-enhanced Business Process Modeling Notation (rBPMN), by integrating BPMN and REWERSE Rule Markup Language (R2ML), a business rule modeling and interchange language. Their approach uses Model Driven Engineering (MDE) principals. In an MDE approach, a model of the system under study should be developed first, and then transformed into executable software entity. A metamodel is a model of the modeling language that defines its syntax and semantics. Modeling languages can have abstract syntax, i.e. machine independent, or concrete syntax, i.e. graphical or textual representation.

The rBPMN language consists of an abstract syntax and a graphical concrete syntax. The abstract syntax of rBPMN is a metamodel defined in the MOF language, which is the result of the integration of the BPMN and R2ML metamodels. The rBPMN graphical concrete syntax is a combination of the BPMN graphical notation and the R2ML graphical notation, known as UML-Based Rule Modeling Language (URML).

As a new language, rBPMN is conceptually defined but it does not have a textual concrete syntax and it lacks of an implementation, thus limiting its use and the evaluation of
its feasibility and effectiveness. The goal of this project is to provide tooling support for rBPMN by designing and implementing an rBPMN modeling editor.

Our proposed approach to develop the rBPMN Editor is to use an MDE approach. We use the Graphical Modeling Framework (GMF) (Eclipse Foundation, 2011b) and the Eclipse Modeling Framework (EMF) (Eclipse Foundation, 2011a) to define models of the rBPMN Editor (domain, graphical, tooling, and mapping models), and perform model transformations to generate the editor source code in Java language. The result is a set of plug-ins that are installed and executed in the Eclipse integrated development environment (IDE). Our solution consists of a multi-diagram editor, where a main diagram represents the business process using a BPMN extended notation, and sub-diagrams represent the details of business rules and vocabulary using an R2ML graphical notation; this notation has additionally been extended in this project, as we recognized some missing elements in the existing version of the notation. Sub-diagrams are connected to the main process diagram through rule set elements. This multi-diagram approach avoids that business rules details interfere with the overall business process visualization. The rBPMN Editor supports modeling service compositions using both interaction and interconnected behavior models, and all types of rules: derivation, integrity, production and reaction rules. The project has shown the feasibility of the practical implementation of the rBPMN language, while effectiveness of the language is investigated through four groups of business process patterns that are well known in the business process community: workflow, service interaction, message exchange, and business rules patterns for agile business process. The rBPMN models for the business process patterns as proposed in (Milanović, 2010) are implemented using the rBPMN Editor, thus confirming that all business process patterns supported by the rBPMN language are also
supported by the rBPMN Editor. Moreover, the effectiveness of rBPMN and its implementation has also been evaluated on three different real-world case studies (English Auction, Dutch Auction, and the UServ Product Derby) following the methodology for developing rule-enabled SOA (Milanović, 2010). The rBPMN Editor project is collaboration with Milanović (2010) who focused on the business process editor, while this integration project focused on the rules editors and their integration with the process editor.

This project report is divided as follow. In Chapter II, we review the related literature starting with MDE and its initiatives, SOA, and MDE approaches for SOA. Then, we review business process and its modeling languages, including BPMN, and business rules, including R2ML and URML. Finally, we review the rBPMN language, its use with business process patterns, and a methodology using rBPMN for developing SOAs. In Chapter III, we present the methodology and design of the rBPMN Editor, by defining and analyzing the requirements, selecting a MDE framework, defining the editor architecture, and describing the overall design. In Chapter IV, the implementation details of the rBPMN Editor are described including how to design business processes, data and rules with it. In Chapter V, we present three case studies of real scenarios using the rBPMN Editor: English Auction, Dutch Auction, and the UServ Product Derby. In Chapter VI, we evaluate and analyze the implemented solution, the MDE framework used, and comparison to similar approaches. Finally, Chapter VII describes the conclusions and future work of this project.
CHAPTER II
REVIEW OF RELATED LITERATURE

This chapter includes a review of MDE including its approaches for developing SOAs, business process modeling, BPMN, integration of business rules into business process models, and the rBPMN language.

Model Driven Engineering

MDE is an approach to software development that suggests that one should first develop a model of a system under study; the model is then transformed into an executable software entity (Gašević et al., 2009). MDE moves the development focus from third generation programming language code to models. The goal of MDE is to increase productivity and reduce time-to-market by focusing on the problem domain rather than on the underlying technology (Sendall and Kozaczynski, 2003).

In MDE, a model is a clear set of formal elements that describes something being developed for a specific purpose and can be analyzed in different ways (Mellor, Clark, and Futagami, 2003). A metamodel is a model of the modeling language, which defines the modeling language syntax and semantics. Bézivin (2004) defines two important relationships in MDE: a particular view of a system can be represented by a model, and each model is conformant to a language of its metamodel.

Modeling languages can have abstract and concrete syntaxes. An abstract syntax is independent of any machine data structure or physical representation of the language utterances, and a concrete syntax is concerned with the physical representation of the
language utterances (Milanović et al., 2009c). Concrete syntaxes can be graphical or textual. For example, the UML standard language has an abstract syntax (UML metamodel), a concrete graphical syntax (UML graphical notation), and a textual concrete syntax, UML XML Metadata Interchange (XMI).

**Model Transformation**

An important aspect of MDE is model transformation. This section describes model transformations, including how they can be classified, and the different types of transformations.

Sendall and Kozaczynski (2003) define model transformation as the process to take source model(s) as input, and produce target model(s) as the process output, by following a set of transformation rules. These transformation rules are expressed in a model transformation language. Transformation languages can be classified as declarative, when relationships between elements of the source and target models are defined regardless the execution order, or imperative, when an explicit sequence of steps need to be executed in order to produce the target model (Gašević et al., 2009).

Model transformation can be classified according to its direction, cardinality, conformity and type, as shown in Table 1. Czarnecki and Helsen (2006) describe the first three criteria. For the direction criterion, transformations can be unidirectional, only allowing transformations from source to the target models, or bidirectional, enabling synchronization of source and target models. The cardinality criterion classifies model transformations depending on the number of models involved. A transformation could be model merging when the number of source models is greater than the number of target models, or model weaving when the number of source models is less than the number of target models. For the
conformity criterion, transformations can be classified as *endogenous* or *rephrasings* when source and target models conform to the same metamodel, or *exogenous* or *translations*, when source and target models conform to different metamodels. Brown et al. (2005) classify model transformations based on the transformation type criterion. Refactoring transformations are used to reorganize existing models based on well-defined criteria, creating a new revised version of the original model. *Model-to-model* transformations convert one or more models into another model(s). *Model-to-text* transformations convert a model into a text-based definition fragment. An example of model-to-text transformation is generating Java source code from UML models. Gašević et al. (2009) define a fourth transformation type, *text-to-model* transformations, that allow for reverse engineering of text format software artifacts into models. An example of text-to-model transformation is generating UML models from existing Java source code.

Table 1. Classification of Model Transformations.

<table>
<thead>
<tr>
<th>Criterion</th>
<th>Categories</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direction</td>
<td>Unidirectional</td>
<td>From source model(s) to target model(s).</td>
</tr>
<tr>
<td></td>
<td>Bidirectional</td>
<td>From source model(s) to target model(s) and vice-versa.</td>
</tr>
<tr>
<td>Cardinality</td>
<td>Model Merging</td>
<td>Number of target models is less than number of source models.</td>
</tr>
<tr>
<td></td>
<td>Model Weaving</td>
<td>Number of target models is greater than number of source models.</td>
</tr>
<tr>
<td>Conformity</td>
<td>Rephrasings</td>
<td>Source and target models conform to same metamodel.</td>
</tr>
<tr>
<td></td>
<td>Translation</td>
<td>Source and target models conform to different metamodels.</td>
</tr>
<tr>
<td>Type</td>
<td>Refactoring</td>
<td>Reorganizing model(s) into new revised version(s).</td>
</tr>
<tr>
<td></td>
<td>Model-to-Model</td>
<td>Converting model(s) into another model(s).</td>
</tr>
</tbody>
</table>
Sendall and Kozaczynski (2003) describe that a key challenge in MDE is to transform models into platform-specific models that can be used by code generation tools. Model transformation tools offer three different ways for defining model transformations, namely: direct model manipulation, intermediate representation, and transformation language support. In direct model manipulation, an internal model representation can be accessed and manipulated using an Application Programming Interface (API). The advantage of this approach is the usage of a general purpose programming language for model manipulation. The disadvantages are the lack of high-level abstractions for specifying transformations in general purpose languages, and the API restriction of the kind of transformations that can be performed. In intermediate representation, a model is exported in a standard format, such as XML, which can be used by external tools to transform it. This approach benefits from the fact that, for example, many UML tools can export and import models using XML Metadata Interchange (XMI). The disadvantage is that using XSLT to transform models requires experience and considerable effort, and it is also performed in a batch mode disallowing user interactions. In transformation language support, a language can be specifically developed for the transformation purpose, thus providing a set of constructs to explicitly express, compose and apply transformations.

In model transformations, the rules of transformation from source models into target models are described by mapping rules within a mapping function. These rules are described at the metamodel level, so that they are applicable to all source models that conform to a
given metamodel (Mellor et al., 2004). That is, mapping functions transform one model into another when executed, and they can be reused when an application or its dependent technology changes (Mellor et al., 2003).

Examples of the languages for defining model transformations include: XSLT, ATLAS Transformation Language (ATL), Xpand, and Query-View-Transformation (QVT). An XSLT transformation approach cannot guarantee that the transformed models are fully semantically correct with respect to the semantics of the target language, since XML and its definition languages (DTD and XSD) do not support any constraining mechanism such as OCL (Ribarić et al., 2008). XSLT supports only model-to-text and limited (XML-based) text-to-model transformations and it lacks of constraints mechanisms. ATL is a model transformation language specified as a metamodel and a textual concrete syntax, based on the Object Management Group (OMG) standard QVT (Eclipse Foundation, 2011e; OMG, 2011). It supports model-to-model transformations, and it allows imperative and declarative programming approaches for defining mapping functions. Xpand (Eclipse Foundation, 2011i) is a statically-type template language that supports model-to-text transformations, aspect oriented programming (AOP), and rich expressions (OCL-like but with Java-like syntax). The following table shows a summary of the model transformation languages reviewed.

Table 2. Model Transformation Languages.

<table>
<thead>
<tr>
<th>Language</th>
<th>Programming Type</th>
<th>Transformations</th>
<th>Constraints</th>
<th>Standard</th>
</tr>
</thead>
<tbody>
<tr>
<td>XSLT</td>
<td>Declarative</td>
<td>model-to-text and XML-based text-to-model</td>
<td>none</td>
<td>None</td>
</tr>
</tbody>
</table>
Model Driven Architecture

There are several MDE initiatives including Model Driven Architecture (MDA) (OMG, 2010), Model-Integrated Computing (MIC) (ISIS, 2011), and Eclipse Modeling Project (Eclipse Foundation, 2011). MDA is a research initiative in the MDE domain being developed by the Object Management Group (OMG). MDA enables the specification of models and their transformation to other models and complete systems (Mellor, Clark, and Futagami, 2003). In MDA, metamodels are defined using the Meta-Object Facility (MOF). Query-View- Transformation (QVT) is a standard for mapping functions between models, providing a standard scheme for querying, viewing, and transforming metamodels represented in MOF.

Gašević et al (2009) describe different levels of abstractions (viewpoints) for analyzing systems in MDA so that they can be seen independent of their target platform (operating system, network, hardware, etc.). In each level of abstraction, a system can be represented by a model of the system under study seen from that viewpoint. MDA defines three viewpoints: computational-independent model (CIM), platform-independent model (PIM), and platform-specific model (PSM). CIM does not include details of the system structure and it is known as the domain model in software engineering. PIM is a computational-dependent model that is assumed to be executed on a technology independent virtual machine. PSM completes the specification of the system by showing the integration
details of the system with the underlying platform. The main goal of MDA is to focus on PIM and CIM rather than PSM. PSMs should be automatically generated using transformation tools.

Figure 1 shows the four-layered architecture of MDA. The M3 layer contains MOF, which specifies how to define metamodels, e.g. metametamodel, generate XML schemas for interchange, and generate application programming interfaces (API) for manipulating actual models (Mellor et al., 2003). MOF is used to define metamodels and also to define MOF itself (MOF self-dependency relationship in Figure 1). The M3 layer provides a common metametamodel that enables the use and the generic management of multiple models and metamodels, and also supports their extensibility and integration (Gašević et al., 2009).

![Figure 1. The four-layered MDA architecture.](image)

The M2 layer contains all metamodels that are defined in MOF (Gašević et al., 2009). That is, definitions of modeling languages are given on this layer, as metamodeling is used
for defining modeling languages. This includes standard metamodels, such as UML and BPMN, and also custom ones. The M1 layer contains models of the real world that are represented by concepts from metamodels of the M2 layer. The M0 layer contains things from the real world that are modeled in the M1 layer.

**Eclipse Modeling Project**

Another MDE initiative is the open source Eclipse Modeling Project (EMP). As part of the Eclipse platform, it provides a unified set of modeling frameworks, tooling, and standard implementations (Eclipse Foundation, 2011h). EMP supports model-to-model transformations (ATL) and model-to-text transformations (Xpand), and both abstract and concrete syntaxes development.

For abstract syntax development, the Eclipse Modeling Framework (EMF) project provides a modeling framework and code generation facility for building tools and other applications based on a structured data model. EMF provides a similar set of languages for metamodeling and model interchange as MDA provides. Instead of using MOF, EMF uses its own metamodeling language called Ecore, which is also used to define itself (Gaševic et al., 2009). Ecore is similar to MOF in its ability to specify classes, structural and behavior features, inheritance, packages, and reflection. However, they differ in terms of life cycle, data type structures, package relationships and complex aspects of association (Steinberg et al., 2008). Ecore avoids some complexities of MOF by focusing on tool integration rather than metadata repository management, resulting in a widely applicable and optimized implementation (Steinberg et al., 2008). Although Ecore is not an OMG standard, it has substantially influenced the latest version of the MOF specification (OMG, 2011b), in terms of the layering of the architecture and the structure of the semantic core. The result is EMOF,
i.e. Essential Meta-Object Facility, a new lightweight core of the metamodel that quite closely resembles Ecore. EMF also includes a Validation Framework (VF) providing constraint definitions for Ecore metamodels through a specific API, and constraint parsing for languages, e.g. support for parsing the content of constraint elements defined in Java or OCL (Eclipse Foundation, 2011a).

For textual concrete syntax development, the Textual Modeling Framework (TMF) project provides frameworks, tools and corresponding editors based on EMF. There are two subprojects under TMF: Xtext and Textual Concrete Syntax (TCS). Xtext allows defining domain specific languages (DSL) using the Extended Backus-Naur Form (EBNF) notation, and it generates a parser, a metamodel and a fully-feature text editor for it (Eclipse Foundation, 2011f). TCS enables the specification of textual syntaxes for DSLs by attaching syntactic information to metamodels (Eclipse Foundation, 2011g).

For graphical concrete syntax development, the Graphical Modeling Project (GMP) provides a set of generative components and runtime infrastructures for developing graphical editors based on EMF and the Graphical Editing Framework (GEF) (Eclipse Foundation, 2011d). GEF allows developers to create rich graphical editors for the Eclipse IDE from existing application models. There are two major initiatives under GMP: Graphical Modeling Framework (GMF) (Eclipse Foundation, 2011b) and Graphiti (Eclipse Foundation, 2011c). GMF consists of a framework and tools to create EMF/GEF based graphical editors for Eclipse using a MDE approach. GMF consists of two main components: GMF Tooling and GMF Runtime. GMF Tooling consists of editors to create and edit models representing the graphical editor, and a generator to produce graphical editors as Eclipse plug-ins from the models.
GMF Runtime provides an execution platform for the generated plug-ins (Plate, 2006). An overview of the development process of GMF-based graphical editors is shown in Figure 2. The first step is to create a domain model. The domain model defines the non-graphical information managed by the editor. Then, a graphical definition model is created to contain information of the graphical elements that will appear in the graphical editor. This model has no direct connection with the domain models that it will provide representation and editing for. Optionally, a tooling definition model is created to be used to design the palette, menus, and toolbars. Then, a mapping model is created to link the graphical and tooling definitions to the selected domain models. Thus, the mapping model provides a total separation of the domain model and the graphical definition, allowing the graphical definition to be reused by multiple domain models. The mapping model allows for creating node mappings and link mappings. A node mapping element is used to map diagram nodes, for example, an UML class. Link mapping elements are used to create connections between elements on the diagram and they can represent several types of reference relationships of the domain model (Gronback, 2009). Link mappings can represent a domain model element or a domain model relationship. For link mappings representing domain model elements, the following information to define it: the metamodel element the connection represents, the metamodel element that contains this connection (containment feature), the source element the connection comes from (source feature), and the target element the connection goes to (target feature). For link mappings representing relationships, it is only needed to specify the target feature since GMF runtime infers the source and containment feature for it. The last model to be created is the generator model that defines implementation details for the code generation phase. Finally, the modeling editor is generated based on previous models. GMF
allows generating an Eclipse IDE plug-in or a standalone application by using the Eclipse Rich Client Platform (RCP) (McAffer, Lemieux, 2005).

Figure 2. Overview of GMF-based development. Adapted from Eclipse Foundation (2011m).

Once the source code of the modeling editor is generated, it can be customized and extended. All class methods generated by GMF are annotated with `@generated` tag indicating that it has been auto-generated. If any change is made in the generated code, it will be lost next time the modeling editor is generated from the models again. In order to customize an auto-generated method and not lose its changes, it is necessary to replace the generated tag with `@generated NOT` tag so that GMF ignores this method when regenerating the modeling editor code. The code snippet below shows an example of how to customize a generated code so that it persists if the editor code is generated again.

```java
/**
 * @generated NOT
 */
protected Connection createConnectionFigure() {
    ConditionFigureDescr figure = new ConditionFigureDescr();
    figure.updateIsNegatedDecoration();
}
In addition to GMF, GMP also includes the Graphiti project for concrete graphical syntax development. Similar to GMF, Graphiti (Eclipse Foundation, 2011c) is also EMF/GEF based. It uses an API-centric approach instead of a MDE approach as GMF, where modeling editors are developed using plain Java code. Graphiti is an alternative to GMF, but while GMF is at a mature stage, being widely adopted, and with more available documentation, Graphiti is a new project at the incubation phase, being initially adopted, and with less documentation and samples available.

**Model Integrated Computing**

In addition to MDA and EMP, Balasubramanian et al. (2006) describe Model-Integrated Computing (MIC) as another MDE research initiative being developed by the Institute for Software Integrated Systems (ISIS) at Vanderbilt University. It uses domain-specific modeling languages (DSML) to represent systems and their relationships, and also support transformations to platform-specific artifacts. DSMLs are visually defined by using Generic Modeling Environment (GME), an open source design environment for creating DSML and program synthesis environments. MIC provides a tool for building model transformations called Graph Rewriting and Transformation (GReAT) that uses sequenced graph rewriting rules for specifying the mapping functions (ISIS, 2011).

ISIS has developed a DSML tool suite called Platform-Independent Component Modeling Language (PICML) to assist in developing, configuring, and deploying systems using component middleware technology such as CORBA Component Model (CCM) (Balasubramanian et al., 2006). Another DSML tool suite developed is the Embedded Control
Systems Language (ECSL) that supports development of distributed embedded automotive applications (Balasubramanian et al., 2006).

In this section, we discussed MDE and some of its most relevant implementations. The following table summarizes the MDE implementations reviewed. In the following sections, we discuss service oriented architecture and its MDE approaches.

Table 3. MDE Implementations.

<table>
<thead>
<tr>
<th>MDE Implementation</th>
<th>Metamodel</th>
<th>Transformation</th>
</tr>
</thead>
<tbody>
<tr>
<td>MDA</td>
<td>MOF</td>
<td>QVT, ATL</td>
</tr>
<tr>
<td>Eclipse</td>
<td>Ecore</td>
<td>ATL, Xpand</td>
</tr>
<tr>
<td>MIC</td>
<td>GME</td>
<td>GReAT</td>
</tr>
</tbody>
</table>

Service Oriented Architecture

In this section, we discuss Service Oriented Architecture (SOA), including its key concepts and languages.

Papazoglou et al. (2007) define services as “autonomous, platform-independent entities that can be described, published, discovered, and loosely coupled in novel ways”. Service-oriented computing (SOC) is a “service-oriented” approach to programming that is based on the idea of composing applications by discovering and invoking network-available services to accomplish some task, independent of specific programming languages and operating systems. Papazoglou et al. (2007) describes the SOC vision as “easily assemble application components into a loosely coupled network of services that can create dynamic business processes and agile applications that span organizations and computing platforms”.

17
In order to achieve this vision, SOA provides an approach to developing software system to provide services to applications or other services, distributed in a network through published and discovered interfaces. Web services are the most promising SOC-based technology that uses the Internet as communication medium and open standards such as: Web Services Description Language (WSDL) for describing Web services, Simple Object Access Protocol (SOAP) for transmitting data, and Universal Description Discovery and Integration (UUID) for service discovery.

Multiple services can be aggregated into a single composite service that can be used in further aggregations or consumed by client applications. This process is called service composition and it involves the roles and functionalities for aggregating multiple services into a single composite one (Papazoglou et al., 2007). Two important aspects of creating business process from services compositions are orchestration and choreography. Orchestration refers to an executable business process that can interact with internal and external services at the message level (Peltz, 2003). These interactions include business logic and execution order, and they can span applications and organizations resulting in a long-lived, transactional, multi-step process model (Papazoglou et al., 2007). Choreography refers to the message exchanges, rules of interaction, and agreements between multiple business processes (Papazoglou et al., 2007).

Orchestration represents one business party controlling the interactions of a business process. On the other hand, choreography represents tracking the message sequence among multiple parties, i.e. the protocol, instead of a specific business process executed by a single party (Peltz, 2003). Orchestration can be compared to a conductor controlling an orchestra, and choreography to a dance that defines how participants will interact with each other.
(Josuttis, 2007). Although there is this distinction between orchestration and choreography, there is a consensus that both should coalesce in a single language and environment since they are complementing concepts (Papazoglou et al., 2007).

Service orchestrations are specified by using a standard XML-based language called Business Process Execution Language (BPEL) for Web Services (OASIS, 2011). BPEL is also referred to as BPELWS, BPEL4WS and WS-BPEL. BPEL is a layer on the top of WSDL, and while the WSDL interface defines the specific operations allowed, BPEL defines how to sequence them (Peltz, 2003).

Milanović et al. (2009b) describe two approaches to modeling service choreographies: interaction models that are built of basic interactions, e.g. message exchanges, and interconnected interface behavior models that define control flows of each party of the interaction. Languages for interaction models include Web Services Choreography Description Language (WS-CDL), Let’s Dance and iBPMN. WS-CDL is a W3C initiative that consists of a declarative XML-based language used to describe the common and collaborative observable behavior of multiple services that need to interact in order to achieve some goal (Kavantzas et al., 2005). The Let’s Dance language mainly targets business analysts and comes with a graphical notation (Decker, Puhlmann, 2007). The iBPMN language is an extension of the Business Process Modeling Notation (BPMN), which is later discussed in this chapter, to support interaction modeling (Decker, Barros, 2008; Decker, Weske, 2011). Languages for interconnected interface behavior models include BPMN and BPEL4Chor. BPEL4Chor is an extension to the abstract BPEL for modeling choreographies (Decker, Puhlmann, 2007).
A more recent research area in SOA is Semantic Web Services. Semantic Web Services are part of the Semantic Web initiative that intends to make the Web more understandable by machines (Heflin and Hendler, 2001), by building an appropriate infrastructure for intelligent agents to run on the Web, performing complex actions on behalf of the users (Hendler, 2001). The Semantic Web requires the Web information to be represented in a standard way that is machine understandable, which is achieved using ontologies. An ontology can be defined as a specification of a conceptualization (Gruber, 1993), i.e. an abstract and simplified view of the world (Gašević et al., 2009). Semantic Web Languages are languages that provide support for leveraging Web-based ontologies in different tasks of the use of SOAs. The most relevant Semantic Web Languages include: Resource Description Framework (RDF) that provides a framework for representing metadata about Web resources; Web Ontology Language (OWL) that is used to define domain terms and their relationship in an ontology; and SPARQL, a query language for RDF, that is intended for querying Web data (Gašević et al., 2009). Semantic Web services augment traditional Web services through Semantic Web annotations to provide automatic service discovery, composition, invocation, monitoring, and interoperation (Payne and Lassila, 2004). The most relevant initiatives for Semantic Web Services include: OWL-S (W3C, 2004), an OWL-based ontology of Web services providing a core set of markup language constructs for describing the properties and capabilities of Web services (Gašević et al., 2009); Web Service Modeling Ontology (WSMO) (W3C, 2005a) that provides a conceptual framework and a formal language for semantically describing Web services; and WSDL-S (W3C, 2005b) that augments the expressivity of WSDL with semantic concepts (Akkiraju et al., 2005).
MDE Approaches for Service Oriented Architectures

In this section, we analyze MDE approaches for SOA by comparing their PIMs, PSMs, and transformation approaches. The following table contains a summary of some of these approaches.

Table 4. MDE Approaches for SOA

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>PIM</td>
<td>EDOC UML Profile</td>
<td>EDOC UML Profile</td>
<td>UML4SOA</td>
<td>UML-S</td>
</tr>
<tr>
<td>Transformation</td>
<td>ATL</td>
<td>Extended OCL 2.0</td>
<td>MDD4SOA</td>
<td>-</td>
</tr>
<tr>
<td>Intermediate Model</td>
<td>-</td>
<td>-</td>
<td>IOM</td>
<td>-</td>
</tr>
<tr>
<td>PSM</td>
<td>WSDL</td>
<td>WSDL</td>
<td>WSDL, BPEL</td>
<td>-</td>
</tr>
<tr>
<td>Implementation</td>
<td>Java</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Bezivin et al. (2004) analyze an MDA approach to Web services, by applying it on the illustrative example of a travel agency. In their first application, PIM is created using UML, and then transformed into PSMs using the Atlas Transformation Language (ATL). The target platforms for the PSMs are Java, WSDL and Java Web Service Developer Pack (JWSDP). Based on their experience in creating transformation rules, they mention some disadvantages for using UML profiles for creating PIMs such as large transformation rules, excessive usage of filters and instructions of flow control, and excessive string comparisons. However, manipulating metamodels based on MOF has a benefit of a simple and clear transformation definition. On a second application, another PIM is created using Enterprise
Distributed Object Computing (EDOC), a specification presenting metamodels and UML profiles for modeling enterprise applications. EDOC uses Enterprise Collaboration Architecture (ECA) patterns, meta-models and profiles. In this second application, EDOC-based PIM was transformed into PSMs in Java and WSDL. The authors concluded that EDOC provided a better representation of the functionality and behavior patterns for a distributed system. They also argue that the semantic gap between PIMs and PSMs can be decisive on the model transformation effort, and suggest the usage of intermediate metamodels to facilitate the metamodel mappings when their semantic distance is significant.

In a later work, Yu et al. (2006) describe a model driven framework based on UML profiles for EDOC, MDA and Web Services. EDOC UML profile extends UML1.4 to model component based enterprise computing systems. In this framework, PIMs are created using the EDOC UML profile, and then decomposed into sub-PIMs where each of them can provide services independently. Sub-PIMs are then transformed into interface models of its corresponding services, and also transformed into service implementation models on specific platforms. The authors argue that the most important part of the model transformation is the transformation rules part and not the transformation language. They define transformation rules using a slightly extended version of OCL, thus making these rules readable and independent of model transformation engines. This overcomes Bezivin et al. (2004) transformation rules issues when using UML profile-based models described in the previous paragraph.

Mayer et al. (2008) describe a model driven approach for SOAs called MDD4SOA (Model-Driven Development for SOA) consisting of an integrated UML2-based SOA modeling approach that includes a UML profile for SOA (UML4SOA) and model
transformation tools for generating code in various target languages. SOA structures are modeled using component and deployment diagrams, and UML4SOA adds new model elements to represent SOA static aspects: services, service descriptions and service interfaces. Service orchestrations are modeled by activity diagrams, which are extended with service-aware elements of UML4SOA (p. 205). The MDD4SOA approach also includes model transformations from MDD4SOA models into BPEL4WS and WSDL based on MDA principles. Their approach is different from previous approaches presented so far, by using an UML profile that is specific for SOA, rather than using the EDOC Profile that is for distributed applications, thus supporting SOA specific aspects. Their transformation approach also differentiates by transforming PIM into intermediate models, Intermediate Orchestration Model (IOM), to reduce the semantic gap between PIM and PSM as suggested by Bezivin et al. (2004).

Dumez et al. (2008) describe another model driven approach for SOA, which uses UML for Services (UML-S), an UML 2.0 profile. UML-S enables the modeling and specification of Web service interfaces with class diagrams, and Web service interactions with activity diagrams. UML-S class diagrams can be generated from Web service descriptions defined through WSDL, thus becoming a user-friendly view for representing Web service WSDL. UML-S conforms to MDE principles and UML-S models can be transformed into platform specific code using a given set of transformation rules. The authors provide WS-BPEL 2.0 transformation rules for almost all of the flow control patterns supported by UML-S: sequence, parallel split, synchronization, exclusive choice, simple merge, multi-choice, synchronizing merge, discriminator, N-out-of-M join, and while loop. However, they do not support the multiple-merge pattern, since it is not supported in WS-
BPEL 2.0. Similar to Mayer et al. (2008) approach, Dumez et al. (2008) approach also proposes an UML profile specific for SOA. However, they only provide transformation rules from UML-S to BPEL, and do not use intermediate model transformations as Mayer et al. (2008) approach does.

According to Wada et al. (2007), it is important in SOA to separate function and non-functional requirements (NFR) for services since this separation improves the reusability of services in different non-functional contexts. They propose a new MDE framework that consists of a feature model that defines non-functional constraints in SOA, a UML profile to specify SOA NFRs, called UP-SNFRs, and an MDE transformation tool called Ark. UP-SNFRs profile extends UML by adding two major concepts: a service that encapsulates the function of a certain element in a system, and a connection that defines how services are connected to each other and how messages are exchanged. A feature model consists of a set of all possible valid configurations of features describing different kinds of constraints which give a clear view of features that the system supports and their constraints. Ark accepts the application model (UML2 class or composite structure diagram) and feature configurations, and transforms them into an application model with NFRs (UML with UP-SNFRs). Then, this model is transformed into a skeleton of the application code, e.g. source code and deployment descriptor for Mule ESB, an open source enterprise bus. This approach defines metamodels using EMF Ecore, rather than MOF, and the model transformation tool uses fmp, a feature modeling tool implemented in EMF, and Eclipse UML2 to read and transform models.

Stein et al. (2008) evaluate the OrVia framework for an MDE approach to implement SOA. OrVia provides an iterative methodology supporting the realization of a business
process during the analysis, design and implementation phases. In OrVia, business process models are transformed into executable process models, and model checking technologies are applied to ensure the consistency of the model information on all levels. The authors were able to successfully apply the OrVia framework in a real world case in the e-Government domain. Although they identified a lack of integration between validation and structured requirements analysis, and no support for transformation of condition expressions, they still believe that OrViA is very useful for bringing model transformation technologies into the business context.

**Business Process Modeling**

The most promising research direction in modeling service compositions is using business process modeling (Pant and Juric, 2008). In this section, we briefly describe the most relevant approaches for business process modeling.

Weske (2007) defines business process as a set of activities performed in coordination in an organizational and technical environment to realize a business goal, where each business process acts on behalf of a single organization but it may interact with business processes of other organizations. The main artifact to implement a business process is business process model, which “consists of a set of activity models and execution constraints between them”. A business process model represents a blueprint of a set of business process instances with the same structure, whereas an activity model represents a blueprint of a set of similar activity instances.

Graml et al. (2007) describe some approaches for business process modeling:
• UML 2.0 activity diagrams (UML-AD) combined with activity sequences, related control logic and data flow can be used to model business process. By using Object Constraint Language (OCL), a declarative language for describing rules that applies to UML model, constraints and pre or post conditions could also be represented.

• Business Process Execution Language (WS-BPEL) (OASIS, 2011) is a language for representing business processes consisting of operations defined through Web Services (WS) interfaces. WS-BPEL defines control flow elements, that support conditional executions and loops definitions, and concurrent flows, that support an interleaved model of concurrency.

• Business Process Modeling Notation (BPMN) (OMG, 2011c) enables graphical editing of service-oriented business process models. It represents business processes as interactions between agents or process roles, graphically represented by swim lanes.

The Business Process Modeling Notation

Modeling of service compositions is primarily based on the BPMN language, as a standard business process modeling language (Milanović et al., 2009; OMG, 2011c). BPMN is the basis of our work, and here we provide an overview of this language.

BPMN is a recent approach for business process modeling which was adopted by Object Management Group (OMG) for standardization (Recker et al., 2006; OMG, 2011c). BPMN defines a business process diagram (BPD) based on a flowcharting technique, thus representing a business process model as a network of graphical objects containing activities
and flow control objects (White, 2004). There are four basic categories of graphical objects in BPMN: flow objects, connecting objects, swimlanes, and artifacts shown in Figure 3.

![Diagram of BPMN elements]

**Figure 3. Core modeling elements in BPMN.**

The three core flow objects are *Event*, *Activity* and *Gateway*. An *Event* represents something that happens during the process and can be categorized into three sub-types depending on when they occur: Start, Intermediate, and End. An *Activity* is a generic term for a work that the business performs and it can be atomic or non-atomic, e.g. compound. There are two types of activities: *Sub-Process* and *Task*. A *Gateway* element is used to control the sequence flow by defining decisions, forks, merges, and joins.

The connecting objects provide means to connect the flow objects, thus providing a basic structure of a business process. There are three types of connecting objects: *Sequence*
Flow, Message Flow, and Association. A Sequence Flow element shows the order of the activities performed in the process. A Message Flow element shows the flow of messages sent and received by two process participants. An Association element provides means to associate data, text and other artifacts with flow objects.

Swimlane is a mechanism to organize activities into separate categories to illustrate different functional capabilities or responsibilities. BPMN supports two types of swimlanes: Pool and Lane. A Pool represents a participant in a process and it is used when the BPD involves two separate business entities or participants. Figure 4 shows a BPD of sales and shipping of goods that contains two participants, Customer and Supplier, and each participant is placed on its own Pool. A Pool can be divided into Lanes, which are used to organize and categorize activities inside a Pool. Thus, Message Flow is used to connect Pools while Sequence Flow is used to connect activities within the Pool. In Figure 4, the activities of the Supplier pool are separated into sales and shipping activities. Thus, the Supplier pool has two lanes called Sales and Shipping. Note in Figure 4 that the Purchase Request activity of the Customer pool connects to activities of the Supplier pool with Message Flow elements, while activities within the same pool are connected with Sequence Flow elements.

![Diagram of Sales and Shipping processes (BPMN)](image)

Figure 4. Sales and Shipping processes (BPMN).
Artifacts provide the ability to extend the basic notation for a specific modeling situation, for example, a vertical market. BPMN defines three types of artifacts *Data Object*, *Group*, and *Annotation*. A *Data Object* element shows how data is required or produced by activities. A *Group* element does not affect the *Sequence Flow*, and it is used for documentation and analysis purposes. An *Annotation* element provides additional text information for the diagram reader.

**Business Process Based Modeling of SOAs**

The use of business process modeling as basis for service composition is the most promising research direction in developing service compositions (Pant and Juric, 2008). This section discusses business process based approaches for modeling SOAs: WebML, BPMN, and UML Activity Diagrams based.

WebML is a conceptual model for specifying data-intensive Web applications (Manolescu et al., 2005; Brambilla et al., 2006, 2007). A WebML specification consists of a data model, a standard Entity-Relationship (E-R) model, one or more hypertext models describing the application logic, and a presentation model describing the user interface of the pages. A WebML hypertext model consists of several site views, each view of a set of interrelated pages, and each page of WebML content units that are elementary components for information publishing (Manolescu et al., 2005). Manolescu et al. (2005) presents a WebML-based approach for designing and deploying Web applications using Web services. They adopted the WSDL standard for describing message exchange patterns and developed new WebML units to support WSDL within WebML. WebML data models are extended to include, in addition to application data, Web service metadata consisting of conversations,
operations and their relationship with the application data, and messages. These extensions enable WebML to represent Web service invocations, the relationships between invocations, and the data units providing service invocation inputs and outputs.

In a later work, Brambilla et al. (2006) extended WebML to support business process modeling. The WebML hypertext model is extended with elements to start and end activities, assign work items to activities, and retrieve application data relevant to activity execution. The data model is extended with process metadata that includes activity type, activity instance, process and case (process instance). These extensions enable WebML to support collaborative workflow based applications that spans multiple individuals, services, and organizations.

Brambilla et al. (2007) propose an MDE methodology to design and develop semantic Web service applications described according to the WSMO standard. Their MDE methodology starts with the design of the business process using BPMN. The design of the data model is done by using WebML data model (enhanced E-R), and ontologies are extracted by transforming the WebML data model into a WSMO compliant ontology. Process ontologies are extracted from the BPMN specification of the underlying business process of the application. Then, the BPMN model and WebML data model can be annotated with concepts from existing ontologies, possibly provided by third parties. The WebML language was enhanced to support queries on ontologies. The final step is the design of the service and user interfaces in WebML, where BPMN diagrams are transformed via XSLT into WebML hypertext models.

In the Service Oriented Architecture section of this chapter, we presented BPEL as a language for specifying service orchestrations. Although some BPEL execution platforms
provide graphical editing tools for defining BPEL processes, they provide a notation that reflects the underlying code, making modelers to reason in terms of BPEL constructs, thus resulting in a low level of abstraction that is not suitable for business process analysts and designers. They also use different graphical notations of BPEL and no standard has been adopted (Schumm et al., 2009). In terms of modeling service orchestrations, BPMN and UML Activity Diagrams (AD) are two typical approaches (Milanović et al., 2009). In both approaches, business process models representing service orchestrations are transformed into executable BPEL code. Zhang and Duan (2008) present some limitations of the transformation methods from AD process models into executable BPEL code: human intervention is needed to identify activity patterns, only handle partial subset of activity diagrams, and do not fully support block and graph structures of BPEL. Zhang and Duan (2008) suggest an automated approach to transforming AD process models into executable BPEL code that decomposes an AD model into regions, and for each region, structural patterns are identified, and then the relevant BPEL code is generated for it. Although their approach is fully automated, it is limited to support only basic control flow patterns: sequence, parallel split, synchronization, exclusive choice, and simple merge.

Transformation from BPMN models into BPEL is a complex task due to the fundamental difference between the two languages: BPMN is a graphical oriented language while BPEL is a mainly block-structured language (Ouyang et al., 2006). In an early work, White (2005) provides an example of how to map BPMN into BPEL, by using an example of a travel booking process, but this work only covered a few aspects of BPMN/BPEL mapping. Ouyang et al. (2006; 2009) presents an algorithm that generates BPEL code by discovering patterns in BPMN models that can be mapped into BPEL structured constructs. Their
approach only supports a subset of BPMN elements such as tasks, events, parallel and XOR gateways. In later work, Ouyang et al. (2007) enhanced the readability of the generated BPEL code by exploiting the block-structured constructs of BPEL. Their pattern-based translation algorithm identifies three patterns of BPMN model fragments: well-structured patterns, quasi-structured patterns, and flow-based patterns. They showed how model fragments identified by these patterns can be mapped into block-structured BPEL constructs.

For flow-based pattern model fragments, they needed some additional control links to capture dependencies between activities located in different blocks. Mendling et al. (2006) introduced different generic transformation strategies between graph-oriented process languages (e.g. BPMN) and block-oriented process languages (e.g. BPEL) and vice-versa. For transformations from BPMN to BPEL, they identified Element-Preservation, Element-Minimization, Structure-Identification, and Structure-Maximization strategies. Gong and Xiong (2009) introduce an approach that handle transformation from multiple BPMN processes containing interaction mismatches into BPEL processes. Interaction mismatch happens when an interaction can interrupt another interaction. They use an interaction mismatch discovery approach that supports both static and runtime interaction mismatches.

The transformation is an iterative process where the interaction mismatch discovery approach is applied to a pair of business processes, and these processes are composed into a single one. The interaction mismatch discovery approach is then applied to the composed process and the remaining process again. The mismatch discovery approach consists of the following steps: reduce process, compute complete mismatch set, and minimize complete mismatch set.

Although these approaches presented in this section provide promising research results in the context of business process based modeling of SOAs, Milanović et al. (2009)
identify some key challenges to be addressed: separation of business process models from
domain models and business vocabularies, resulting in the need to connect the business
vocabulary with the message types used in the processes; lack of formal language for
defining business conditions, e.g. executable condition based on business vocabulary; and
limited support for dynamic changes of business processes.

**Business Rules**

The Business Rules Group (2000) defines business rule as “a statement that defines or
constrains some aspect of the business. It is intended to assert business structure or to control
or influence the behavior of the business”. Business rules are a first-class citizen of the
requirement world, and are essential for business and technology models (Business Rules
Group, 2003). Wagner et al. (2005) describe the following categories of business rules
identified in the literature:

- **Integrity**: a condition that must hold to the state or behavior of a business
  process.
- **Production**: an action that is produced under certain conditions.
- **Derivation**: a statement of knowledge that is derived from other knowledge by
  inference or a mathematical calculation.
- **Reaction**: an action that must be taken in response to events (triggering, pre
  and post conditions). Reaction rules are also known as ECA (Event Condition
  Action) rules.

Milanović and Gašević (2009) describe the two major areas in the context of business
process modeling and SOAs: *rule interchange* and *rule modeling*. The Rule Interchange
Format (RIF) is a W3C Recommendation being the standard format for rule interchange (W3C, 2010). Other two initiatives in the rule interchange domain are RuleML (RuleML, 2011) and REWERSE Rule Markup Language (R2ML) (R2ML, 2006). Both initiatives support all four business rule categories described above. In the rule modeling domain, OMG developed two initiatives: the Semantics of Business Vocabulary and Rules (SBVR) (OMG, 2008) and the Production Rule Representation (PRR) (OMG, 2009). The SBVR defines a structured English vocabulary, and the PRR provides a high level modeling of a common production rule implementation. Although R2ML was originally designed to support rule interchange, it is also a comprehensive rule modeling language, built completely by using MDE principles. Thus, R2ML is considered a general rule language while the other abovementioned languages are used only either for rule interchange (RIF and RuleML) or for representing a specific type of rules (SBVR and PRR).

**R2ML**

In this section, we describe the core elements of the R2ML that will be used in this project, given that rBPMN builds upon BPMN and R2ML. On the top level of the R2ML metamodel, the *RuleBase* class is composed of multiple rule sets (*RuleSet* abstract class) and vocabularies as indicated in Figure 5. There is a concrete rule set class for each type of rule containing zero-to-many rules of that specific type.
Integrity rule is associated to a constraint (LogicalFormula abstract class), and it has two subclasses: AlethicIntegrityRule for constraints expressing the necessity, possibility, and impossibility, and DeonticIntegrityRule for constraints expressing obligativity, permission, and prohibition (R2ML, 2006). A derivation rule is associated to many conditions (AndOrNafNegFormula) and to one conclusion (LiteralConjunction). A production rule is associated to one or more conditions, but it is also associated to many post conditions, and it produces one-to-many actions (ProgramActionExpression). A reaction rule is composed of many conditions and may have one post condition. It is also composed of one triggering event and to one triggered event (EventExpression). Rules conditions and post conditions are
instances of AndOrNafNegFormula, the base class of atoms. Rules conclusions are instances of LiteralConjunction which are composed of one or more atoms. Figure 6 shows all concrete atom classes that can be used to define rules conditions and conclusions.

Figure 6. Atoms in the R2ML metamodel. Adapted from R2ML (2006).

Figure 7 presents all subclasses of EventExpression. Events can be categorized in a single event, i.e. AtomicEventExpression, or complex events, i.e. events that have a composition relationship with EventExpression. Complex events are:

SequenceEventExpression that is composed of two or more events that occur in a particular order; ParallelEventExpression that is composed of two or more events that occur at the same time; ChoiceEventExpression that is composed of two or more events that are mutually exclusive, i.e. only one of them will occur; and AndNotEventExpression that is composed of two events where the first event occurs but the second does not.
URML

R2ML has a graphical concrete syntax called UML-Based Rule Modeling Based Language (URML). URML extends the UML metamodel with the concept of rule, thus allowing visual rule modeling based on UML class models (Likichev, Jamar, 2009). URML supports derivation, production and reaction rules. In URML, rules are represented as a circle with an internal label describing the rule type and a rule identifier attached to it as indicated in Figure 8. The internal label is "DR" for derivation rule, "PR" for production rule, and "RR" for reaction rule.

Figure 8. Rule elements in URML.

Rule conditions are represented as incoming arrows to the rule element while rule conclusions as outgoing arrows. URML has three types of conditions (classification, role, and association) that are used by all supported rules, and four types of conclusions (classification, role, attribution, and association) that are used by derivation rules only.
Figure 9 shows a derivation rule in URML that describes the following business rule: "If the first airport of the trip is not the same as the last airport, then it is a one way trip". The association condition, e.g. the blue arrow from the Trip class to the derivation rule, contains two annotations: a filter expression "firstAirport<>lastAirport" that filters out instances of the condition classifier Trip, and a rule variable x. The derivation rule conclusion is represented by the red arrow from the rule element to the OneWayTrip class. In this case, it is a classification conclusion.

![Figure 9. Derivation rule in URML.](image)

Production and reaction rules support post conditions and actions. Post conditions are represented similar to conditions, e.g. outgoing arrow from the rule element, but with a double arrowhead. Actions are also represented as outgoing arrows with double arrowhead, but with an additional label indicating the type of action: "A" for assert, "R" for retract, "U" for update, and "I" for invoke.

Figure 10 shows a production rule in URML that describes the following business rule: "If the weather at the airport is overcast, then the status of all flights should change to delayed". This production rule updates the attribute status of Flight instances to "delayed" if two conditions are met. The update action is indicated by the green double arrowhead from
the production rule to the *Flight* class. The first condition is a classification condition, a blue arrow from the *Airport* class to the production rule, filtering instances of *Airport* with weather attribute set to "overcast". The second condition is an association condition, the blue arrow from the *hasFlights* association to the production rule, filtering instances of *Flight* that are associated with *Airport* instances filtered by the first condition.

![Production rule in URML.](image)

Reaction rules have another kind of incoming arrows that represent events, but it uses a bold arrowhead to be visually distinguished from condition arrows.

**Mappings Between URML and R2ML XML**

Lukichev and Wagner (2006) describe a mapping between URML metamodel and R2ML XML. Their work is focused to provide these mappings for the Strelka tool, an URML modeling editor (Strelka, 2006), to perform model-to-text transformation from URML models into R2ML XML so that their models can be deployed to rules engines. They defined a compositional mapping for URML metamodel into R2ML XML markup that supports derivation, production and reaction rules, plus conditions, conclusions, filters, and actions. Their work does not include events since the reaction rules metamodel for URML and R2ML was still in progress at that time. The following table has a summary of their mappings (filters are not included).
Table 5. URML and R2ML XML Mappings.

<table>
<thead>
<tr>
<th>URML Metamodel</th>
<th>R2ML XML Metamodel</th>
</tr>
</thead>
<tbody>
<tr>
<td>DerivationRule</td>
<td>DerivationRule</td>
</tr>
<tr>
<td>ProductionRule</td>
<td>ProductionRule</td>
</tr>
<tr>
<td>ReactionRule</td>
<td>ReactionRule</td>
</tr>
<tr>
<td>ClassificationCondition</td>
<td>ObjectClassificationAtom</td>
</tr>
<tr>
<td>AssociationCondition</td>
<td>AssociationAtom</td>
</tr>
<tr>
<td>RoleCondition</td>
<td>ReferencePropertyAtom</td>
</tr>
<tr>
<td>ClassificationConclusion</td>
<td>ObjectClassificationAtom</td>
</tr>
<tr>
<td>AssociationConclusion</td>
<td>AssociationAtom</td>
</tr>
<tr>
<td>RoleConclusion</td>
<td>ReferencePropertyAtom</td>
</tr>
<tr>
<td>AttributionConclusion</td>
<td>AttributionAtom</td>
</tr>
<tr>
<td>AssignAction</td>
<td>AssignActionExpression</td>
</tr>
<tr>
<td>CreateAction</td>
<td>CreateActionExpression</td>
</tr>
<tr>
<td>DeleteAction</td>
<td>DeleteActionExpression</td>
</tr>
<tr>
<td>InvokeAction</td>
<td>InvokeActionExpression</td>
</tr>
</tbody>
</table>

Note. Lukichev and Wagner (2006) mapping is based on earlier versions of R2ML metamodel. As of R2ML v0.5, assign, create, and delete actions were replaced by update, assert, and retract actions.

**Mappings Between R2ML and R2ML XML**

Milanović et al. (2009c) defines the mappings rules between R2ML concrete syntax (R2ML XML schema) and R2ML abstract syntax (R2ML metamodel):
• Each metamodel class is mapped to an XML element and a complexType in the XML schema. The names of the XML element and complexType are the same as the metamodel class name.

• If the metamodel class is abstract, then the corresponding XML element is also abstract.

• Each data type attribute of a metamodel class is mapped to an XML attribute of the corresponding XML element.

• MOF association and composite association are mapped respectively to an XML attribute and to an XML element that are part of the content model of the XML complexType generated from the class referencing this association.

**Business Rules and Business Process Models Integration**

BPMN has become a standard for business process modeling being widely adopted in the industry. A representation analysis of BPMN concluded that there is no representation for state in BPMN, thus the representation of “business rules that rely on state and transformation laws will be unclear” (Recker et al., 2006). Recker et al. conducted interviews with BPMN practitioners from Australian organizations and government agencies, resulting in 63% of the participants having the need for representing business rules in BPMN diagrams. The usage of business process models is a common approach in the development of SOAs, but they are too rigid for dynamic adaptations of the business logic (Milanović, Gašević, 2009, p. 64). For the reasons abovementioned, the integration of business rules into business process models has become an important research topic.
According to Milanović and Gašević (2009), there are two major research categories of integration of business processes and rules: fully rule-based approaches and integration of business rules into process-oriented models (hybrid approaches). The fully rule-based approaches aim to model business processes fully by using business rules, usually done with production and reaction rules. The hybrid approaches propose methods for integration of business rules and business process modeling languages. Some issues of fully rule-based approaches include: difficult visualization of the overall business process and its constitutive parts relationships due to the level of details added by business rules; unexpected behavior hard to determine upfront can occur due to the business process execution being fully driven by reasoning algorithms; lack of effective and unified support for different rules types, and rules are typically represented in implementation languages which are not suitable to be used in business process modeling languages. The remaining part of this section discusses the most relevant approaches to integrate business rules into business process models.

An early work on the integration of business rules and business processes is the approach presented by Krogstie et al. (1991). In their approach, they start from a high level process modeled using data flow diagrams (DFD), and successively decompose it into lower level processes until the lowest level of decomposition is achieved, i.e. the process logic, and then use the External Rule Language (ERL) to describe the process logic. They proposed coupling the process and the rule models by defining constructs in one model that indicates the existence of constructs in the other model. Following their work, McBrien and Seltveit (1995) presented a technique to couple a process modeling language called PID with ERL to support certain types of business rules (integrity, derivation, and reaction). Knolmayer et al. (2000) describe a fully rule-based approach for modeling business processes and workflows.
Their approach supports only reaction rules, through an extended version of the ECAA (Event - Condition - Action - Alternate Action) notation. The ECAA notation adds an alternative action that is taken when the condition is false. The early approaches presented so far, do not support software modeling standards and provide integration at the graphical syntax level only. Other approaches that provide integration at different levels (hard-coded, metamodel based, Web service based) are presented as follow.

In addition to support integration at the graphical syntax level, Meng et al. (2002) approach also support it at the metamodel level. In their approach, a dynamic workflow model (DWM) extends the Workflow Process Definition Language (WPDL) model by adding new modeling constructs (connectors, events, triggers and rules) to enable the integration of rules with processes.

Charfi and Mezini (2004) proposed a hybrid service composition approach that combines business processes with rules, using aspect-oriented programming (AOP) to encapsulate rules into aspect modules. To achieve this, they implemented AO4BPEL, an aspect-oriented dialect of BPEL that supports the implementation of integrity, derivation and reaction rules. In their implementation, business rules are physically separated from business process, where each business rule is mapped to an aspect in AO4BPEL. Business rules can then be dynamically activated and deactivated at process interpretation time, since AO4BPEL supports dynamic weaving. Their approach lacks of software modeling standards, provides no support for production rules, and the integration is hard-coded into the implemented language (Milanović and Gašević, 2009).

Rosenberg and Dustdar (2005) propose an approach to integrating rule-based knowledge that is accessible through business rules engines, into BPEL in a service-oriented
way. Their approach uses an Enterprise Service Bus (ESB) as an integration platform. In the ESB, there is a BPEL engine and a Web service gateway connected. The BPEL connects through an adapter and sends messages through the ESB to the Web service gateway, which calls external services or waits for their responses. A business rule broker service provides a plug-in based mechanism to integrate different business rules engines. Rule interceptor service is the bridge between business rules and executable BPEL process, where two types of interceptors are executed before and after the BPEL activity. A transformation engine is used to transform XML messages to other formats used by business rules engines. Their approach lacks of modeling standards, provides no support for production rules, and the integration is Web service based and hard-coded into the implemented language (Milanović and Gašević, 2009).

Orriëns and Yang (2006) propose a framework for designing business collaborations called Business Collaboration Design Framework (BCDF) based on a rule-based approach. Their approach integrates RuleML into BCDF models at the graphical and textual syntax levels, and does not use business process modeling standards.

Bry et al. (2006) propose a fully rule-based approach that uses the XChange rule language for describing business process with reaction rules in an executable manner. According to them, some of the benefits of a reaction rule based approach for specifying business processes include: business process requirements usually expressed as reaction rules, easy integration with other rules commonly used in business applications, flexibility and easy error and exception handling of reaction rules. However, there are some limitations: reaction rules do not always reflect the familiar procedural and imperative way of thinking of people from imperative and object oriented programming, reaction rules usually do not have
a local state that is specific and internal to the current process instance, monitoring of business processes based on reaction rules is not straight forward as for BPMN or BPEL, there is no clear notion of which events are expected next, no support for graphical representation of reaction rules in BPMN. Their approach only support reaction rules and does not provide a graphical representation of a process model or use software modeling standards (Milanović and Gašević, 2009).

Recent hybrid approaches support software modeling standards, such as BPMN, and they include: Goedertier and Vanthienen (2006), Graml et al. (2007), and Eijndhoven et al. (2008).

Goedertier and Vanthienen (2006), as cited by Milanović and Gašević (2009), is the only approach presented in this section that supports all four types of business rules. They developed EM-BRACE, a framework for business process modeling based on business rules that uses hard-coded rules in the PENELope (Process Entailment from the Elicitation of Obligations and Permissions) language.

Graml et al. (2007) present three groups of patterns for integration of business rules in business process driven development of SOAs: control flow decisions, data constraints, and process compositions. Their approach enables modifying control flow, data flow and activities of business processes at runtime, resulting in an agile business processes. They implemented these patterns with standard BPEL process automation technology and a simple rule based service component. Their approach does not provide integration of any formally defined rule language and it uses natural language for defining rules instead (Milanović and Gašević, 2009).
Eijndhoven et al. (2008) propose a rule-based approach to improving change management and maintainability of business process by using reaction rules and workflow patterns to model the variable parts of the process. Their method consists of three steps: identify variable and non-variable segments in the process, identify an appropriate combination of workflow patterns to model the behavior of each variant in a variation point, and implement workflow patterns using business rules. In their approach, business processes are defined with BPMN language, and rules with iLOG JRules. Their approach only supports reaction rules and it is primarily related to implementation of business processes (Milanović and Gašević, 2009).

According to Iacob and Jonkers (2008), most orchestration languages have significant limitations in supporting business rules, and there are two options to solve this shortcoming: integrating existing orchestration languages and tools with business rules languages and extend existing orchestration languages with rules specification constructs. They propose an MDE framework for rule-based design of SOA. They incorporate business rules in the MDE process in order to raise the level of abstraction at which business logic is integrated in application design in the context of SOA. Their proposal integrates design models and rule specifications in horizontal model merging transformations between design models and rules specifications for each abstraction level (architectural, PIM, and PSM).

Other approaches also include distributed and agent-oriented business rules. VIDRE is a distributed service-oriented business rule engine proposed by Rosenberg et al. (2006) that facilitates reuse of business logic, by enabling business processes and enterprise applications to access business rules accessed as Web services. Their solution introduces distributed business rule execution that removes the limitation of a centralized execution of business
rules, and provides a plug-in mechanism to use arbitrary rule engines. Ali et al. (2006) propose an agent-oriented framework to automate the inclusion of business rules into the business process. The proposed agent framework uses an agent to communicate between the business process component and the business rule component, and it applies business logic defined by a business actor. The rule agent keeps track of new rule changes, updating new rules in the business process, and addition of new business rules objects. Business analysts or users define business processes and rules using a form entry user interface, and they are stored in the system in XML format. The agent-oriented framework makes business rules more customizable and understandable, it reduces complexity, and it applies business rules more effectively.

In this section, we reviewed the most relevant approaches to integrate business rules in business process. The following table summarizes these approaches, showing the languages used for business processes and rules, type of integration, supported rules and standards used.

<table>
<thead>
<tr>
<th>Approach</th>
<th>Process Language</th>
<th>Rule Language</th>
<th>Integration</th>
<th>Rule Type</th>
<th>Standards</th>
</tr>
</thead>
<tbody>
<tr>
<td>Krogstie et al. (1991)</td>
<td>DFD extended</td>
<td>ERL</td>
<td>Graphical syntax</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>McBrien and Seltveit (1995)</td>
<td>PID</td>
<td>ERL</td>
<td>Graphical syntax</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Knolmayer et al. (2002)</td>
<td>ECAA extended</td>
<td>ECA</td>
<td>Graphical syntax</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Meng et al. (2002)</td>
<td>WPDL</td>
<td>Graphical syntax and metamodel</td>
<td>-</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td>Authors</td>
<td>Language</td>
<td>Modeling Language</td>
<td>Type</td>
<td>Support</td>
<td>Support</td>
</tr>
<tr>
<td>------------------</td>
<td>----------</td>
<td>-------------------</td>
<td>-----------------------</td>
<td>---------</td>
<td>---------</td>
</tr>
<tr>
<td>Charfi and Menziní (2004)</td>
<td>BPEL</td>
<td>AO4BPEL</td>
<td>Hard-coded</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Rosenberg and Dustdar (2005)</td>
<td>BPEL</td>
<td>Rules engines</td>
<td>Web service and hard-coded</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Orriëns and Yang (2006)</td>
<td>BCDF</td>
<td>RuleML</td>
<td>Graphical and textual syntaxes</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Bry et al. (2006)</td>
<td>XChange</td>
<td>XChange</td>
<td>Hard-coded</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Graml et al. (2007)</td>
<td>BPEL</td>
<td>OOP language</td>
<td>Graphical syntax and hard-coded</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Eijndhoven et al. (2008)</td>
<td>BPMN</td>
<td>iLOG JRules</td>
<td>Model Transformation</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Rosenberg et al. (2006)</td>
<td>BPEL</td>
<td>RuleML</td>
<td>Web service and hard-coded</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Ali et al. (2006)</td>
<td>XML</td>
<td>XML</td>
<td>Agent oriented and hard-coded</td>
<td>+</td>
<td>-</td>
</tr>
</tbody>
</table>

Note. Rule types are rated with “+” if the approach supports it, and “-” if it is not supported.

Adapted from Milanović and Gašević (2009).

The Rule-enhanced Business Process Modeling Language

Milanović and Gašević (2009) propose a new modeling language that integrates both rule and process-oriented modeling. The new modeling language is called Rule-enhanced Business Process Modeling Notation (rBPMN). This section describes the rBPMN language, business process patterns in rBPMN, and a methodology for developing rule-enable SOA using rBPMN.
The rBPMN language is a result of integration of the BPMN and R2ML languages at their metamodel levels (Milanović and Gašević, 2009). The selection of BPMN was based on its broad user adoption, comprehensiveness in covering business process concepts, and user rich experience. The selection of R2ML was based on its support for both rule modeling and rule interchange, support for all type of rules, proven and rich rule modeling language, previous experience in integrating with software modeling languages, and the goal of using MDE principles.

Milanović and Gašević (2009) defined the rBPMN metamodel by importing the elements from the BPMN and R2ML metamodels. The BPMN2 metamodel contains 74 classes, R2ML metamodel contains 124, and rBPMN adds 7 new classes with OCL constraints. The rBPMN metamodel contains a new element RuleGateway, which is an extension of the BPMN Gateway element and refers to one or more R2ML Rules (see Figure 11). Like the BPMN Gateway, the Rule Gateway is used to control the sequence flow, but it provides the usage of rules in process models.

![Figure 11. rBPMN Rule Gateway element.](image)

rBPMN extends the BPMN Activity package by supporting BPMN Tasks to be triggering (R2MLTrigerringTask class) or triggered (R2MLTriggeredTask class), and supporting sub-processes as production rule actions for R2ML reaction and production rules.
(R2MLTriggeredSubProcess class). rBPMN provides ways to connect underlying data models to business rules, by representing any BPMN message with an R2ML AtomicEventExpression. This is accomplished by extending the BPMN StructureDefinition class, which specifies the message structure, with a new subclass R2MLMessageType. The rBPMN R2MLMessageType class can have one R2ML AtomicEventExpression, thus allowing rBPMN messages to be directly mapped to triggering or triggered event expressions of R2ML ReactionRules.

In addition to elements that connect BPMN to R2ML, rBPMN also defines constraints, in the form of integrity constraints such as OCL invariants, to guarantee a consistent metamodel. The rBPMN metamodel enriched with these constraints define a set of well-formedness rules, and rBPMN well-formed models are those who are fully compliant with these rules (Milanović et al., 2009c). An example of an rBPMN metamodel constraint is: “When a rule gateway has a directly preceding event in its sequence flow in a process model, the rule gateway must have at least one reaction rule attached to it” (Milanović and Gašević, 2009). This constraint is represented in OCL as:

```ocl
class RuleGateway
inv: let sequenceFlow : SequenceFlow.allInstances() -> select(c | c.targetRef = this) -> asSequence() -> first()
  in sequenceFlow.sourceRef.oclIsTypeOf(Event)
  implies
  this.rule -> exists(e | e.oclIsKindOf(ReactionRule))
```

rBPMN can be used for modeling of SOAs, and its use in that context has been investigated through its support for modeling of message exchange patterns, service interaction patterns, workflow patterns, and business rules patterns. In the following sections, we describe the rBPMN support for these patterns.
Message Exchange Patterns in rBPMN

Message Exchange Patterns (MEP) define type and order of messages exchanged between services and service requestors (Milanović and Gašević, 2009). MEPs can be categorized into inbound patterns, initiated by service requestors through an incoming message, or outbound patterns, initiated by the service itself through an outgoing message. The following MEPs are supported by rBPMN:

- **In-Only**: consists of one message received by a service from the service requestor, where no fault message may be generated. A Rule Gateway models the service as a reaction rule where the input message is the triggering event, rule condition is the condition to execute the service and the service operation is the triggered action.

- **Robust In-Only**: a variation of in-only MEP where faults can be triggered by the message. rBPMN introduces an Exception handler sub-process, which is used to send the fault message to the service requestor.

- **In-Out**: represents request/response messages being exchanged by the service and service requestor. In this particular MEP, the triggered event part of the rule is a sequence of an Action representing the service operation, and an output message.

- **In-Optional-Out**: similar to In-Out, but the response message is optional. A second Gateway is added containing a condition to decide if a response message should be sent.

- **Out-Only**: only one message is sent by the service, usually used for message notification, where no fault message may be generated.
- Robust Out-Only: a variation of Out-Only where fault message may be generated. The service requestor uses a Rule Gateway to decide to return or not a fault message to the service based on its internal logic.
- Out-In: consists of two messages, but unlike In-Out, it is the service who initiates the message exchange.
- Out-Optional-In: similar to Out-In, but the second message which is received by the service is optional. It is also similar to In-Optional-Out, but in the opposite direction.

Service Interaction Patterns in rBPMN

Service interaction patterns (SIP) are used for modeling service choreographies (Barros et al., 2005, 2007). Service choreography tracks the message sequences between parties and sources when creating business process from composite Web services (Peltz, 2003). The rBPMN language supports both approaches for modeling choreographies: interaction models and interconnected interface behavior models.

rBPMN adds to the standard BPMN support to capture several choreography aspects: multiplicity of participants, references, and correlation information (Milanović et al., 2009b; Milanović & Gašević, 2010). Multiplicity of participants refers to the need to model multiple participants of the same type (Pool). rBPMN adds the “Multiple-instance participant” marker to distinguish multiple participants from each other in the same pool, see the symbol at the bottom of Pool 2 in Figure 12. The references aspect refers to the need to distinguish one participant from others, in order to know which participant executed some action in a process. rBPMN introduces the concept of participant set, which contains zero or more references to participants. In Figure 12, the participant set is the element using the “<par>”
annotation. In this example, each participant of Pool 1 participant set will perform the Send 1 task, e.g. send a message, to multiple instance participants of Pool 2.

Correlation information refers to the need to identify which participant sent a message, so that the response message can be sent back to that participant. rBPMN introduces an association from the message to participant. Figure 13 shows a participant set associated with message flows to identify the participants who sent the message.

Figure 12. rBPMN One-to-Many Send pattern.

Figure 13. Passing a participant reference over the message flow in rBPMN.
Milanović (2010) describes the SIPs supported by rBPMN. Table 7 shows all SIPs categorized into four groups, and the business process modeling languages that support each pattern (Milanović & Gašević, 2010). The rBPMN language fully support eleven of the thirteen SIPS, it partially supports the Dynamic Routing pattern by using rule gateways on data contained in the original request or in one of the intermediate steps (Milanović & Gašević, 2010), and it does not support the Atomic Multicast Notification pattern (no other language does). It is important to note that rBPMN is the only language that models the Contingent Requests pattern (Milanović et al., 2009b; Milanović & Gašević, 2010). Thus, rBPMN provides better support of SIPs in comparison to other languages.

Table 7. Service interaction patterns support of business process modeling languages.

<table>
<thead>
<tr>
<th>Group</th>
<th>Pattern</th>
<th>Let’s Dance</th>
<th>BPMN</th>
<th>WS-CDL</th>
<th>iBPMN</th>
<th>rBPMN</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single-transmission bilateral</td>
<td>Send</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td></td>
<td>Receive</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td></td>
<td>Send/Receive</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Single-transmission multilateral</td>
<td>Racing Incoming Messages</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td></td>
<td>One-to-Many Send</td>
<td>+</td>
<td>-</td>
<td>+/-</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td></td>
<td>One-from-Many Receive</td>
<td>+</td>
<td>-</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td></td>
<td>One-to-Many Send/Receive</td>
<td>+</td>
<td>-</td>
<td>+/-</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Multi-transmission</td>
<td>Multi-Responses</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td></td>
<td>Contingent Requests</td>
<td>+/-</td>
<td>-</td>
<td>+/-</td>
<td>+/-</td>
<td>+</td>
</tr>
</tbody>
</table>
Atomic Multicast Notification - - - - - -
Routing Request with Referral + - + + +
Relayed Request + - + + +
Dynamic Routing - - +/- - +/-

Note. Patterns are rated with “+” if the language directly supports it, with “+/-” if not supported but it can be worked around, and “-” if it is not supported. Adapted from Milanović and Gašević (2010).

Workflow Patterns in rBPMN

Workflow patterns are used for modeling service orchestrations. Service orchestration refers to an executable business process that can interact with internal and external services, thus representing the one party perspective of service compositions (Peltz, 2003). Although BPEL is a standard language for Web service orchestrations, it lacks of support for integration of business rules (Charfi, Mezini, 2004). BPMN and UML activity diagrams are typical approaches for modeling service orchestrations, and BPMN currently provides ways to represent workflow patterns (Milanović et al., 2009). However, rBPMN represents workflow patterns by using rules in addition to business process models, thus producing richer models by precisely describing business process in a declarative way, and by allowing only changing rules without redesigning the whole process during design or real time.

Workflow patterns consist of 21 patterns that can be categorized into six groups. Table 8 compares workflow pattern support of rBPMN with other workflow and business process modeling languages, and shows that rBPMN increases the support level for workflow patterns, especially on advanced branching and synchronization, multiple instances, and state based patterns (Milanović et al., 2009).
Table 8. Workflow patterns support of business process modeling languages.

<table>
<thead>
<tr>
<th>Group</th>
<th>Pattern</th>
<th>Business process modeling language</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>UML</td>
</tr>
<tr>
<td>Basic control-flow</td>
<td>Sequence</td>
<td>+</td>
</tr>
<tr>
<td></td>
<td>Parallel Split</td>
<td>+</td>
</tr>
<tr>
<td></td>
<td>Synchronization</td>
<td>+</td>
</tr>
<tr>
<td></td>
<td>Exclusive Choice</td>
<td>+</td>
</tr>
<tr>
<td></td>
<td>Simple Merge</td>
<td>+</td>
</tr>
<tr>
<td>Advanced branching and synchronization</td>
<td>Multi-choice</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>Multi-Merge</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>Discriminator</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>Synchronizing Merge</td>
<td>-</td>
</tr>
<tr>
<td>Structural</td>
<td>Arbitrary Cycles</td>
<td>+</td>
</tr>
<tr>
<td></td>
<td>Implicit Termination</td>
<td>+</td>
</tr>
<tr>
<td>Multiple-Instances (MI)</td>
<td>MI without Synchronization</td>
<td>+</td>
</tr>
<tr>
<td></td>
<td>MI with a Priori Design Time Knowledge</td>
<td>+</td>
</tr>
<tr>
<td></td>
<td>MI with a Priori Runtime Knowledge</td>
<td>+</td>
</tr>
<tr>
<td></td>
<td>MI without a Priori Runtime Knowledge</td>
<td>-</td>
</tr>
<tr>
<td>State-based</td>
<td>Deferred Choice</td>
<td>+</td>
</tr>
<tr>
<td></td>
<td>Interleaved Parallel Routing</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>Milestone</td>
<td>-</td>
</tr>
</tbody>
</table>
Patterns are rated with “+” if the language directly supports it, with “+-” if not supported but it can be worked around, and “-” if it is not supported. Adapted from Milanović et al. (2009, p. 12).

**Business Rules Patterns for Agile Business Process in rBPMN**

The business rules patterns for agile business process defined by Graml et al. (2007) (Business Rules and Business Process Models Integration section) are also supported by rBPMN. Milanović et al. (2011) describes modeling these 9 business rules patterns using rBPMN language, and the following table summarizes them.

Table 9. Business Rules Patterns for Agile Business Process in rBPMN

<table>
<thead>
<tr>
<th>Pattern Group</th>
<th>Pattern</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Control flow</td>
<td>Decision logic abstraction</td>
<td>Models decisions using DR and PRs external to the business process.</td>
</tr>
<tr>
<td>decisions</td>
<td>Decision node to business rule binding</td>
<td>Uses mapping to dynamically bind decisions to DR and PRs using their id property.</td>
</tr>
<tr>
<td></td>
<td>Decision with flexible input data</td>
<td>Allows DR and PRs to access process data using a process context, thus allowing dynamic input data changes during runtime.</td>
</tr>
<tr>
<td></td>
<td>Decision flexible output</td>
<td>Using PRs and its output data, it enables dynamically assignment of activities to decision output branches.</td>
</tr>
<tr>
<td>Data constraints</td>
<td>Constraints at predefined checkpoint</td>
<td>Uses IRs to check constraints validity on activity input and output data.</td>
</tr>
<tr>
<td></td>
<td>Constraints at multiple checkpoints</td>
<td>Extends previous pattern by checking constraints validity in multiple places of the process.</td>
</tr>
<tr>
<td></td>
<td>Constraints enforced by</td>
<td>Uses IRs and PRs to externalize business process</td>
</tr>
</tbody>
</table>
external data context data thus allowing data changes being perceivable by external party.

<table>
<thead>
<tr>
<th>Dynamic Business Process Composition</th>
<th>Business rule-based subprocess selection</th>
<th>Uses PRs to dynamically select specific parts of a business process.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Business Process Composition</td>
<td>Business rule-based process composition</td>
<td>Uses DRs and PRs to dynamically compose a business process from business process fragments.</td>
</tr>
</tbody>
</table>

**Note.** Business rules are referred as derivation (DR), production (PR), and integrity (IR).

**Methodology for Developing Rule-enabled SOA**

Milanović (2010) defines a methodology for developing rule-enabled SOAs using the rBPMN language. This methodology is based on MDE and consists of 9 phases as shown on Figure 14.

![Figure 14. Methodology for developing rule-enabled SOAs (Milanović, 2010).](image)

Here, we briefly describe these phases. Phases 1 to 4 will be used in Chapter V to model the case studies developed to evaluate the rBPMN Editor.
• **Requirements specification** is the initial phase where a business analyst gathers information about the application domain and business functions, and produces a project requirements document.

• **Process design** is the phase where a process modeler defines a general business process for the application. This phase is divided into the following steps:
  
  o Define an initial abstract business process with individual activities in BPMN, based on the requirements specification.
  
  o Identify activities that represent points of variability of the business process. Variable activities are activities that cannot be directly executed and need to be further specified.
  
  o Estimate the level of variability based on the frequency of changes, implementation responsibility, understanding of implications, source of change, and scope.
  
  o Determine if variability points will be implemented using business rules based on the level of variability estimated.
  
  o Identify the appropriated business process patterns to be used. Workflow patterns are used to model concrete variability points, and service interaction patterns to model choreographies.

• **Data design** consists of defining a domain model by using the information collected during the requirements specification.

• **Rule and policy design** is the phase where rules and policies are added to the business process.
• *Orchestration and choreography generation* consists of using a model transformation approach to generate executable orchestrations (e.g. BPEL) or choreographies (e.g. WS-CDL) from the rule-based business process.

• *Implementation* phase consists of implementation generation activities such as choreography executability on a specific platform, UDDI registration and location, generation of support Java application, etc.

• *Operation* phase consists of the execution of business process on the concrete platform.

In previous sections, we described the rBPMN language and its usage in message exchange, service interaction, and workflow patterns. The rBPMN language is the basis of our work to implement a modeling editor that realizes the rBPMN language, which is described in the following chapters. The rule-enabled SOA development methodology presented here is used in Chapter V to develop case studies for the rBPMN Editor.
CHAPTER III
METHODOLOGY AND DESIGN

As described in Chapter II, rBPMN is a new modeling language, a result of the integration of BPMN and R2ML languages through MDE principles, conceptually defined by Milanović and Gašević (2009). The rBPMN language definition consists of an abstract syntax and a graphical concrete syntax. The abstract syntax of rBPMN is a metamodel defined in MOF language, which is the result of the integration of BPMN and R2ML metamodels. This kind of integration is known as model weaving as explained in Chapter 2. The rBPMN graphical concrete syntax is a combination of BPMN graphical notation and URML. Milanović and Gašević (2009) have not formally defined a textual concrete syntax for rBPMN; however a textual concrete syntax is also be needed for exchanging models.

Currently, there is no tooling support for rBPMN that allows users to create models and perform transformations. The lack of tooling support limits the use and evaluation of the feasibility, usability, and effectiveness of rBPMN. WebML, for example, has the support of WebRatio, an MDE environment that provides modeling editor and automatic generation of Web applications, thus allowing this language to be evaluated according to the criteria abovementioned. The goal of this project is to overcome this obstacle by providing tooling support for rBPMN. This chapter describes the methodology and design of a modeling editor for the rBPMN language. We start by analyzing the requirements, then describing the architecture of the rBPMN Editor, and then we follow the GMF-based methodology reviewed in Chapter II.
Requirements Analysis

The rBPMN Editor needs to address the following high level requirements:

- Fully support the rBPMN metamodel and ensure that all semantic constraints (OCL) of the language are enforced;
- Fully support rBPMN graphical syntax so that users can create instance models using a diagram surface;
- Support diagram partitioning, where business process can be designed on a main diagram, and business rules can be detailed on separated sub-diagrams;
- Support a textual concrete syntax of rBPMN for model serialization;
- Support the development of SOAs.

For the first requirement, *fully support the rBPMN metamodel and ensure that all semantic constraints of the language are enforced*, we need to make sure that the rBPMN Editor enforce all constraints defined over the rBPMN metamodel so that it produces well-formed rBPMN models, as explained in Chapter II.

For the second requirement, *fully support rBPMN graphical syntax so that users can create instance models using a diagram surface*, the rBPMN Editor should support the rBPMN graphical syntax, which is a combination of the BPMN graphical syntax, URML, and additional elements.

For the third requirement, *support diagram partitioning*, users can create a main diagram modeling the business process containing some high level business rules, and then they can create sub-diagrams from it where business rules can be fully detailed. By keeping business rules details away from the main diagram, we do not interfere with the overall business process visualization. An example of diagram partitioning support is shown in
Figure 15 (process diagram) and Figure 16 (reaction rule sub-diagram). This example shows a Health Information System (HIS) submitting a request for patient information to an Electronic Health Record (EHR) to demonstrate the In-Out MEP with fault message (Milanović and Gašević, 2009). In Figure 15, the rule gateway checks if the user is registered and then it executes the `getInfo` activity, otherwise an event error is generated.

![Business process diagram in rBPMN](image)

Figure 15. Sample of a business process diagram in rBPMN (Milanović & Gašević, 2009).

Using the rule gateway of the process diagram, users can open a separate diagram where the business rules details can be defined. Figure 16 shows the reaction rules that are associated with the rule gateway of the process diagram.
Figure 16. Sample of a business rule diagram in rBPMN (Milanović & Gašević, 2009).

For the fourth requirement, support a textual concrete syntax of rBPMN for model serialization, rBPMN has an abstract syntax (metamodel) and a graphical concrete syntax (BPMN and URML), but lacks a textual concrete one. It is necessary to support a textual concrete syntax for model serialization.

For the fifth requirement, support of development of SOAs, we need to support different perspectives for SOA development by being able to model message exchange patterns, service interaction patterns, and workflow patterns described in Chapter II. In
addition, we should support transformations of rBPMN models into concrete service composition descriptors, e.g. support rBPMN to BPEL transformations.

Architecture

This section discusses the architecture and high-level design aspects of the rBPMN Editor. First, we decide which MDE framework will be used to develop the rBPMN Editor, then we analyze the diagram partitioning requirement that might result in one or more modeling editors to be created based on the chosen MDE framework, and finally we provide a component view of the application being developed.

MDE Framework

The first step is to define the MDE framework being used. Based on our review of MDE frameworks in Chapter II, we decided to use the Eclipse GMF framework for the rBPMN Editor development due to the following reasons:

- GMF provides a concrete graphical syntax development support;
- GMF development is a MDE approach that raises the level of abstraction for developing modeling editors in Eclipse, by auto-generating the modeling editor source code that can be extended, thus letting us focusing on the problem domain being solved rather than implementation details;
- GMF allows generating either an Eclipse IDE plug-in or a standalone application that runs on multiple operating systems;
- GMF is at a mature development stage, widely adopted, and documented (textbook, online forums, and blog posts) in contrast with other EMP graphical
syntax development frameworks such as Graphiti, which is still in the incubation phase;

- GMF is built on EMF, thus providing an abstract syntax development support;
- Ecore is similar to the MOF standard, but avoid its complexities and focuses on tool integration, and substantially influenced the latest version of MOF specification;
- EMF Validation Framework allow for enforcing all semantic constraints of rBPMN;
- EMF supports XMI for model serialization;
- As part of EMP, GMF supports model transformation with ATL (model-to-model) and Xpand (model-to-text).

Diagram Partitioning

In this section, we analyze the *diagram partitioning* requirement, to determine the modeling editors to be developed using GMF. In order to support the *diagram partitioning* requirement, we need a main modeling editor for the business process part, and four sub-editors for the business rule part (one sub-editor per rule type). In the main editor, modelers define business processes by using BPMN graphical elements and business rules by using rule gateways. The details of business rules are specified on a sub-editor for that specific rule type. For each modeling editor, we need to specify the metamodel element that is the diagram container element, i.e. the element holding all elements placed in the diagram.

For the main editor, the element container is *Collaboration*. As part of the rBPMN core package, the *Collaboration* element is used to describe interactions between two or more business entities and business roles, e.g. messages exchanged between participants of the
collaboration (Milanović et al., 2008). Figure 17 describes the *Collaboration* element containing two or more *Pools*, representing the *Participants* that take part in the *Collaboration*, whereas interactions are shown by *MessageFlow* that connect two *Pools*.

![Collaboration Diagram](image)

Figure 17. Core Package in the rBPMN metamodel (Milanović & Gašević, 2009).

In the rBPMN metamodel, a *RuleGateway* element can contain many *RuleSet* elements. The editor container element must be a concrete class. Since *RuleSet* is an abstract class, we need to use its concrete subclasses, i.e. *DerivationRuleSet*, *ProductionRuleSet*, *ReactionRuleSet*, and *IntegrityRuleSet*, so we would have one sub editor per rule type.

The main-sub-editor approach allows users to model business processes using the main rBPMN Editor, and then from concrete *RuleSet* elements associated to *RuleGateway* elements, they can open a second model diagram containing only the business rule part, e.g. the *RuleSet* and its contained elements. The second model opens on one of the rules sub-editors, thus being an R2ML model only, not containing any business process information (e.g., BPMN). The following table summarizes the modeling editors being developed and their metamodel container element:
Table 10. rBPMN Editors.

<table>
<thead>
<tr>
<th>Modeling Editor</th>
<th>Type</th>
<th>Container Element</th>
</tr>
</thead>
<tbody>
<tr>
<td>rBPMN</td>
<td>Main Editor</td>
<td>Collaboration</td>
</tr>
<tr>
<td>Derivation Rule</td>
<td>Sub-Editor</td>
<td>DerivationRuleSet</td>
</tr>
<tr>
<td>Production Rule</td>
<td>Sub-Editor</td>
<td>ProductionRuleSet</td>
</tr>
<tr>
<td>Reaction Rule</td>
<td>Sub-Editor</td>
<td>ReactionRuleSet</td>
</tr>
<tr>
<td>Integrity Rule</td>
<td>Sub-Editor</td>
<td>IntegrityRuleSet</td>
</tr>
</tbody>
</table>

*Component View*

In this section, we define the components being developed for the rBPMN Editor.

Figure 18 shows the component view of the rBPMN Editor including the dependencies on the EMP frameworks being used, i.e. GMF, GEF and EMF.
Figure 18. rBPMN Editor component view. For the sake of illustration, only the reaction rule component is displayed, but similar components for other types of rules are also provided.

The components representing the rBPMN user interface (UI) portion are at the top level. There is one UI component per modeling editor: the main editor for business processes (ca.athabasca.semtech.rbpmn.diagram) and the reaction rules sub-editor (ca.athabasca.semtech.rbpmn.diagram.reactionrules). We omitted the sub-editors for the other types of rules, as specified in Table 10, for the sake of simplicity, but they are similar to the reaction rule component. All modeling editors run on the top of GMF Runtime, and are implemented as Eclipse IDE plug-ins, being auto-generated using the GMF Tooling, and their generated Java code being customized as needed. As common functionality between all modeling editors is expected, there is an extra component to be used for all editors (ca.athabasca.semtech.rbpmn.common) that is hand coded to provide common functionality such as diagram partitioning and ability to share same diagram model between multiple editors.

The ca.athabasca.semtech.rbpmn component has two purposes. At design time, we define the models used by the GMF MDE approach (metamodel, graphical definition, tooling definition, mapping definition) and then generate the source code of the other components using EMF and GMF Tooling. It also generates its own source code using EMF that includes classes and interfaces representing the rBPMN metamodel. At runtime, it provides the object model of the rBPMN metamodel to be used by the other rBPMN Editor components.

The ca.athabasca.semtech.rbpmn.edit component contains the UI-independent portion of all editors. This component connects to the Eclipse UI Framework and contains item providers and commands to be used by the UI components. Item providers are objects
that use the *Adapter* design pattern, to provide a different interface for the rBPMN metamodel objects of the `ca.athabasca.semtech.rbpmn`, so that they can be viewed and edited in the modeling editors. This component is implemented as Eclipse plug-in, fully auto-generated by EMF, and it depends on GEF.

The following table summarizes the components being created for the rBPMN Editor, indicating its purpose and whether source code is auto-generated or hand-coded.

Table 11. rBPMN Editor Components.

<table>
<thead>
<tr>
<th>Component</th>
<th>Type</th>
<th>Purpose</th>
<th>Source Code Generation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Auto</td>
</tr>
<tr>
<td>rbpmn.diagram</td>
<td>GMF</td>
<td>UI for business process editor</td>
<td>+</td>
</tr>
<tr>
<td>rbpmn.diagram.deriavationrules</td>
<td>GMF</td>
<td>UI for derivation rules editor</td>
<td>+</td>
</tr>
<tr>
<td>rbpmn.diagram.integrityrules</td>
<td>GMF</td>
<td>UI for integrity rules editor</td>
<td>+</td>
</tr>
<tr>
<td>rbpmn.diagram.productionrules</td>
<td>GMF</td>
<td>UI for production rules editor</td>
<td>+</td>
</tr>
<tr>
<td>rbpmn.diagram.reactionrules</td>
<td>GMF</td>
<td>UI for reaction rules editor</td>
<td>+</td>
</tr>
<tr>
<td>rbpmn.common</td>
<td>N/A</td>
<td>Common UI functionality</td>
<td>-</td>
</tr>
<tr>
<td>rbpmn.edit</td>
<td>EMF</td>
<td>UI support</td>
<td>+</td>
</tr>
<tr>
<td>rbpmn</td>
<td>EMF</td>
<td>At runtime: Classes and interfaces representing the rBPMN metamodel. At design time: EMF and GMF models design for MDE approach.</td>
<td>+</td>
</tr>
</tbody>
</table>

Note. Component names are abbreviated, i.e. `rbpmn` means `ca.athabasca.semtech.rbpmn`, and so on. Source code generation are rated with “+” if the approach supports it and with “-” if it is not supported.
In this section, we defined the components being developed for the rBPMN Editor. In the following section, we define the GMF models that are necessary to be developed for the GMF MDE approach.

**rBPMN Editor Models**

The GMF MDE approach to develop concrete graphical syntaxes requires the creation of a set of models for each modeling editor defined in Table 10. Each set of models consists of a domain model (`ecore`), a graphical definition model (`gmfgraph`), a tool definition model (`gmftool`), a mapping definition model (`gmfmap`), and a generator model (`gmfgen`). Figure 19 shows all GMF models to be created for all editors with their respective dependencies.

Figure 19. rBPMN Editor models.
The domain model for the rBPMN Editor is the rBPMN Ecore metamodel (`rBPMN.ecore`). All rules editors share the same R2ML metamodel (`R2ML.ecore`). EMF supports creating models referencing other models, so that they can be extended. In the case of `rBPMN.ecore`, it refers to the BPMN metamodel (`BPMN.ecore`) and to `R2ML.ecore`, and it extends them by defining additional elements such as `RuleGateway`, `R2MLMessageType`, and `R2MLTriggeredSubProcess`, as explained in Chapter II. A common graphical definition model will be created for the rules editors, named `URML.gmfggraph`, since they have the same graphical elements for the vocabulary part, and some common graphical elements for the rule part. For example, rule conditions are available for derivation, production and reaction rule editors, and actions are available for production and reaction rules editors. Sharing a same graphical model reduces the duplication of information across the rule editors. Each graphical definition model and corresponding tooling definition and domain models are mapped through a mapping definition model as shown in Figure 19. This model is used by GMF to create the generator model, through model-to-model transformation as indicated by the “M2M” arrow. The generator model is used by GMF to generate the editor source code through a model-to-text transformation as indicated by the “M2T” arrow.

In the following sections, we discuss the design details of the abovementioned models.

Domain Model

As described in the previous section, the rBPMN Editor domain models are: 
`R2ML.ecore`, `BPMN.ecore` and `rBPMN.ecore`. `R2ML.ecore` is created using the R2ML version 0.5 metamodel specification, `BPMN.ecore` is created using BPMN 2.0 specification,
and rBPMN.ecore using Milanović and Gašević (2009) specification. Although these Ecore metamodels follow their respective specifications, it is necessary to make a few adjustments so that the proper mapping definition model can be created. GMF Tooling provides a base framework for building a graphical modeling editor, but it has some limitations. One example of limitation is when a graphical element does not directly map to a metamodel element, thus making it harder to create their mapping. This limitation becomes evident when mapping the R2ML metamodel to the graphical definition that is URML based. In order to overcome these limitations, we need to make some adjustments to the Ecore metamodels so that we can create the proper mapping model. This section discusses the changes made in the R2ML Ecore metamodel.

**Events and Activities**

For reaction rules modeling, we need to be able to represent triggering and triggered events, and also activities. In Figure 20, a reaction rule is triggered by Event A and if a certain condition holds, it triggers Event B and invokes an activity.

![Diagram](image)

**Figure 20. Reaction rule with events and activity.**
In the R2ML metamodel, events are represented as *AtomicEventExpression*. However, there is not a metamodel element that represents an activity. An *ActionEventExpression* class is introduced to represent activities, inheriting from the existing *AtomicEventExpression*.

**Rules Activities and Process Activities**

An activity in the rules diagram (Figure 20) should have a corresponding BPMN activity in the process diagram. In order to provide this association between URML activities and BPMN activities, we need to add a *flowNode* association to the *ActionEventExpression* class, so that an association can be created.

**Rules Events and Process Messages**

Similar to activities, an event in the rules diagram (Figure 20) should be associated to a message in the process diagram. In order to related URML events and BPMN messages, we inherit the R2ML *AtomicEventExpression* class from the BPMN *ItemDefinition* class. This allows using the existing *structureRef* attribute of *Message* class to choose an *AtomicEventExpression*.

**RuleSet Vocabulary Entries and Events**

*RuleSet* elements are represented in the rBPMN Editor as an element attached to a *RuleGateway*. In the rules editors, *RuleSets* represent the element container of these editors, where rules details are modeled. In order to represent vocabulary elements, such as the class in Figure 20, we need to create an association between *RuleSet* and *VocabularyEntry*. In the R2ML metamodel, a *RuleSet* is associated to a *Vocabulary*, which is then associated to *VocabularyEntry*. We are adding a direct association between *RuleSet* and *VocabularyEntry*, so that vocabulary entries can be directly represented inside a rule set in the graphic representation. This new association is show in Figure 21.
In addition to rules and vocabulary entries, events and activities must also be graphically represented inside a rule set as shown in Figure 20, thus, a zero-to-many association between RuleSet and EventExpression (events abstract base class) is also added.

Abstract Classes for Rules

Rule conditions are graphically represented as arrows and mapped to domain elements using link mappings that connect a class, association, or property to a rule element. In the R2ML domain model, each concrete class of Rule, except for IntegrityRule, contains its own conditions attribute of AndOrNafNegFormula type, as shown in Figure 5. Since each concrete rule class has its own conditions attribute, each concrete rule will have their own mapping between the condition graphical element and the corresponding domain model element. In order to reduce the number of mappings and increase reusability, we created an abstract class AbstractConditionRule that inherits from Rule (Figure 22). The conditions attribute is then moved from DerivationRule, ProductionRule, and ReactionRule classes, to this new class. The AbstractConditionRule class allows reusing the same mappings for conclusions in all rules, thus reducing the number of mappings being created, without compromising the essence of R2ML.

Figure 21. RuleSet vocabulary entries.
In addition to conditions, postCondition and producedAction also present the same problem, since both ReactionRule and ProductionRule define these attributes (Figure 5). We applied the same solution of creating an abstract class AbstractPostConditionActionRule and moving the postCondition and producedAction attributes to it as shown in Figure 22.

**Bidirectional Association between Rules and Atoms**

As mentioned in the previous section, link mapping elements are used to define the mappings for rules conditions. As described in Chapter II, the information needed for a link mapping is: domain element, containment, source end, and target end. For classification condition, a link mapping is defined using ObjectClassificationAtom as the domain element, AbstractConditionRule.conditions as containment, ObjectClassificationAtom.type as target but ObjectClassificationAtom has no attribute to define the source of the connection, i.e. a Rule attribute. In order to be able to create this link mapping, we need to extend R2ML Ecore by creating this new attribute. This problem is not restricted to classification condition, but to
all other conditions and conclusions as well. Instead of introducing this new attribute to each atom class (i.e. `ObjectClassificationAtom`, etc.), we introduce it to the base class `AndOrNafNegFormula`. Thus, `AndOrNafNegFormula` is extended with an attribute named `rule`, of type `Rule`. As a result, `Rule` and `AndOrNafNegFormula` that originally have a one-to-many association will then have a bidirectional association as shown in Figure 23.

![Figure 23](image)

Figure 23. Extending `AndOrNafNegFormula` for a bidirectional association with a `Rule`.

The problem abovementioned also applies to rule conclusions, and it is also solved by the extending the `AndOrNafNegFormula` class in addition to the change described next.

**Derivation Rule Conclusions**

Rule conclusions should be mapped to concrete classes of `Atom` as shown in Table 5. In the R2ML metamodel, derivation rule is not directly associated atoms, but through `LiteralConjunction` class as shown in Figure 5. This becomes an issue when defining link mappings for conclusions since GMF does not allow for an indirect association when defining their containment, so `DerivationRule.conclusions` attribute cannot be used. To overcome this limitation, a direct association between `DerivationRule` and `Atom` is introduced through a new attribute named `conclusionsAtoms` as shown in Figure 24.

![Figure 24](image)

Figure 24. Derivation rule conclusions.
**Condition Filters**

In URML, conditions may have filters to exclude instances of a condition classifier. URML uses OCL-like syntax in order to express filters, while R2ML metamodel uses terms to express them. Figure 25 shows `ObjectClassificationAtom` associated to an `ObjectTerm`.

![Diagram of ObjectClassificationAtom associated to an ObjectTerm](image)

Figure 25. R2ML ObjectClassificationAtom (R2ML, 2006).

In order to represent filters as OCL-like syntax in the domain model, so that they can be rendered in the diagram canvas, the `Atom` class of the R2ML Ecore is extended with a new string attribute named `filter`.

**Class Variables for Conditions and Conclusions**

Another extension needed for the R2ML Ecore is to represent variables for rules conditions and conclusions. Thus, string attributes are added to the R2ML Ecore elements that represent these conditions and conclusion as show in Table 12.

Table 12. Attributes for Conditions and Conclusions.

<table>
<thead>
<tr>
<th>R2ML.ecore element</th>
<th>Attribute Name</th>
<th>Ecore Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>ObjectClassificationAtom</td>
<td>classVariable</td>
<td>EString</td>
</tr>
<tr>
<td>PropertyAtom</td>
<td>classVariable</td>
<td>EString</td>
</tr>
<tr>
<td>ReferencePropertyAtom</td>
<td>sourceVariable</td>
<td>EString</td>
</tr>
<tr>
<td></td>
<td>targetVariable</td>
<td>EString</td>
</tr>
</tbody>
</table>
Graphical Definition Model

In the rules editors, the graphical definition model (URML.gmfgraph) uses an extended version of URML. We are leveraging the work of Strelka 0.3 for Eclipse, an URML based modeling editor (Strelka, 2006) implemented using GMF, by reusing part of their graphical definition model and extending it (Strelka 0.3 for Eclipse only supports derivation and production rules). An important difference between their work and ours is that Strelka uses a URML metamodel, while we use a R2ML metamodel, thus resulting in totally different mappings. URML does not provide graphical notation for all elements of R2ML. In this section, we define graphical elements for R2ML that are not covered by URML, but are still needed to implement the rBPMN Editor. Once defined the graphical representation for each element, a graphical definition model is created that contains the figure description for each element.

Complex Events Graphical Notation

URML defines a graphical notation for AtomicEventExpression but not for complex events: AndNotEventExpression, SequenceEventExpression, ParallelEventExpression and ChoiceEventExpression. In R2ML, complex events have a composition relationship with atomic events. We propose a graphical notation that represents this composition as a container of atomic events as shown in Figure 26. Symbols used in our proposed notation are such to resemble those of BPMN as much as possible (e.g., parallel or exclusive choice).
These graphical containers are differentiated by the symbol placed at the header. Events inside can be positioned in a row or column layouts. A row layout would be more appropriated for a `SequenceEventExpression` where events are executed in a serial order thus given the visual representation of a sequence as shown in Figure 26. A column layout would be more appropriated for the other complex events as in Figure 26 since it gives a better visual representation of parallelism, choice and exclusion. However, we do not want to enforce the layout and let rule designers to decide the best way to represent it.

Since complex events are subclasses of atomic events, we should also allow for placing complex events inside other complex events (i.e., nesting event expressions). Figure 27 shows a `ParallelEventExpression` containing a `ChoiceEventExpression`, meaning that three atomic events occur in parallel: A, B and C or D.
In addition to the graphical notation, the rBPMN Editor should allow changing the order of the elements placed inside a complex event by drag and drop. This is very important for `SequenceEventExpression` and `AndNotEventExpression` where the order of their inner events matters, and rules designers should be able to change it instead of recreating it from scratch.

**Integrity Rules Graphical Notation**

R2ML support all four rule types, but according to the URML metamodel latest version (R2ML, 2006), i.e. version 0.2, there is no URML support for integrity rules. For the integrity rules sub-editor, URML is extended by defining a graphical representation for integrity rules. For the integrity rule itself, the same graphical representation used for other rules is used here, but with using the “IR” label as shown in Figure 28. An OCL invariant is connected to the rule defining the constraint. Optionally, a constrained class can be connected to the OCL invariant.
R2ML Graphical Concrete Syntax

As previously mentioned, URML is used as the basis for the R2ML graphical concrete syntax, and extend it with the graphical notation presented in the previous sections. This section first defines how R2ML metamodel maps to URML and then presents the graphical concrete syntax being used.

For derivation, integrity, production, and reaction rules, the rule element of the URML metamodel maps directly to an R2ML metamodel element of the same name, e.g. URML DerivationRule maps to R2ML DerivationRule and so on. The URML metamodel defines three types of conditions (association, classification, and property) that need to be mapped to R2ML metamodel elements. In the R2ML metamodel, a derivation rule contains one or more conditions (e.g., concrete instances of the AndOrNafNegFormula abstract class as shown in Figure 5). Thus, each URML condition type is mapped to a specific concrete subclass of AndOrNafNegFormula: association condition maps to ReferencePropertyAtom, classification condition to ObjectClassificationAtom, and property condition to PropertyAtom. In other words, depending on the instance type of an R2ML condition, a specific URML element is mapped to it. The same applies to conditions of reaction and production rules. Production rules or reaction rules may contain a post condition, which is graphically similar to classification condition but it uses a double arrowhead instead of a single one, and it is also mapped to an ObjectClassificationAtom element.
URML conclusions have similar mappings as conditions: association conclusion maps \textit{ReferencePropertyAtom}, classification conclusion to \textit{ObjectClassificationAtom}, and property conclusion to \textit{PropertyAtom}. In addition, attribution conclusion is mapped to \textit{AttributionAtom}. URML supports four types of actions: assert, retract, update and invoke. These actions are mapped respectively to: \textit{AssertActionExpression}, \textit{RetractActionExpression}, \textit{UpdateActionExpression}, and \textit{InvokeActionExpression}. Finally, since URML is an extension of UML to add support for business rules, the vocabulary part of R2ML is graphically represented using UML elements. Table 13 shows the graphical concrete syntax that is used for creating the graphical definition model.

Table 13. R2ML Graphical Concrete Syntax (URML Extended).

<table>
<thead>
<tr>
<th>Graphical Notation</th>
<th>R2ML Metamodel</th>
<th>Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\text{DR})</td>
<td>DerivationRule</td>
<td>Association Condition: from \textit{ReferenceProperty} to \textit{Rule}.</td>
</tr>
<tr>
<td>(\text{id: 1})</td>
<td></td>
<td>Association Conclusion: from \textit{Rule} to \textit{ReferenceProperty}.</td>
</tr>
<tr>
<td>(\text{IR})</td>
<td>IntegrityRule</td>
<td></td>
</tr>
<tr>
<td>(\text{id: 1})</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(\text{PR})</td>
<td>ProductionRule</td>
<td></td>
</tr>
<tr>
<td>(\text{id: 1})</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(\text{RR})</td>
<td>ReactionRule</td>
<td></td>
</tr>
<tr>
<td>(\text{id: 1})</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(\text{filter})</td>
<td>\textit{ReferencePropertyAtom}</td>
<td></td>
</tr>
<tr>
<td>(\text{variable})</td>
<td>\textit{PropertyAtom}</td>
<td>Property Condition: from \textit{Property} to \textit{Rule}.</td>
</tr>
</tbody>
</table>
Property Conclusion: from Rule to Property

Classification Condition: from Class to Rule

Classification Conclusion: from Rule to Class

Post Condition: from a ReactionRule or ProductionRule, to a Class.

Assert Action: from a ReactionRule or ProductionRule, to a Class.

Retract Action: from a ReactionRule or ProductionRule, to a Class.

Update Action: from a ReactionRule or ProductionRule, to a Class.

Invoke Activity Action: from a ReactionRule or ProductionRule, to an ActionEventExpression.

ActionEventExpression

AtomicEventExpression

triggeringEventExpr

triggeredEventExpr
ChoiceEventExpression

ParallelEventExpression

SequenceEventExpression

AndNotEventExpression

MessageType

OCLInvariant

Class

ReferenceProperty

Class Association: from a Class to a Type.

Datatype
Tooling Definition Model

Each rules editor has a different tool palette containing the tools to create the rules and vocabulary elements for that particular type of rule. Table 14 shows the creation tools that are supported on each rule editor. If a tool is supported, then it is marked with “+”, otherwise it is marked with “-”. Based on the information provided in Table 14, tooling definition models (gmftool) are created for each rule editor.


<table>
<thead>
<tr>
<th>Palette Creation Tool</th>
<th>Rules Editor</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>DR</td>
</tr>
<tr>
<td>Derivation Rule</td>
<td>+</td>
</tr>
<tr>
<td>Integrity Rule</td>
<td>-</td>
</tr>
<tr>
<td>Production Rule</td>
<td>-</td>
</tr>
<tr>
<td>Reaction Rule</td>
<td>-</td>
</tr>
<tr>
<td>Association Condition</td>
<td>+</td>
</tr>
<tr>
<td>Classification Condition</td>
<td>+</td>
</tr>
<tr>
<td>Property Condition</td>
<td>+</td>
</tr>
<tr>
<td>Post Condition</td>
<td>-</td>
</tr>
<tr>
<td>Association Conclusion</td>
<td>+</td>
</tr>
<tr>
<td>Classification Conclusion</td>
<td>+</td>
</tr>
<tr>
<td></td>
<td>+</td>
</tr>
<tr>
<td>----------------------</td>
<td>---</td>
</tr>
<tr>
<td>Property Conclusion</td>
<td>+</td>
</tr>
<tr>
<td>Attribution Conclusion</td>
<td>+</td>
</tr>
<tr>
<td>OCL Invariant</td>
<td>-</td>
</tr>
<tr>
<td>Invariant Connection</td>
<td>-</td>
</tr>
<tr>
<td>Constraint Class Connection</td>
<td>-</td>
</tr>
<tr>
<td>Assert Action</td>
<td>-</td>
</tr>
<tr>
<td>Retract Action</td>
<td>-</td>
</tr>
<tr>
<td>Update Action</td>
<td>-</td>
</tr>
<tr>
<td>Invoke Action</td>
<td>-</td>
</tr>
<tr>
<td>Invoke Activity Action</td>
<td>-</td>
</tr>
<tr>
<td>Activity</td>
<td>-</td>
</tr>
<tr>
<td>Triggering Event</td>
<td>-</td>
</tr>
<tr>
<td>Triggered Event</td>
<td>-</td>
</tr>
<tr>
<td>Event</td>
<td>-</td>
</tr>
<tr>
<td>Choice Event</td>
<td>-</td>
</tr>
<tr>
<td>Parallel Event</td>
<td>-</td>
</tr>
<tr>
<td>Sequence Event</td>
<td>-</td>
</tr>
<tr>
<td>And/Not Event</td>
<td>-</td>
</tr>
<tr>
<td>Message Type</td>
<td>-</td>
</tr>
<tr>
<td>Message Type Connection</td>
<td>-</td>
</tr>
<tr>
<td>Class</td>
<td>+</td>
</tr>
<tr>
<td>Attribute</td>
<td>+</td>
</tr>
<tr>
<td>Operation</td>
<td>+</td>
</tr>
<tr>
<td>Association</td>
<td>+</td>
</tr>
</tbody>
</table>
Note. Rules editors are rated with “+” if the creation tool is supported, and with “-” if it is not supported.

Mapping Definition Model

Once defined the domain, graphical definition, and tooling definition models, a mapping definition model is created to map elements from these three models. Figure 29 shows a subset of rBPMN elements (events) that need to be mapped.

![Diagram of mapping definition model of rBPMN events](image)

Figure 29. Visualization of mapping definition of rBPMN events.
As mentioned in Chapter II, there are two types of mappings: diagram nodes mappings (event and activity in Figure 29) and link mappings (actions in Figure 29). For diagram nodes, the mapping consists of three elements: metamodel element, creation tool element, and graphical element. The node mapping is used for elements such as rules, events, complex events, message type, constraint, class, enumerator, data type, etc. For connecting objects, Table 15 shows all link mappings needed for the rules editors. Some link mappings may apply to multiple editors, and it is indicated by the Rule Type column. For link mappings representing metamodel elements, it is specify the element they represent, the source and target ends, and the element that stores them, i.e. containment. In Table 15, the link mapping for classification condition is defined as: metamodel element is ObjectClassificationAtom, source end is its rule attribute, target end is its type attribute (inherited from base class AndOrNafNegFormula), and containment is the conditions attribute of the AbstractConditionRule metamodel class. For link mappings representing relationships, such as generalization, invariant, and constrained class, it is specified only the target end as explained in Chapter II. In Table 15, the Invariant connection has the target end set to IntegrityRule.invariant:OCLInvariant, and then GMF allows connections from IntegrityRule (class) to OCLInvariant (invariant attribute type).
Table 15. Mapping definitions for connecting objects of rules editors.

<table>
<thead>
<tr>
<th>Name</th>
<th>Rule Editor</th>
<th>Element</th>
<th>Source End</th>
<th>Target End</th>
<th>Containment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Classification</td>
<td>D</td>
<td>P</td>
<td>R</td>
<td>I</td>
<td>ObjectClassificationAtom</td>
</tr>
<tr>
<td>Condition</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>rule: Rule</td>
</tr>
<tr>
<td>Classification</td>
<td>D</td>
<td>P</td>
<td>R</td>
<td>I</td>
<td>ObjectClassificationAtom</td>
</tr>
<tr>
<td>Conclusion</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>rule: Rule</td>
</tr>
<tr>
<td>Condition</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>rule: Rule</td>
</tr>
<tr>
<td>Conclusion</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>rule: Rule</td>
</tr>
<tr>
<td>Condition</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>rule: Rule</td>
</tr>
<tr>
<td>Conclusion</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>rule: Rule</td>
</tr>
<tr>
<td>Conclusion</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>rule: Rule</td>
</tr>
<tr>
<td>Post Condition</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>ObjectClassificationAtom</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>rule: Rule</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>rule: AbstractPostConditionActionRule</td>
</tr>
<tr>
<td>---------------</td>
<td>-----</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>-------------------------</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>context:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>AbstractPostConditionActionRule</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Class</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>rule:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>AbstractPostConditionActionRule</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>class:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Class</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>rule:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>AbstractPostConditionActionRule</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>context:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Class</td>
</tr>
<tr>
<td>Expression</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>rule:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>AbstractPostConditionActionRule</td>
</tr>
<tr>
<td>Triggered</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>ReactionRule.triggere Event</td>
</tr>
<tr>
<td>Event</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>EventExpression</td>
</tr>
<tr>
<td>Triggering</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>ReactionRule.triggere Event</td>
</tr>
<tr>
<td>Event</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>EventExpression</td>
</tr>
<tr>
<td>Message Type</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>AtomicEventExpression.type</td>
</tr>
<tr>
<td>Connection</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>EventType</td>
</tr>
<tr>
<td>Association</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>Property.domain:Class</td>
</tr>
<tr>
<td>Generalization</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>Type.superType:Type</td>
</tr>
<tr>
<td>Invariant</td>
<td>+</td>
<td></td>
<td></td>
<td></td>
<td>IntegrityRule.</td>
</tr>
<tr>
<td>Constrained</td>
<td>+</td>
<td></td>
<td></td>
<td></td>
<td>OCLInvariant.</td>
</tr>
<tr>
<td>Class</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>constrainedElement:Class</td>
</tr>
</tbody>
</table>

*Note.* Connections are marked with “+” if a specific rule editor supports it and with “-” if it is not supported. The Rule Editor legend is: “D” for derivation rule, “P” for production rule, “R” for reaction rule, and “I” for integrity rule. Source, target and containment columns are formatted as Class.attribute:AttributeType.
In addition to the mapping information, constraints can also be added to the link mappings to filter instances of source and target end of some connecting objects. For triggered and triggering events, a constraint is needed to allow only instances of *AtomicEventExpression*. Since the metamodel defines *triggeredEvent* and *triggeringEvent* attributes of *ReactionRule* as *EventExpression*, any concrete instance of *EventExpression* could be used including activities and actions as shown in Figure 29. In order to restrict triggered and triggering events to refer to events only, we need to add a constraint that instances of *ActionEventExpression* (activities) and *ProgramActionExpression* (actions) are not allowed. Similar to triggered and triggering events, message type connection should also have a constraint. A message type connection links an event or activity to a message type that defines the data structure of a Web service message. In this case, the constraint should enforce that the source end is not an instance of *ProgramActionExpression*, thus allowing only events and activities as shown in Figure 29. Table 16 shows a summary of the constraints to be added for link mappings.

**Table 16. Constraints for rules editor link mappings.**

<table>
<thead>
<tr>
<th>Link Mapping</th>
<th>Constraint</th>
</tr>
</thead>
<tbody>
<tr>
<td>Triggered Event</td>
<td>Target cannot be an instance of <em>ActionEventExpression</em> or <em>ProgramActionExpression</em>.</td>
</tr>
<tr>
<td>Triggering Event</td>
<td>Target cannot be an instance of <em>ActionEventExpression</em> or <em>ProgramActionExpression</em>.</td>
</tr>
<tr>
<td>Message Type Connection</td>
<td>Source cannot be an instance of <em>ProgramActionExpression</em>.</td>
</tr>
</tbody>
</table>
Once all models have been defined (see Figure 19), the source code of the rBPMN Editor can be generated with GMF Tooling through its MDE approach. This source code can be extended with customizations described in the following section.

**Custom Code**

The auto-generated Java code for the rBPMN Editor generated from the models described in the previous sections can be customized if we need a different behavior or need to extend its functionality. In this section we briefly describe the most relevant customizations needed for the rBPMN Editor.

**Shared Editing Domain**

In the diagram partitioning approach, the overall business process diagram is edited in the main rBPMN Editor, and its rule sets can be detailed in separate URML-like diagrams in the rules editors. However, all these separate diagrams represent the same rBPMN model underneath but in different views: overall business process view and rule set detail views. This implies that when a particular view is changed, this change should reflect on the underneath rBPMN model, and changes on the rBPMN model might affect other views of the same model. This is modeling editor behavior is known as shared editing domain. The shared editing domain behavior is not auto-generated by GMF Tooling and we will be using a solution provided by the gmftools open source project (gmftools, 2011). This custom code will be implemented in the ca.athabascau.semtech.rbpmn.common component as seen in Figure 18, so that it can be used by all modeling editor UI components.
**Complex Event Containers**

As mentioned in the previous section, *SequenceEventExpression* and *AndNotEventExpression* must have their inner events in a specific order. The behavior provided by the auto-generated code always adds new events at the end of the container and it does not allow for changing the inner events order. In order to provide the *drag and drop ordering* and *add at a specific position* behaviors, we implement the recipes provided by Eclipse Foundation (2011j). This custom code is only related to the reaction rules sub-editor, and thus should be placed at the `ca.athabasca.semtech.rbpmn.diagram.reactionrules` component.

**Negated Conditions and Conclusions**

As shown in Table 13, rules conditions and conclusions are represented as arrows. If a condition or conclusion is negated (i.e. `IsNegated` property is true), then its arrow is crossed at the origin. The rules editors should automatically cross or uncross the condition/conclusion arrow whenever the `IsNegated` property changes to true or false. This dynamic behavior of changing the graphical representation of an element based on its state is not provided by the generated editors thus needs to be manually implemented.

Each condition/conclusion has a concrete subclass of `ConnectionNodeEditPart` that provides its UI functionality. In order to implement the behavior abovementioned, it is necessary to customize these condition/conclusion edit part classes by overriding the `handleNotificationEvent` method to listen to changes in the condition/conclusion state, and then update its graphical representation by crossing or uncrossing it based on the `IsNegated` property. This same technique can be used to dynamically update a graphical representation of a diagram element based on state changes of the underling model element.
Class Attribute Parser

The default behavior when adding an attribute to a class is to use the entered text as the name of the attribute, and then the attribute type must be set in the editor property window. Ideally, if the entered text is the format “name : Type”, then it should automatically set the attribute name and type. To implement this behavior, it is necessary to extend the MessageFormatParser class that is auto-generated, by creating a subclass AttributeParser and overriding the getParseCommand to parse the text input and set the appropriated properties.

Additional Design Details

This project is collaboration with Milanović (2010) who focused on the rBPMN process editor, while this project focused on the rBPMN rules editors and their integration with the process editor. Similar design details for the business process parts of the rBPMN Editor are available in Milanović’s thesis (Milanović, 2010).

Model Transformations

This subsection briefly discusses model transformations required by the rBPMN modeling editor, in order to support rule interchange or to generate concrete service composition descriptions (i.e. BPEL). Although these models are important for the complete editor, they are not implemented due to time available for and an expected amount of work for a master’s project, and are cited here for future work reference.

For rule interchange, rBPMN Editor could export its model into R2ML XML or RIF. Although RIF has recently became the standard format for rule interchange (W3C, 2010), we look at R2ML XML since rBPMN is already based on R2ML. In the rBPMN rules editors, rules are graphically modeled using URML, stored as R2ML models, and they should be
transformed into R2ML XML for rule interchange. This scenario has two transformations: a model-to-model transformation from URML to R2ML, and a model-to-text transformation from R2ML to R2ML XML. Figure 30 shows these transformations by representing two technical spaces (TS): MDE and XML. In the MDE TS, rules are graphically represented as URML models that conform to an URML metamodel, which are transformed into R2ML models that conform to R2ML metamodel. This model-to-model transformation is already handled by GMF since the rules editors use URML as graphical definition and R2ML metamodel as the domain model. So, URML models are stored as R2ML models, using XMI serialization.

![Diagram](image_url)

Figure 30. R2ML transformations.
The second transformation, a model-to-text transformation, translates R2ML models (abstract syntax) into R2ML XML rules (textual concrete syntax). R2ML XML rules, part of the XML TS, conform to R2ML XML schema, which conforms to the XML meta-schema (XSD.xsd), e.g. a schema for XML schemas, which conforms to itself. A transformation language instance, also conforms to a transformation language metamodel (omitted in the Figure 30 for simplicity) uses a mapping function between R2ML metamodel and R2ML XML schema when it is executed by transformation engine. The transformation language to be used is Xpand, since it is the model-to-text transformation language of the MDE framework used for developing the rBPMN Editor. The mappings between R2ML and R2ML XML are defined based on Milanović et al. (2009c) work mentioned in Chapter II. A similar approach for RIF and other potential rule formats can be used.
CHAPTER IV

IMPLEMENTATION DETAILS

This chapter describes the implementation details of the rBPMN Editor that includes the development environment used to implement it, the rBPMN process editor and its use for process design, and the rules editors and their use for data and rules design.

rBPMN Editor Development

The tools used for the rBPMN Editor development are:

- Eclipse Galileo 3.5 SR1 – Modeling Package: an Eclipse package including the Eclipse Modeling Project components, i.e. EMF, GMF, etc.
- Java Development Kit (JDK) 6.0 Update 16 (or latest update).
- GMF SDK Experimental: an SDK that provides the org.apache.batik libraries used to support Scalable Vector Graphics (SVG) files.
- Microsoft Visio 2010 to develop SVG files.

The Eclipse Modeling Package provided all the tools to build the models described in Chapter III (Figure 19). Although most of the graphical elements were defined using GMF, some of them, such as the header figure of complex events shown in Figure 26, were created using SVG files, and then referenced in the graphical definition model. Custom code was implemented using the Java language and external references were used as described in Chapter III. We use GMF Tooling to generate the rBPMN Editor as Eclipse plug-ins. It generates a set of Java archive (JAR) files that are installed in Eclipse. These JAR files are also available for download (Gašević et al., 2011).
rBPMN Main Editor

After the rBPMN Editor is installed in the Eclipse environment, users are able to create new rBPMN diagrams using the New File Wizard, and selecting rBPMN Diagram as shown in Figure 31.

![New File Wizard](image)

Figure 31. Creating rBPMN diagram wizard.

When the New File Wizard finishes, two files are created for the new rBPMN diagram: a model file and a diagram file. The model file (rbpmn extension) contains the XMI serialization of the rBPMN model. The diagram file (rbpmn_diagram extension) refers to the elements of the model file, containing the information of how they are displayed in the diagram canvas (position, size, etc.). Figure 32 shows both files created in the Project Explorer at the editor left side (sample.rbpmn and sample.rbpmn_diagram). At the center,
the diagram file is opened and it is initially empty. At the right, the tool palette provides the available tools to create elements in the diagram.

![Figure 32. rBPMN Editor with an empty diagram.](image)

The diagram in Figure 32 is opened in the main rBPMN editor, i.e. the process editor. The tool palette contains BPMN elements grouped into multiple sections, including the Rules section that contains the new elements introduced by rBPMN as shown in Figure 33.
Process Design with the rBPMN Editor

The rBPMN Editor can be used to develop rule-enabled SOAs using the methodology presented in Chapter II. The rBPMN Process Editor provides the necessary tools for the process design step. Figure 34 shows the process diagram created in the rBPMN Editor for the In-Out MEP with fault message example presented in Figure 15. To create this diagram,
we start with the empty diagram from Figure 32, and click on the Pool element under the Swimlanes section of the tool palette, and then click on the diagram surface for the pool to be created. We name the first pool as Health Information System (HIS), and repeat the same steps to create a second pool named Electronic Health Record (EHR).

Figure 34. Process design with rBPMN Editor for the Patient Info Request sample as per case study from (Milanović & Gašević, 2009).

For the HIS pool, a Start Event element and an Task named Request Patient Info are added, and then connected with a Sequence Flow element. For the EHR pool, a Message Intermediate Catch Event and a SubProcess are added and connected with a Sequence Flow.
The Request Patient Info task is connected to the Message Intermediate Catch Event through a MessageFlow element, with a Message element attached to it. In the EHR pool subprocess, the following elements are added: a StartEvent, a RuleGateway, a task named “getInfo”, an EndEvent, an ErrorEndEvent, and a ReactionRuleSet named Rule Set 1. They are connected with SequenceFlow elements as shown in Figure 34. The RuleGateway is connected to the ReactionRuleSet with a RuleSet Connection (dashed line). A Message Catch Intermediate Bondary Event is attached to the subprocess border, and a task named Handle Exception is added to the EHR pool, and they are connected with a SequenceFlow. In the HIS pool, an EventBasedGateway is placed after the Request Patient Info task, where the process flows to two Message Intermediate Catch Events, and then it ends. The getInfo and Handle Exception tasks are connected to these Message Intermediate Catch Events using Message Flows with Message elements attached. In Figure 34, the getInfo task is selected, and the Properties window at the bottom allows changing properties of this task, such as its name.

In addition to interconnected interface behavior models as shown in Figure 34, the rBPMN Editor also support the creation of interaction models. Figure 35 show an interaction model built using the rBPMN Editor for the patient info request sample. In this model, a Message Flow element is connected from the bottom border of the HIS pool to the top border of the EHS pool, and a Message Start Event element is attached to the Message Flow. Similar procedure for the response messages, but in the opposite direction and using a Message Catch Intermediate Events attached. A rule gateway is added to the diagram and connected to the EHS pool by using an Executed by Association element. A reaction rule set is added to the EHS pool, and connected to the rule gateway using the RuleSet Connection element.
Finally, *Sequence Flow* elements are added to connect the *Message Start Event* to the rules gateway, and to the *Message Catch Intermediate Events*.

![Diagram](image.png)

**Figure 35.** Interaction model for the Patient Info Request sample.

**rBPMN Rules Editors**

One important difference between the process diagram built with the rBPMN Editor (Figure 34) and the process diagram presented by Milanović and Gašević (2009) (Figure 15), is the presence of the reaction rule set (*Rule Set 1*) attached to the rule gateway. In the rBPMN Editor implementation, a rule set is used to define the details of the business rules. If the user double-clicks on this rule set, then an additional diagram tab will open in the rBPMN Editor where these rules can be modeled (Figure 36). In this case, it opens the Reaction Rule sub-editor since the rule set used in the process diagram is a *ReactionRuleSet*. Note that when
the rules diagram opens, it is still using the same model and diagram files as the process diagram. The information is still stored in the same place, but the rBPMN Editor is providing a detailed view (rule set view) instead of an overall view (process view). For each rule set added to the process diagram, a tab would be opened to model the rules details, by invoking the respective sub-editor based on the type of the rule of the rule set.

Figure 36. rBPMN Reaction Rules Editor with an empty diagram.

In addition to supporting the rBPMN Process Editor, rules editors can also run as standalone modeling editors. Users can create standalone rules diagrams that are not connected to process diagrams, thus working as an URML modeling tool similar to Strelka
(Strelka, 2006). For standalone mode, rules diagrams can be created directly from the New File Wizard as shown in Figure 37.

![Figure 37. Creating rules diagram wizard.](image)

Similar to the rBPMN diagram wizard, rules diagram wizards create the model and diagram files. Diagrams files have different extension based on the type of rule, but model files have the same rbpmn extension as the process diagram. Model files differ by their top level element as shown in Table 17. Although model files have the same extension since they are all XMI serialization of rBPMN models, you could not use a model file from one rule editor in another rule editor since their top level element differ.

### Table 17. rBPMN Editor New File Wizards.

<table>
<thead>
<tr>
<th>File Wizard</th>
<th>Diagram File Extension</th>
<th>Model File</th>
</tr>
</thead>
</table>

106
<table>
<thead>
<tr>
<th>Rule</th>
<th>Extension</th>
<th>Top Level Element</th>
</tr>
</thead>
<tbody>
<tr>
<td>rBPMN</td>
<td>rbpmn_diagram</td>
<td>Rbpmn</td>
</tr>
<tr>
<td>Derivation Rule</td>
<td>derivationrule_diagram</td>
<td>Rbpmn</td>
</tr>
<tr>
<td>Integrity Rule</td>
<td>integrityrule_diagram</td>
<td>Rbpmn</td>
</tr>
<tr>
<td>Production Rule</td>
<td>productionrule_diagram</td>
<td>Rbpmn</td>
</tr>
<tr>
<td>Reaction Rule</td>
<td>reactionrule_diagram</td>
<td>Rbpmn</td>
</tr>
</tbody>
</table>

Each rule editor has different tool palette as defined in the tooling definition model in Chapter III (Table 14). Figure 38 shows the tool palette for each rule editor. All palettes have two groups: Rules and Vocabulary. The Vocabulary group is identical for Derivation and Integrity Rule editors, and for Production and Reaction Rules editors, it contains two additional elements: Message Type and Message Type Connection.
Figure 38. Tool palettes of the rBPMN rules editors: Derivation Rule, Integrity Rule, Production Rule, and Reaction Rule.

Data and Rules Design with the rBPMN Rules Editors

Similar to the rBPMN process editor, the rBPMN rules editors provide the necessary tools for the data and rules design steps of the methodology to develop rule-enabled SOAs presented in Chapter II. Figure 39 shows the rules diagram created in the rBPMN Editor for
the conceptual reaction rules diagram presented in Figure 16. To create this diagram, we start the data design from the empty reaction rule diagram shown in Figure 36. The data design consists of defining the messages and rules related data objects using the palette tools of the Vocabulary group. In the business process diagram (Figure 16), we have three messages: get student information request, student information response, and request failed response. These messages are defined using the Message Type element from the Vocabulary palette. For the request fail InfoRequestFail, the Properties window is used to change the Stereotype property from its default value to fault message event type. Attributes are added to the message type by clicking on the Attribute palette tool, and then in the attributes region of the message type. In addition to message types, data objects are also defined using the Class palette tool. Associations can be created between classes and also between a class and a message type as shown in Figure 39 for the GetInfoRequest message type and the NameAddressCredential class.

For rules design, the palette tools from the Rules group are used. Reaction rules elements are added and their ids are set. Two events are added Get Info and Request Fail. The Triggering Event and Triggered Event are used to connect the reaction rule to its events. Reaction rules can also invoke activities by using the Invoke Activity Action tool as shown for getInfo activity in Figure 39.
Figure 39. Data and rule design with the rBPMN Editor for Patient Info Request sample as per the case study from (Milanović & Gašević, 2009).

Once events and activities are added, they should be connected to their related messages using the Message Type Connection tool from the Vocabulary palette. In Figure 39, reaction rule 2 is associated with the valid credentials condition that is created with the Classification Condition tool, by connecting reaction rule 1 to the RegisterUser class and entering the filter expression (\texttt{req.nameAddressCredential.name = u.username and req.nameAddressCredential.credential = u.type}). The Properties window of the
classification condition is used to enter the \textit{Class Variable} value, which is “u” in this case. Similar to reaction rule 2, reaction rule 1 has the same condition but it is negated as indicated by the cross mark at its origin. In order to negate a condition, users need to set the \textit{Is Negated} property to \textit{true} in the \textit{Properties} window as indicated in Figure 39.

\textbf{Connecting Rules and Process Diagrams}

For connecting rules activities to business process tasks, the \textit{Flow Node} property of an activity should be set to the corresponding task in the process diagram. In Figure 40, the \textit{Flow Node} property of the \textit{getInfo} activity is set to \textit{Task getInfo} that represents the \textit{getInfo} task of the process diagram of Figure 34.

![Figure 40. rBPMN Editor – Connecting rules activities to process tasks.](image)
For connecting messages of the process diagram to events of the rules diagram, the Structure Ref property of the message element should be set to the corresponding event of the rules diagram. In Figure 41, the Structure Ref property of the message in the process diagram on the left side is set to Atomic Event Expression Get Info that corresponds to the Get Info event of the rules diagram on the right side.

Rules in the rBPMN Main Editor

Although rules can be modeled in full details using the rules editors, the process editor also enables designing rules. The Rules section of the main editor palette (Figure 33) provides elements to define rules within a rule set in the process diagram. Here, rules can be defined at a high-level and then detailed using the rules editors later. In Figure 42, the
The diagram shows an implementation of the *Multiple Instances without Synchronization* workflow pattern. In this pattern, an activity (*Activity 1*) is executed multiple times and a rule gateway defines the looping condition for this execution. In the rule set element, a running counter (*counter* attribute) is used for the logical loop. After the *Activity 1* is executed, the rule gateway uses a reaction rule (id: 1) to verify if this activity has been executed *total* times. It uses a negated condition of *counter = total*. If this condition is false, it then increases the counter by one (update action) and the process flows back to the gateway before *Activity 1*. This process loop continues until *counter = total* is true, and then the process executes *Activity 2*.

![Diagram showing a workflow pattern with a rule gateway and reaction rule.](image)

**Figure 42.** Rules modeling in the rBPMN process editor.

**rBPMN Model File**

The rBPMN Editor stores its models in XML format using XMI serialization. Figure 43 shows a partial listing of the *sample.rbpmn* model file for the diagrams of Figure 34 and Figure 39. The listing in Figure 43 includes the entire subprocess of the EHR pool from Figure 34, and the reaction rule set portion is highlighted within the red rectangle.
If any changes are made directly to the model file, then the process and rules diagram will automatically refresh. For example, if an element is deleted from the model file, then its corresponding graphical element will be automatically removed from the diagram. The same
is valid when adding new elements or changing existing properties. When adding new
elements, the affected diagrams will have the corresponding graphical element added to the
top left corner. When changing existing properties, for example the name of an activity, the
corresponding diagram is refreshed and the new name is shown.

In this chapter, we reviewed the implementation of the rBPMN Editor showing the
process editor, rules editors and how to use them to design process, data and rules. In the next
chapter, we describe how the rBPMN Editor can be used to model some real case scenarios.
CHAPTER V

CASE STUDIES

This chapter discusses how the rBPMN Editor can be used to model online auctions (English and Dutch auctions) and vehicle insurance scenario (UServ Product Derby), using the methodology for developing rule-enabled SOAs described in Chapter II. These case studies are an addition to the case studies created by Milanović (2010): modeling service orchestrations, modeling choreographies, and modeling agile business processes.

English Auction

Online auctions have become very popular with the spread of the Internet. English auction is a type of auction where the price is successively raised until only one bidder remains (McAfee, McMillan, 1987). Here, we present a case study based on an online English action similar to the one used by the ebay.com website.

Requirements Specification

In an English auction, a seller creates a new auction request by setting a starting price, price increment and duration of the auction. The auction system receives the new auction request and if it is valid, starts the auction. Once started, bidders can submit bids to the auction system. The first bid is accepted if it is greater or equal than the starting price. A subsequent bid is accepted if it is greater or equal than the previous bid plus the price increment. When the auction ends, the first bidder becomes the winner, a payment is collected from the buyer, and it is sent to the seller after fees are deducted. If there is no first bidder, then the auction is unsuccessful.
Process Design

The first step of the process design is to create a high level business process model using the rBPMN Editor. Figure 44 shows a business process in rBPMN consisting of three participants represented as pools: seller, auction system, and buyers. The seller starts the process by sending a new auction request to the auction system, represented by the Request New Auction task. The new auction request message contains the required information to create a new auction: good information, starting price, price increment, and duration of the auction. The auction system receives this message using a message catch intermediate event, and it checks if the new auction request information is valid (Validate Auction task). If it is valid, then a new auction is created (Create Auction task), and a confirmation message is sent back to the seller indicating the time the auction has started. If it is invalid, the auction system sends a message to the seller with the rejection reason using the Reject Auction task, and the process ends. Once the auction is created, buyers can submit bids. The buyer pool is annotated with the multiple-instance participant marker, i.e. the ||| symbol, to indicate that more than one buyer can submit bids. A buyer submits a bid using the Submit a Bid task. The auction system receives this message using a message catch intermediate event, and checks if the bid is valid (Validate Bid task). If it is invalid, it rejects it by sending a message to the buyer with the rejection reason using the Reject Bid task. If the bid is valid, a confirmation message is sent to the buyer using the Accept Bid task. In either way, the process flows back to the message catch intermediate event where it can catch additional bid request events and the auction time out event.
Figure 44. English Auction high-level business process in rBPMN language.
When the auction times out, the Determine Winner task is executed to determine the buyer with the highest bid. A payment request message with the amount due is sent to the winner using the Request Payment task. The winner receives this message using a message catch intermediate event, and sends the payment information message using the Send Payment task. The seller receives this message, and sends the product delivery information message to the buyer using the Send Product task. For all remaining buyers, the auction system sends an end of auction message using the End Auction multiple instance task. Buyers receive it using a message catch intermediate event and their processes end.

Once the high level business process is defined, we need to identify variability points and estimate their level of variability. There are three variability points (variable tasks): Validate Auction, Validate Bid, and Determine Winner tasks. For each variability point, we determine its level of variability and decide if we will model it using business rules or not.

The first variability point, Validate Auction task, represents a decision that can be based initially on a simply validation of the information of the new auction request (starting price, increment, etc.), but it can be extended to verify seller’s auction history, previous delivery delays, quality of goods sold, thus determining the seller’s reliability and rejecting bids from sellers that are not reliable. This results in a high frequency of changes. In addition, the source of change is internal, i.e. within the organization. Based on this, the first variability point should be modeled using business rules. The second variability point, Validate Bid task, has also an internal source of change, and a high frequency of changes since it should verify if the new bid is higher than previous ones and might have additional verifications to determine buyers reliability (credit history, purchase history, delayed payments, etc.). Thus, the second variability point should also be modeled using rules. The third variability point,
Determine Winner task, has also an internal source of change. Here, if the auction has a winner, a payment request will be sent to the auction winner, and auction fees will be calculated. The calculation of the fees can vary since the auction system can give discounts to the seller based on seller sales history. Thus, the third variability point should also be modeled using rules since the frequency of changes is high.

The final step of the process design is to determine the appropriated pattern for these variability points. For the first variability point, we will use the Exclusive Choice workflow pattern to model the orchestration. When the request new auction message is received, if the condition is true (i.e. valid request), then the Create Auction task is executed. If the condition is false, then the Reject Auction task is executed. For modeling the choreography, we use a Send/Receive service interaction pattern, where the seller participant sends a request to the auction system participant, who then returns a response message: a create auction response or a fault auction response. For the second variability point, we have a similar situation as the first variability point: the process flow will execute the Accept Bid task or the Reject Bid task depending on the valid bid condition, and a buyer participant sends a bid request message and receives a bid confirmation response or a bid fault response. Thus, we choose the Exclusive Choice pattern for the orchestration, and Send/Receive pattern for the choreography. For the third variability point, we need to use a combination of two workflow patterns Parallel Split pattern and a multiple instance pattern for the orchestration. Here, we have two tasks that can be performed in parallel: Request Payment task and End Auction task. Since multiple instances of the End Auction task can be created and run concurrently, all instances must be completed before the process flow continues, but the number of instances is not known at design time, then we choose the Multiple Instances without Synchronization
pattern. For the choreography modeling, we choose the *One to Many Send* pattern, since the auction system sends several messages to multiple buyers in parallel. The following table summarizes the patterns used for all variability points of the English Auction process.

Table 18. Patterns used for the English Auction Case Study.

<table>
<thead>
<tr>
<th>Variability Point</th>
<th>Workflow Pattern (orchestration)</th>
<th>Service Interaction Pattern (choreography)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Exclusive Choice</td>
<td>Send/Receive</td>
</tr>
<tr>
<td>2</td>
<td>Exclusive Choice</td>
<td>Send/Receive</td>
</tr>
<tr>
<td>3</td>
<td>Parallel Split + Multiple Instances without Synchronization</td>
<td>One to Many Send</td>
</tr>
</tbody>
</table>

Before starting the data design using the rBPMN Editor, we need to replace variable activities with rule gateways according to the selected patterns. All the patterns in Table 18 are implemented using reaction rules. Although we are not defining these rules at this point, it is necessary to know the type of rule to be used so that we can attach the proper *Rule Set* to the *Rule Gateway* element, by using a *RuleSet Connection*. Once all rule sets are placed on the process diagram, then we will be able to open the rules editor for each rule set and start the data design step. Figure 45 shows the business process diagram with rule gateways and respective rule sets on the variability points.
Figure 45. English Auction business process in rBPMN language.
Data Design

During data design, we define the domain model (data objects and messages) using the rules editor of the rBPMN Editor. For the first variability point, we open Rule Set 1 in the rules editor, and define the messages related to the Send/Receive service interaction pattern for the first variability point. The seller sends a NewAuctionRequest message to the auction system, and it should have the starting price, price increment, duration of the auction, and the good information. This message is annotated with the message event type stereotype to allow us to use it in activities and rules as input and output. The auction system provides two responses. The CreateAuctionResponse message is sent back to the seller if the new auction request is valid, and it should have the starting time of the auction. The FaultAuctionResponse message is sent back to the seller if the new auction request is invalid, and it should have the information describing the reason for rejecting the new auction (reason attribute).

For the second variability point, we open Rule Set 2 in the rules editor to define the messages and related data objects. The buyer sends a BidRequest message to the auction system, and this message should have the bid data object. The bid data object should contain the value of the bid, the auction object, and the buyer (User type). Figure 46 shows the BidRequest message and its associated objects. The BidResponse message is sent back to the buyer if the bid is valid and accepted. The FaultBidResponse message is sent back to the buyer if the bid is invalid, i.e. there is already a higher bid for this auction. This message should have the information describing the reason to reject it (reason attribute).
For the third variability point, we open *Rule Set 3* in the rules editor to define its related messages. A *RequestPayment* message is sent to the buyer who wins the auction, and it should have the amount to be paid (*Decimal* type), and the buyer and payee (*User* type). The *EndAuction* message is sent to those buyers who did not win the auction.

**Rule Design**

The first variability point, new auction request is valid, is modeled using two reaction rules triggered by the *New Auction Request* event as shown in Figure 47. Reaction rule 1 is associated with the condition of a valid auction request, and it invokes the *Create Auction* activity followed by a *CreateAuctionResponse* message sent to the seller. Reaction rule 2 is associated with the negate condition of a valid auction request, and it invokes the *Reject Auction Request* activity followed by a *FaultAuctionResponse* message sent to the seller. After modeling the business rules, we need to connect the rules activities to activities (tasks, subprocess, etc.) of the business process diagram. In the *Create Auction* activity property window, we set its *Flow Node* property as the *Create Auction* task of the process diagram.
For the *Reject Auction Request* activity, we set its *Flow Node* property as the *Reject Auction* task of the process diagram.

![Diagram showing the process flow for auction requests and reactions rules.](image)

Figure 47. English Auction Rule Set 1

The second variability point, determine if the buyer's bid is valid, is modeled using two reaction rules triggered by the *Bid Request* event as shown in Figure 48. The *Bid Request* event is associated with the *NewAuctionRequest* message sent by the buyer participant. A bid is valid if it is greater than or equal to the current minimum bid price. The current minimum bid price is equal to the auction starting price plus the auction bid increment when there are no bids entered yet, or it is equal to the highest bid price plus the auction bid increment when it is the first auction bid. In Figure 48, reaction rule 3 is associated with the positive condition of a valid bid, and it triggers two actions: an action to invoke the *Accept Bid* activity followed by a *BidResponse* message sent to the buyer, and another action to update the auction highest
bid \( (\text{highestBid} = \text{bid}) \). Reaction rule 4 is associated with the negate condition of a valid bid, and it triggers an invoke action to the \textit{Reject Bid} activity, followed by a \textit{FaultBidResponse} message sent to the buyer. For the \textit{Accept Bid} activity, we set its \textit{Flow Node} property to the \textit{Accept Bid} task of the process diagram, and for the \textit{Reject Bid} activity, we set its \textit{Flow Node} property to the \textit{Reject Bid} task.

Figure 48. English Auction Rule Set 2.

The third variability point, determine if an auction has a winner when it ends, is modeled by using three reaction rules triggered by the \textit{Auction Timeout} event as shown in Figure 49. In this rule set, we will loop through the list of bids to request the payment from the buyer associated to the winner bid, and notify the other buyers that the auction has ended. In Figure 49, reaction rule 5 is associated with the negate condition of remaining in the loop (i.e., \( \text{counter.value} < \text{bids.size} \)) and it triggers the \textit{Stop Counter} activity. Reaction rule 6 is associated to the condition of remaining in the loop and the condition that the current bid is
the highest bid (i.e., \(\text{counter.value} < \text{bids.size and bid} = \text{bid.auction.highestBid}\)). It triggers an invoke action to the \textit{Request Payment} activity followed by a \textit{RequestPayment} message sent to the buyer, and it also triggers an update action to increment the counter by one (i.e. \(\text{counter.value} = \text{counter.value}_{@pre} + 1\)). Reaction rule 7 is associated to the condition of remaining in the loop and the negate condition that the current bid is the highest one. It triggers an invoke action to the \textit{End Auction} activity followed by an \textit{EndAuction} message sent to the buyer, and it also triggers an update action to increment the counter by one. For the \textit{Request Payment} activity, we set its \textit{Flow Node} property to the \textit{Request Payment} task of the process diagram, and for the \textit{End Auction} activity, we set its \textit{Flow Node} property to the \textit{End Auction} task.
Connecting Process Messages to Rules Events and Activities

During the rule design, we connected activities created in the rules editor to activities in the process diagram by setting the Flow Node property of the rules activity to the related process activity. In addition, we go back to the process diagram, attach Message data objects to Message Flow connections, and set its Structure Ref property as the related rule event (AtomicEventExpression) or activity (ActionEventExpression) associated to the Message Type in the rules diagram. For the auction request message, we set its Structure Ref property
to Atomic Event Expression New Auction Request, which represents the New Auction Request event of Rule Set 1. We repeat the same procedure for all remaining messages in the process diagram.

**Interaction Model**

Another way to represent the process diagram is by creating an interaction model that focus on the choreography aspects. For the first variability point, we first add a Message Flow element from the Seller pool to the Auction System pool. Then, we attach a Message Start Event to the Message Flow to represent the New Auction Request message as shown in Figure 50. The auction system sends two possible response messages back to the seller. For each response message, we add a Message Flow from the Auction System pool to the Seller pool, and attach a Message Catch Intermediate Event to the Message Flow (see Auction Rejected and Auction Created in Figure 50). We add a Rule Gateway close to the Auction System pool, and connected it to the pool by using an Executed by Association element. We then add a reaction rule set (Rule Set 1) to the Auction System pool, and connect it to the Rule Gateway using a RuleSet connection. We finally connect the Message Start Event to the Rule Gateway, and the Rule Gateway to the Message Catch Intermediate Events using Sequence Flow elements. We follow a similar approach for the second and third variability points. The data design and rule design are the same as described in the previous sections.
In this case study, we show how the rBPMN Editor can be used to model a Dutch auction. The Dutch auction is very similar to the English auction. In a Dutch auction, the seller sets an initial high price, and then lowers it until one bidder accepts the current price (McAfee, McMillan, 1987). Due to their similarities, we only present the differences between these two auctions.

**Requirements Specification**

In a Dutch auction, a seller creates a new auction request by setting the minimum price, starting price, price decrement and the time interval to decrement the current price. The auction system receives the new auction request and if it is valid, starts the auction with the
starting price. Once started, bidders can submit bids to the auction system. A bid is accepted if it is greater or equal than the minimum price. The auction price is decremented according to the time interval defined when creating the auction. When the price is decremented, if the highest bid is greater than the current price, then the auction successfully ends. A payment is collected from the buyer, and it is sent to the seller after fees are deducted. In the case of a draw, then the bid entered first wins (first come first serve). If the highest bid is not greater than the current price, then the auction continues until the current price becomes lower than the highest bid (successful auction) or the current price becomes lower than the auction minimum price (unsuccessful auction).

Process Design

The first step of the process design is to create a high level business process model using the rBPMN Editor. The high level business process model for the Dutch auction is almost identical to the English auction shown in Figure 44. It has the same set of variability points: Validate Auction task, Validate Bid task, and Determine Winner task. The only difference is the third variability point (Determine Winner task) that needs to loop back the process flow to the event-based gateway before the timer intermediate catch event if there is no winner. As described for the English auction, we need to use a combination of the Parallel Split pattern and the Multiple Instances without Synchronization pattern. In addition, we also need use the Arbitrary Cycles pattern to model the process loop to continue the auction with the decremented price when there is no winner. Table 19 summarizes the patterns used for the Dutch auction variability points.

Table 19. Patterns used for the Dutch Auction Case Study.
<table>
<thead>
<tr>
<th>Variability Point</th>
<th>Workflow Pattern (orchestration)</th>
<th>Service Interaction Pattern (choreography)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Exclusive Choice</td>
<td>Send/Receive</td>
</tr>
<tr>
<td>2</td>
<td>Exclusive Choice</td>
<td>Send/Receive</td>
</tr>
<tr>
<td>3</td>
<td>Parallel Split + Multiple Instances without Synchronization + Arbitrary Cycles</td>
<td>One to Many Send</td>
</tr>
</tbody>
</table>

Figure 51 shows the Dutch auction business process in rBPMN. Once an auction is created, the *Reset Price Timer* task is executed to reset the timer for the period that the auction current price is valid. When the time-out event occurs, the process flows to the rule gateway that verifies if the auction has a winner (*Rule Set 3*). If the auction does not have a winner, then the auction current price is decremented. If the current price is still greater than the auction minimum price, then the process loops back to the *Reset Price Timer* task, and the auction continues with a decremented price.
Figure 51. Dutch Auction business process in rBPMN language.
**Data Design**

During data design, we define the domain model (data objects and messages) using the rules editor of the rBPMN Editor. For the first variability point, we have the same set of messages used in the English auction case study, but some attributes are different. The `NewAuctionRequest` should have the minimum price, starting price, price decrement, price duration interval, and the good information. The `CreateAuctionResponse` and `FaultAuctionResponse` messages are identical as the English auction. For the second variability point, we have the same messages and data objects as the English auction (Figure 46). The only difference is regarding the Auction class that has an additional attribute, `minimumPrice` as `Decimal`, and it has the `interval` attribute instead of `duration`, and the `decrement` attribute instead of `increment`. For the third variability point, we use the exact same `RequestPayment` and `EndAuction` messages as described for the English auction.

**Rule Design**

The first variability point (auction validation) is modeled using two reaction rules that are triggered by the `New Auction Request` event as shown in Figure 52. This is almost identical to the English auction `Rule Set 1`, except for the condition of a valid auction. Here, we consider an auction as valid if the minimum price, starting price and decrement are greater than zero.
Figure 52. Dutch Auction Rule Set 1.

The second variability point (bid validation) is modeled using three reaction rules triggered by the Bid Request event as shown in Figure 53. The Bid Request event is associated with the NewAuctionRequest message sent by the buyer participant. A bid is valid if it is greater than or equal to the minimum bid price. In Figure 53, reaction rule 3 is associated with the condition of a valid bid and the condition that it is the highest bid \((bid.auction.highestBid = null \text{ or } bid.value > bid.auction.highestBid.value)\), and it triggers an action to update the auction highest bid. Reaction rule 4 is associated with the condition of a valid bid, and it triggers an invoke action to the Accept Bid activity, followed by a BidResponse message sent to the buyer. Reaction rule 5 is associated with the negate condition of a valid bid, and it triggers an invoke action to the Reject Bid activity, followed by a FaultBidResponse message sent to the buyer.
The third variability point (determine auction winner) is modeled using four reaction rules triggered by the *Price Timeout* event as shown in Figure 54. In this rule set, we will loop through the list of bids to request the payment from the buyer associated to the winner bid, and notify the other buyers that the auction has ended.
In Figure 54, reaction rule 6 is associated with the condition that the highest bid is less than the current auction price (\(\text{bid.auction.highestBid.value < auction.price}\)) and it triggers an invoke action to the Reset Price Timer activity, and also an update action to decrement the auction current price (\(\text{bid.auction.price} = \text{bid.auction.price}@\text{pre} - \text{bid.auction.decrement}\)).

Reaction rule 7 is associated with the negate condition of remaining in the loop (i.e. \(\text{counter.value < bids.size}\)), and it triggers an invoke action to the Stop Counter activity.
Reaction rule 8 is associated with the condition of remaining in the loop, the condition that the current bid is the highest bid, and the bid is greater than the current auction price:
\[ \text{counter.value} < \text{bids.size} \text{ and } \text{bid} = \text{bid.auction.highestBid} \text{ and } \text{bid.auction.highestBid.value} \geq \text{auction.price} \]. It triggers an invoke action to the Request Payment activity followed by a RequestPayment message sent to the buyer, and it also triggers an update action to increment the counter by one, i.e. \( \text{counter.value} = \text{counter.value}@pre + 1 \). Reaction rule 9 is associated to the condition of remaining in the loop, the negate condition that the current bid is the highest one, and the bid is greater than the current auction price: \[ \text{counter.value} < \text{bids.size} \text{ and } \text{bid} \neq \text{bid.auction.highestBid} \text{ and } \text{bid.auction.highestBid.value} \geq \text{auction.price} \]. It triggers an invoke action to the End Auction activity followed by an EndAuction message sent to the buyer, and it also triggers an update action to increment the counter by one.

**Connecting Process Messages to Rules Events and Activities**

Here, we go back to the process diagram, attach Message data objects to Message Flow connections, and set the Structure Ref property of the Message element as the related rule event (AtomicEventExpression) or activity (ActionEventExpression) associated to the Message Type in the rules diagram.

**Interaction Model**

The interaction model for a Dutch auction is identical to the English auction as shown in Figure 50.

**UServ Product Derby**

This case study is based on the original UServ Product Derby Case Study (n.d., Business Rules Forum), a vehicle insurance service scenario, consisting of a business process for
vehicle insurance application and annual policy renewal, with business rules expressed in a natural English language. Giurca (2006) defines core ontological concepts (classes and variables) for the UServ Product Derby using R2ML 0.4 to target rule systems like JBossRules 3.0.6 and JenaRules 2.5.3. Here, we will model the UServ Product Derby using rBPMN language, thus representing business processes enriched with business rules.

Requirements Specification

The vehicle insurance scenario consists of two business processes: vehicle insurance application and annual policy renewal. When a vehicle application is received, it is validated, and if it is invalid, the applicant is notified to correct the information. If it is valid, then the policy is processed. The first step of the policy processing is to determine the policy eligibility score by using eligibility business rules: automobile eligibility, driver eligibility, and eligibility scoring. If the policy is not eligible, then the applicant is notified. If it is eligible, then the annual premium is determined, the application is accepted, and the policy owner is notified with the premium. The annual policy renewal is triggered once a year, before the policy expires, and it uses the same policy processing described above.

Process Design

The first step of the process design is to create a high level business process model using the rBPMN Editor. Figure 55 shows a business process in rBPMN consisting of two participants represented as pools: policy owner and financial service. The policy owner starts the process by sending an insurance application request to the financial service, represented by the Apply for Insurance task. The insurance application request message contains the vehicle(s) and driver(s) information. The financial service receives this message using a message catch intermediate event, and it checks if the insurance application information is
valid (*Validate Application* task). If it is valid, the *Process Policy* subprocess is executed. If it is invalid, the financial service sends a message to the policy owner participant indicating the information to be corrected using the *Notify to Correct Info* task, and the process ends.

![Diagram of UServ Product Derby high-level business process in rBPMN language.](image)

Figure 55. UServ Product Derby high-level business process in rBPMN language.

The *Process Policy* subprocess starts with the *Determine Eligibility* task. This task determines the car and driver eligibilities to calculate the policy eligibility score. The eligibility score determines if the policy application is eligible or not. If it is ineligible, the financial service sends a message to the policy owner notifying the ineligibility of the insurance application using the *Notify of Ineligibility* task. If it is eligible, the *Process Policy* subprocess continues by executing the *Determine Premium* task to calculate the policy premium, and then the *Notify with Premium* task that sends a message to the policy owner to inform about the new policy premium. The *Process Policy* subprocess ends and the main process of the *Financial Service* pool ends as well. The annual policy renewal is modeled by adding a *timer start event* to the *Financial Service* pool, and connecting it to the *Process Policy* subprocess.

140
Once defined the high level business process, we need to identify variability points and estimate their level of variability. In Figure 55, we identified two variability points: *Validate Application* and *Determine Eligibility* tasks. The first variability point, *Validate Application* task, represents a decision based on the verification that the insurance application has the required vehicle(s) and driver(s) information. However, the financial institution could extend this to verify if the policy owner has a bad insurance history, or the vehicle ownership is authentic, or the driver(s) has accident history, and make the decision to stop the insurance application process at this point. This results in a high frequency of changes and an internal source of change. Thus, the first variability point should be modeled with business rules. The second variability point, *Determine Eligibility* task, represents a decision to consider if the insurance application is eligible or not, based on the vehicle and driver eligibility. One of the factors used to determine vehicle eligibility is the theft probability. New vehicle models are released to market every year, and combined with police theft report updates will require frequently updates to the high theft probability auto list used to determine the vehicle eligibility. This results in a high frequency of changes and an internal source of change. Thus, the second variability point should also be modeled with business rules.

The final step of the process design is to determine the appropriated pattern for these variability points. For the first variability point, we will use the *Exclusive Choice* workflow pattern to model the orchestration since we have a condition (valid insurance application) that determines which activity will execute: the *Process Policy* subprocess (condition is true) or the *Notify to Correct Info* task (condition is false). For choreography modeling, we use a *Send/Receive* service interaction pattern, where the policy owner participant sends a request to the financial service participant, who then returns the correct info notification message.
For the second variability point, the calculated eligibility score determines which task will be executed next. In our high level design, we have two possible tasks: *Determine Premium* and *Notify of Ineligibility*. However, depending on the eligibility score, human intervention will be necessary to decide which of the above tasks to execute. If the eligibility score is low (< 100), then the policy is eligible, if it is high (> 250), then it is ineligible. If the eligibility score is between 100 and 250, then an authorized person in financial institution should review the policy application, and approve it by accepting the risks or not. This process is called underwriting. This external event was not considered during the high level design, but it should be considered to proper model a real insurance application scenario. We need to add a third possible choice, i.e. *Underwriting* task, after determining the eligibility. This task represents a human decision to consider the policy eligible or not. Here, we should use the *Multiple Choice* pattern which will execute one of the abovementioned tasks based on the calculated eligibility score. After the execution of the *Underwriting* task, we use an *Exclusive Choice* pattern to execute the *Determine Premium* task or the *Notify of Ineligibility* task depending on the underwriting decision. Thus, the second variability point will use the *Multiple Choice* and *Exclusive Choice* workflow patterns for the orchestration. For modeling the choreography, we will use the *Send* service interaction pattern, where the financial service sends an ineligibility notification message or a premium notification message to the policy owner. The following table summarizes the patterns used for these variability points.

Table 20. Patterns used for the UServ Product Derby Case Study.

<table>
<thead>
<tr>
<th>Variability Point</th>
<th>Workflow Pattern</th>
<th>Service Interaction Pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Exclusive Choice</td>
<td>Send/Receive</td>
</tr>
</tbody>
</table>
Before starting the data design using the rBPMN Editor, we need to replace the variable activities with rule gateways according to the selected patterns. For the first variability point, we use a rule gateway with a reaction rule set associated as in previous case studies (Figure 56). For the second variability point, we will use two rule gateways: one implementing the Multiple Choice pattern based on the eligibility score, and another implementing the Exclusive Choice pattern based on the underwriting decision. In the first rule gateway, we will use production rules. The eligibility score is calculated based on the car eligibility and driver eligibility, and they are calculated based on another car and driver attributes. This calculation is performed by applying a sequence of production rules to generate actions to update other attributes, until we determine the eligibility score. We could also use reaction rules, but since there is no message sent to start the calculation (triggering event), we decided to only use production rules. We have around 30 business rules (n.d., Business Rules Forum) to determine the eligibility score. In order to improve the readability of the models, they will be divided into three rule sets associated to the same rules gateway: RS2 Car Eligibility, RS3 Driver Eligibility, and RS4 Eligibility Score. For the second rule gateway, we also use production rules based on the underwriting decision, so we attach the RS5 Underwriting production rule set to it as shown in Figure 56.
Figure 56. UServ Product Derby business process in rBPMN language.
Data Design

During data design, we define the domain model (data objects and messages) using the rules editor of the rBPMN Editor. For the first variability point, we open the RS1 Application Validation rule set in the rules editor, and define the messages related to the Send/Receive service interaction pattern. The policy owner sends an InsuranceApplicationRequest message to the financial service, which should contain the vehicle and driver information. This message is annotated with message event type stereotype. If the insurance application is invalid, the financial service sends a CorrectInfoNotification message to the policy owner, which should contain the insurance application errors to be corrected, i.e. errors attribute as String. This message should be annotated with fault message event type stereotype.

For the second variability point, we open the RS2 Car Eligibility rule set to define the Car and Model classes as shown in Figure 57. The attributes are based on the business rules descriptions from Business Rules Forum (n.d.).

For the third variability point, we open the RS3 Driver Eligibility rule set and define the Driver class as shown in Figure 57.

For the fourth variability point, we open the RS4 Eligibility Score rule set to define the messages related to the Send pattern and data classes. The PremiumNotification message is sent to the policy owner if the insurance application has been accepted and it should contain the premium of the new policy. This message should be annotated with the message event type stereotype. The IneligibilityNotification message is sent to the policy owner when the insurance application is not accepted, and it should contain the reason information about the ineligibility. This message should be annotated with the fault message event type stereotype. We also define the Policy class as shown in Figure 57 and the Car and Driver class with
attributes that are relevant to this rule set. For the Car class, we use the autoEligibility attribute. For the Driver class, we use the isEligible, ageType and isHighRisk attributes.

Finally, we open the RS5 Underwriting rule set and define messages since they will also be used here, and create the UnderwritingDecision class with a Boolean attribute named underwrite to represent the underwriting decision.

```
Figure 57. UServ Product Derby vocabulary.
```

**Rule Design**

In the rule design, we start with the models created using the rBPMN Editor during the data design. At that stage, we only defined message types and data classes, and here we will add business rules.

For the first variability point, we open the RS1 Application Validation reaction rule set in the rules editor. We model it using three reaction rules triggered by the Insurance
Application event as shown in Figure 58. Here, we describe a basic validation rule that the insurance application policy should contain the vehicle and driver information. The InsuranceApplicationRequest message has a one-to-many association with the Car and Driver classes. Reaction rule 1 handles the scenario that the insurance application request has the complete information. It has two association conditions verifying that hasCar and hasDriver associations exist. It then triggers an invoke action to the Process Policy activity.

In the Process Policy activity property window, we set its Flow Node property as the Process Policy subprocess of the process diagram. Reaction rule 2 models the insurance application with missing vehicle information, while reaction rule 3 with missing driver information. Thus, reaction rule 2 has the negate association condition that hasCar association exist, and reaction rule 3 has the negate association condition that hasDriver association exist. Both reaction rules 2 and 3 trigger an invoke action to the Notify to Correct Info activity. The Flow Node property of the Notify to Correct Info activity should be the Notify to Correct Info task of the process diagram.

Figure 58. UServ Product Derby Rule Set 1.
Note that we only check if hasCar and hasDriver associations exist for the sake of simplicity. We could extend the rule diagram to check if the information in each Car and Driver object (attributes) is missing or consistent, thus requiring much more than just three reaction rules as shown in Figure 58.

For the second variability point, we open the RS2 Car Eligibility rule set in the rules editor. We use production rules to determine if the car theft probability is high, its potential occupant injury rating, and then we determine the auto eligibility as shown in Figure 59. The production rules from 4 to 7 determine the potential theft rating (potentialTheftRating). In production rule 4, for example, if the car model has a high theft probability, then the car theft potential rating is high. The production rules from 8 to 11 determine the potential of occupant injury rating (potentialOccupantInjuryRating). Based on the potentialTheftRating and potentialOccupantInjuryRating, production rules 12 to 14 determine the auto eligibility.
Figure 59. UServ Product Derby Rule Set 2.

For the *RS3 Driver Eligibility* rule set, we use production rules to determine if the driver is eligible, driver age type (young or senior), and if it is a high risk driver as shown in Figure 60. Production rules from 15 to 20 calculate driver age type and eligibility, while production rules from 21 to 23 determine if it is a high risk driver.
For the **RS4 Eligibility Score** rule set, we use production rules to determine the eligibility score. In Figure 61, production rules 24 and 25 increases the eligibility score based on the auto eligibility calculated in the **RS2 Car Eligibility** rule set. Production rules 26 to 28 increase the eligibility score based on the driver eligibility calculated in the **RS3 Driver Eligibility** rule set. Once the policy eligibility score is determined, production rules 29 to 31 will produce an invoke action to the appropriated activity based on the eligibility score calculated. For each activity, we should set their **Flow Node** property to the corresponding activity in the process diagram.
Finally, we open the RS5 Underwriting rule set in the rules editor. In Figure 62, we use two production rules, where production rule 32 is associated with the underwrite condition and it triggers an action to invoke the Determine Premium activity followed by a PremiumNotification message sent to the policy owner. Production rule 33 is associated with the negate condition of underwriting the policy, and it triggers an action to invoke the Notify of Ineligibility activity, followed by an IneligibilityNotification message sent to the policy owner. We also set the Flow Node properties of the rules to the corresponding activity in the process diagram.
Connecting Process Messages to Rules Events and Activities

Back to the process diagram, we attach Message data objects to Message Flow connections, and set the Structure Ref property of the Message element as the related rule event (AtomicEventExpression) or activity (ActionEventExpression) associated to the Message Type in the rules diagram.

Interaction Model

The interaction model for the User Product Derby is shown in Figure 63. For the first variability point, the Insurance Application message is represented as a Message Flow element with a Message Start Event attached, coming from the Policy Owner pool to the Financial Service pool. A rule gateway is attached to the Financial Service pool, where we place the RSI rule set. If the insurance application is invalid, the Notify to Correct Info
message is sent back to the policy owner; otherwise, the process flow goes on to the second rule gateway associated to the RS2, RS3 and RS4 rule sets. If the eligibility score is low, then the policy owner receives a Premium Notification message. If it is high, the policy owner receives an Ineligibility Notification message. If it is medium, then the third rule gateway is executed, which is associated with the RS5 rule set representing the underwriting decision rules as explained in previous sections. If the underwriting decision is true, then a Premium Notification message is sent to the policy owner; otherwise, an Ineligibility Notification message is sent instead. The business rules defined in all rule sets of the interaction model are the same as those defined in the previous Rule Design section.

Figure 63. UServ Product Derby interaction model.

In this chapter, we presented three case studies showing how to use the rBPMN Editor and how to apply the methodology for developing rule-enabled SOAs described in Chapter II. Next chapter will discuss the evaluation and analysis of the rBPMN Editor.
CHAPTER VI

EVALUATION AND ANALYSIS

This chapter discusses the evaluation and analysis of the rBPMN Editor, starting with the development framework used; aspects of the implementation including vocabulary support, orchestration and choreography modeling, supported business process patterns and how their implementation differ from the models proposed by Milanović (2010); and comparison of the rBPMN Editor and the Strelka editor.

Development Approach

The Eclipse GMF framework enabled the rapid application development of the rBPMN Editor. Its MDE approach let us focus on the problem being solved rather than the details of Eclipse plug-ins and user interface development. A few changes in the rBPMN metamodel were necessary (Chapter IV) to overcome some GMF limitations, but these changes did not compromise the rBPMN language semantics. The documentation is limited to text books, i.e. Gronback (2009) and Steinberg (2008), online documentation, tutorials and blog posts, e.g. Eclipse Foundation (2011j); and we leveraged the GMF discussion forum (Eclipse Foundation, 2011k) to supplement this limited documentation. The GMF framework is at a mature development stage, but this project has not been so active in the past few years, which indicates that the GMP future focus will be on the new Graphiti framework. Although having limited documentation and facing the learning curve of this technology, we would not be able to implement the rBPMN Editor with the same functionality during this project timeframe without leveraging the GMF framework.
rBPMN Editor Implementation

The rBPMN Editor supports the creation of vocabulary elements, such as classes and message event types, in both process and rules editors as presented in Chapter IV and Chapter V. This allows defining data structures for rules conditions and conclusions, and message types, and thus resulting in rBPMN models with complete rules and message descriptions that enables the generation of complete service descriptions. Although vocabulary definition is supported, the rBPMN Editor does not support external vocabulary in the current implementation, i.e. an rBPMN model being able to refer another rBPMN model and use its vocabulary elements. An external vocabulary could be used to define a set of primitive types (String, Integer, Boolean, etc.) that could be used to define attributes for classes and message event types. These primitive types do not need to be graphically represented in the diagrams, but we should be able to use them as the Type property of classes and message event types. In the implementation of the case studies in Chapter V, classes were created for each primitive type in order to define attributes of primitive types.

The current implementation only supports one graphical element for each model element, but ideally it should support multiple graphical elements for the same model element. This is particularly important for shared vocabulary when a rule set can use a class as a condition, and the same class is also used in another rule set of the same rule gateway. An example of this scenario is in the UServ Product Derby case study, where the Driver class is defined in the RS3 rule set (Figure 60) to determine the driver eligibility, and it is also defined in the RS4 rule set (Figure 61) as conditions to production rules to determine the policy eligibility score. This results in redundant information since the underlying rBPMN model contains more than one definition of the same class. Ideally, the Driver class could be
defined only once in the rBPMN model and be displayed in many places as needed in the diagrams. One option would be allowing dragging elements from the model in the editor’s *Project Explorer* and dropping them on the diagram as needed. The figure below shows the *Driver* class under the *RS3* rule set being added to the *RS4* rule set diagram. Due to its implementation effort and project time constraints, this solution has not been implemented at this time, but it will be considered in future work.

Figure 64. Creating more than one graphical element for the same model element.

The rBPMN Editor diagram partitioning enables the creation of separate diagrams containing business rules details for the same rBPMN model. This separation allows the dynamic update of business process logic without changing the business process itself and also avoids business rules details to interfere with the overall visualization of the business process, thus improving its readability. The readability of business rules is also improved by separating business rules into multiple diagrams when the number of rules is too high. Rule gateways allow multiple rule sets to be connected so that different type of rules are supported, but it also allows multiple rule sets of the same rule type to improve readability. In
the UServ Product Derby case study, there are 28 production rules to determine the policy eligibility, and these rules are placed into three separated production rule sets instead of a single one. The RS2 rule set (Figure 59) shows 11 production rules to determine the car eligibility with a layout focusing on its readability, where the top-left part contains rules to determine the potentialTheftRating, the bottom-left area with rules to determine the potentialOccupantInjuryRating, and the top-right area with the rules to determine the autoEligibility.

In the following sections, we evaluate other aspects of the rBPMN Editor by analyzing the business process patterns supported and how their implementation differs from the models proposed by Milanović (2010).

Business Process Patterns Support

In Chapter 2, we described all business process patterns supported by the rBPMN language. For the evaluation of the rBPMN Editor, models of the patterns, as proposed in (Milanović, 2010), were implemented to verify if they are fully supported by the rBPMN Editor.

The rBPMN Editor supports all Workflow patterns described in Table 8, except for the Interleaved Parallel Routing pattern since this pattern is not supported by the rBPMN language (Milanović et al., 2009). The models for the Workflow patterns were implemented based on the models proposed by Milanović (2010) and are presented in Appendix A – Workflow Patterns. For all Workflow patterns, the rBPMN Editor supported the modeling of process flows, business vocabularies, and business rules to dynamically change the sequence flow at runtime as supported by the rBPMN language. Thus, this implementation confirms
that all Workflow patterns supported by the rBPMN language (Milanović, 2010) are also supported by the rBPMN Editor. Here, we only discuss the Multiple Instances with no a Priori Runtime Knowledge pattern due to space limitation. This pattern is chosen since it is not supported by all languages, i.e., only by rBPMN and AORML (Table 8), and due its implementation complexity where reaction rules are used to dynamically change the number of required instances and to define complex conditions to invoke activities. In this pattern, the number of instances of a given activity is only known immediately before instances of that activity type need to be created (Milanović, 2010). In its implementation, the process diagram (Figure 84) is similar to the model proposed by Milanović (2010), but the reaction rule set diagram (Figure 85) contains more details. These rules define that two activities will be executed in parallel multiple times (count attribute) and one these activities (Activity 2) might increase the count attribute. The parallel execution of these activities is modeled using the reaction rule 1 that triggers a parallel complex event containing these two activities, if the running counter has not reach its limit (negate condition of $counter = count$). The increase of the count attribute by Activity 2 is modeled using an Increase count event that triggers a reaction rule (id: 3) to update the count attribute by increasing it by one. Another reaction rule (id: 2) is used to stop the multiple execution of these activities, i.e. if the running counter reaches its limit (count), then the End activity is invoked and the process resumes. The rBPMN Editor enabled modeling the Multiple Instances with no a Priori Runtime Knowledge pattern by supporting complex events (e.g. parallel event), and by using an extra event that is raised when the number of instances of these activities changes during runtime. The traceability between rule and process flow elements is supported through Flow Node property of the rule activity.
The rBPMN Editor fully supports the creation of *Message Exchange* patterns. Appendix B – Message Exchange Patterns contains the *Message Exchange* pattern models created with the rBPMN Editor, and they are similar to the models proposed by Milanović (2010). The rBPMN Editor provides the necessary vocabulary elements to define message types, including associated data objects, and enables the traceability between these structural elements to message elements in the process diagram (*Structure Ref* property).

The rBPMN Editor fully supports eleven of the thirteen *Service Interaction* patterns described in Table 7. The *Dynamic Routing* pattern is partially supported since the rBPMN language partially supports it. The *Atomic Multicast Notification* pattern is not supported since the rBPMN language does not support it. Appendix C contains all *Service Interaction* patterns implemented using the rBPMN Editor and they are based on the models proposed by Milanović (2010). Thus, this implementation confirms that all *Service Interaction* patterns supported by the rBPMN language (Milanović, 2010) are also supported by the rBPMN Editor. For *Service Interaction* patterns that contains multiple participants, such as the *One to Many Send* pattern implemented for the English Auction case study (see rule gateway/rule set 3 in Figure 45), the rBPMN reaction rule editor provides all the necessary elements to define the business rules (Figure 49) to decide when interactions with multiple parties ends (*Stop activity*), or whether they are successfully completed (*Request Payment* activity) or not (*End Auction* activity). The implementation of *Service Interaction* patterns in interaction models has a small difference between models proposed by Milanović (2010) and those models created using the rBPMN Editor. In the models proposed by Milanović (2010), rules conditions and vocabulary can be placed anywhere in the diagram, while using the rBPMN Editor they are placed inside a rule set, and the rule set must be inside a pool or a subprocess.
Thus, rule sets are placed inside the pool that is connected to the rule set associated rule gateway (see Figure 65).

![Diagram](image)

**Figure 65.** The Send/Receive pattern interaction model: (a) proposed by Milanović(2010); (b) implemented using the rBPMN Editor.

There were a couple of minor layout issues observed while creating interaction models that will be later addressed: pools cannot be placed side by side which could be helpful in some interaction models, and message start and intermediate catch events are attached to the message flow at a fixed position (middle), but ideally they could be attached at any position along the message flow, thus improving the visualization of the workflow in the interaction model as shown in Figure 66.
Here, we discuss the Dynamic Routing pattern since it is the only pattern partially supported by the language and rBPMN Editor, and also because it contains the highest number of pools thus making these issues more evident. Figure 67 shows the Dynamic Routing pattern implemented using the rBPMN Editor which supports all the same elements of the model proposed by Milanović (2010), but its readability could be improved if a same layout of Figure 66 could be implemented, i.e. place Pool 2 and Pool 1 side by side and place message start and intermediate catch events at the message flows so that facilitates the visualization of the workflow.
In all models implemented for the Service Interaction patterns and for the case studies (Chapter V), interaction models were created based on their respective interconnected behavior interface models, thus, supporting the generation of choreographies from orchestrations. The rBPMN language supports the compliance between orchestrations and choreographies, where rules defined in orchestrations can also be used in choreographies (Milanović, 2010). Thus, when creating these interaction models (choreography), their rules diagrams were created identical to the rules diagrams previous created for the interconnected interface behavior models. This result in duplication of work, and an improvement for the rBPMN Editor would be to allow multiple process diagrams to share business rules diagrams.
The rBPMN Editor fully support all Business Rules Patterns for Agile Business Processes described in Table 9. Appendix D – Business Rules for Agile Business Process Patterns contains all patterns implemented using the rBPMN Editor and they are based on the models proposed by Milanović (2010). Thus, this implementation confirms that all Business Rules Patterns for Agile Business Processes supported by the rBPMN language (Milanović, 2010) are also supported by the rBPMN Editor. Here, we discuss all patterns of this category since the main focus of this project is on the rBPMN rules editors and their integration with the business process editor.

In the Control flow decision patterns group, the Decision Logic Abstraction pattern implementation separates derivation and production rules diagrams (Figure 135 and Figure 136) from the main process diagram (Figure 134) thus enabling decision logic update during runtime without redeploying the business process. In the Decision Node to Business Rule Binding pattern implementation (Figure 137), rule sets are placed on a separate pool (Rule Repository), so that they can be reused by different rule gateways within different business processes. Since rules sets are shared between multiple rule gateways, the activities within the production rule set cannot be connected to specific tasks in the business process through the Flow Node attribute. Instead, the Rule attribute of the outgoing Sequence Flow elements of the rule gateway can be set to the corresponding production rule, thus providing the mapping between the process flows to the rules. In the Decision with Flexible Input Data pattern implementation (Figure 138), a task before the rule gateway is used to fill up a process instance context data object that is used to populate the input data of the rule sets attached to the rule gateway. In the Decision Flexible Output pattern implementation, the process diagram created (Figure 139) is similar to the model proposed by Milanović (2010),
but the production rule diagram (Figure 140) is implemented in details where production rules are used to decide which output branch of the rule gateway to follow. The production rule that represents the decision to follow the dynamic branch (id: 3) uses an update action to set the service\textit{Name} attribute in the \textit{Result} data object. This shared vocabulary is used by the \textit{Lookup Service} task in the process diagram to assign a custom task to be executed right after it, thus allowing the production rule to choose a custom task at runtime that was not defined in the process flow during design.

In the \textit{Data constraints} patterns group, the implementation of the \textit{Constraints at Predefined Checkpoint} pattern (Figure 141) pattern uses two integrity rule diagrams (Figure 142 and Figure 143) to define the constraints for the corresponding rule gateways, thus checking the validity on the input and output data of the \textit{Task} activity. In the \textit{Constraints at Multiple Checkpoints} pattern, a single constraint is enforced at multiple places and its implementation achieve this by attaching multiple gateways to the same integrity rule set (Figure 144). In the \textit{Constraint Enforced by External Data Context} pattern implementation, the process diagram (Figure 147) is similar to the model proposed by Milanović (2010) where the \textit{External process} pool acts as an external business process context upon which constraints are enforced (Milanović, 2010). In this implementation, two rule sets (reaction and integrity) are attached to the rule gateway of the \textit{External process} pool. The reaction rule diagram (Figure 148) demonstrates how to pass the process instance context data to the external process with the rBPMN Editor. The \textit{Receive Data} event represents the message sent to the \textit{External process} pool in the process diagram, while the \textit{CheckConstraintsRequest} element defines the message type that is associated to the process instance context object. This association is a condition of the reaction rule that invokes an update action to update
some attribute(s) based on the process instance context, that later is be used for constraints checking in the integrity rule set diagram (Figure 149).

In the *Dynamic Business Process Composition* patterns group, the *Business Rule-Based Subprocess Selection* pattern implementation is similar to implementation of the *Decision Flexible Output* pattern above mentioned. The process in the main diagram (Figure 150) selects a subprocess instead of an activity, and the production rule set diagram (Figure 151) invokes an update action to set the name of the subprocess instead of the service name. The *Business Rule-Based Process Composition* pattern implementation is similar to the model proposed by Milanović (2010). In the process diagram (Figure 152), two rule sets are added (derivation and reaction). The derivation rule diagram (Figure 153) shows a fact being derived by the derivation rule, and the production rule diagram (Figure 154) shows two production rules with conditions based on that derived fact. If a condition is satisfied, the production rule invokes the activity to start the corresponding process. The *Flow Node* property of each invoked activity is set to the *Start Event* element of the corresponding process being invoked. That is, the rBPMN Editor enables the modeling of production and reaction rules to invoke business processes by associating rules activities to process start events through the *Flow Node* property.

**Comparison between Strelka and rBPMN Editor**

In this section, we compare Strelka with the rBPMN Editor. Strelka has two different versions: Strelka 0.2 for Fujaba, a plug-in for the Fujaba UML tool, and Strelka 0.3 for Eclipse, a plug-in for Eclipse IDE that is also implemented using GMF (Strelka, 2006). As
mentioned in Chapter III, we leverage the work of Strelka for Eclipse by reusing part of their GMF graphical definition model and extending it.

Our implementation differs from Strelka (Strelka, 2006) in both implementation and functionality aspects. Both versions of Strelka use URML and UML metamodels as the domain model, and rBPMN rules editor uses the R2ML metamodel. Thus, their mappings are between URML/UML metamodel elements and URML/UML graphical elements, while ours are between R2ML metamodel elements and URML/UML graphical elements. Although both Strelka for Eclipse and the rBPMN Editor are implemented using GMF, Strelka for Eclipse is a standalone Eclipse-based application while the rBPMN Editor runs as Eclipse plug-in. As to functionality, Strelka for Eclipse supports only derivation and production rules in the same modeling editor; Strelka for Fujaba supports derivation, production and reaction rules; and the rBPMN Editor supports all four types of rules but in separated editors. The rBPMN Editor lacks of supporting rule interchange, Strelka for Fujaba supports native serialization of URML models in R2ML XML (Ribarić, 2008), and Strelka for Eclipse supports saving rules as R2ML XML and also translating rules to Jess, Jena2, JBoss and F-Logic directly from the editor (Strelka, 2006). This built-in rules translation feature uses an external Web service that seems to be not supported anymore at the time of this writing. In addition to business rules support, the rBPMN Editor supports business process modeling using BPMN2 notation and it also supports the integration between business process and business rules, while both Strelka versions only support business rules. Finally, Strelka for Fujaba is the only editor that supports R2ML to WSDL and WSDL to R2ML transformations (Ribarić, 2008). The following table summarizes the comparison between these editors.

Table 21. Strelka and rBPMN Editor Comparison.
<table>
<thead>
<tr>
<th>Category</th>
<th>Strelka 0.2 for Fujaba</th>
<th>Strelka 0.3 for Eclipse</th>
<th>rBPMN Editor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rules Domain Model</td>
<td>URML and UML metamodels</td>
<td>URML and UML metamodels</td>
<td>R2ML metamodel</td>
</tr>
<tr>
<td>Rules Graphical Representation</td>
<td>URML</td>
<td>URML</td>
<td>URML extended</td>
</tr>
<tr>
<td>Application Type</td>
<td>Fujaba plug-in</td>
<td>Standalone Eclipse-based</td>
<td>Eclipse plug-in</td>
</tr>
<tr>
<td>Business Process</td>
<td>-</td>
<td>-</td>
<td>BPMN2</td>
</tr>
<tr>
<td>Rules and Business Process</td>
<td>-</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td>Process Integration</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Derivation Rules</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Integrity Rules</td>
<td>-</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td>Production Rules</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Reaction Rules</td>
<td>+</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td>R2ML XML Transformation</td>
<td>+</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>Built-in Rule Interchange</td>
<td>Jess</td>
<td>+/-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>Jena 2</td>
<td>+/-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>JBoss</td>
<td>+/-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>F-Logic</td>
<td>+/-</td>
<td>-</td>
</tr>
<tr>
<td>R2ML to WSDL Transformation</td>
<td>+</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>WSDL to R2ML Transformation</td>
<td>+</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Note. Features are indicated with “+” if supported, with “+/-” it partially supported, and with “-” if not supported. Although Strelka for Eclipse documentation describes its support for
rule interchange, it has been marked as partially supported since this functionality was not working at the time of this writing.

For derivation rules, the same conditions and conclusions elements are supported. For production rules, Strelka supports the same actions except for *invoke activity action* and *activity*. Strelka also lacks of support for post conditions. For vocabulary, the rBPMN Editor does not support interface, interface realization, and package. UML associations in the rBPMN Editor are limited in comparison to Strelka UML associations, since rBPMN Editor associations are only unidirectional while Strelka supports bidirectional. This limitation is a result of mapping UML to R2ML metamodel vocabulary that does not have all elements and attributes needed by UML. Although the R2ML metamodel could be changed to provide all information needed by UML, only necessary changes were made in the R2ML metamodel not to break its semantics. Strelka supports primitive types (*Boolean*, *Float*, *Integer* and *String*) in class attributes while the rBPMN Editor does not, and thus requiring primitive types to be created as classes. In a summary, the rBPMN Editor provides better and complete support for rules, business processes and integration of rules and business processes, while Strelka provides better vocabulary and rule interchange support. The following table contains a complete comparison between Strelka and rBPMN Editor in terms of rules and vocabulary supported.

Table 22. Strelka and rBPMN Editor Comparison - Rules and Vocabulary Support.

<table>
<thead>
<tr>
<th>Element</th>
<th>Strelka 0.2 for Fujaba</th>
<th>Strelka 0.3 for Eclipse</th>
<th>rBPMN Editor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Derivation Rule</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Integrity Rule</td>
<td>-</td>
<td>-</td>
<td>+</td>
</tr>
</tbody>
</table>

168
<table>
<thead>
<tr>
<th>Category</th>
<th>+</th>
<th>-</th>
<th>+</th>
<th>+</th>
</tr>
</thead>
<tbody>
<tr>
<td>Production Rule</td>
<td></td>
<td></td>
<td>+</td>
<td></td>
</tr>
<tr>
<td>Reaction Rule</td>
<td></td>
<td>+</td>
<td></td>
<td>+</td>
</tr>
<tr>
<td>Association Condition</td>
<td>+</td>
<td>+</td>
<td></td>
<td>+</td>
</tr>
<tr>
<td>Classification Condition</td>
<td>+</td>
<td>+</td>
<td></td>
<td>+</td>
</tr>
<tr>
<td>Property Condition</td>
<td>+</td>
<td>+</td>
<td></td>
<td>+</td>
</tr>
<tr>
<td>Post Condition</td>
<td>-</td>
<td>-</td>
<td></td>
<td>+</td>
</tr>
<tr>
<td>Association Conclusion</td>
<td>+</td>
<td>+</td>
<td></td>
<td>+</td>
</tr>
<tr>
<td>Classification Conclusion</td>
<td>+</td>
<td>+</td>
<td></td>
<td>+</td>
</tr>
<tr>
<td>Property Conclusion</td>
<td>+</td>
<td>+</td>
<td></td>
<td>+</td>
</tr>
<tr>
<td>Attribution Conclusion</td>
<td>-</td>
<td>-</td>
<td></td>
<td>+</td>
</tr>
<tr>
<td>OCL Invariant</td>
<td>+</td>
<td>-</td>
<td></td>
<td>+</td>
</tr>
<tr>
<td>Invariant Connection</td>
<td>-</td>
<td>-</td>
<td></td>
<td>+</td>
</tr>
<tr>
<td>Constraint Class Connection</td>
<td>-</td>
<td>-</td>
<td></td>
<td>+</td>
</tr>
<tr>
<td>Assert Action</td>
<td>+</td>
<td>+</td>
<td></td>
<td>+</td>
</tr>
<tr>
<td>Retract Action</td>
<td>+</td>
<td>+</td>
<td></td>
<td>+</td>
</tr>
<tr>
<td>Update Action</td>
<td>+</td>
<td>+</td>
<td></td>
<td>+</td>
</tr>
<tr>
<td>Invoke Action</td>
<td>+</td>
<td>+</td>
<td></td>
<td>+</td>
</tr>
<tr>
<td>Invoke Activity Action</td>
<td>-</td>
<td>-</td>
<td></td>
<td>+</td>
</tr>
<tr>
<td>Activity</td>
<td>-</td>
<td>-</td>
<td></td>
<td>+</td>
</tr>
<tr>
<td>Triggering Event</td>
<td>+</td>
<td>-</td>
<td></td>
<td>-</td>
</tr>
<tr>
<td>Triggered Event</td>
<td>+</td>
<td>-</td>
<td></td>
<td>-</td>
</tr>
<tr>
<td>Event</td>
<td>+</td>
<td>-</td>
<td></td>
<td>+</td>
</tr>
<tr>
<td>Choice Event</td>
<td>-</td>
<td>-</td>
<td></td>
<td>+</td>
</tr>
<tr>
<td>Parallel Event</td>
<td>-</td>
<td>-</td>
<td></td>
<td>+</td>
</tr>
<tr>
<td>Element</td>
<td>+</td>
<td>-</td>
<td>+</td>
<td></td>
</tr>
<tr>
<td>---------------------------------</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td></td>
</tr>
<tr>
<td>Sequence Event</td>
<td></td>
<td></td>
<td>+</td>
<td></td>
</tr>
<tr>
<td>And/Not Event</td>
<td></td>
<td></td>
<td>+</td>
<td></td>
</tr>
<tr>
<td>Message Type</td>
<td></td>
<td></td>
<td>+</td>
<td></td>
</tr>
<tr>
<td>Message Type Connection</td>
<td></td>
<td></td>
<td>+</td>
<td></td>
</tr>
<tr>
<td>Class</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td></td>
</tr>
<tr>
<td>Attribute</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td></td>
</tr>
<tr>
<td>Operation</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td></td>
</tr>
<tr>
<td>Association</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td></td>
</tr>
<tr>
<td>DataType</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td></td>
</tr>
<tr>
<td>Enumeration</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td></td>
</tr>
<tr>
<td>Enumeration Literal</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td></td>
</tr>
<tr>
<td>Generalization</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td></td>
</tr>
<tr>
<td>Interface</td>
<td>+</td>
<td>+</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>Interface Realization</td>
<td>+</td>
<td>+</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>Package</td>
<td>+</td>
<td>+</td>
<td>-</td>
<td></td>
</tr>
</tbody>
</table>

*Note.* Elements are indicated with “+” if supported or with “-” if not supported. For the sake of simplicity, this table does not include business process elements (i.e., BPMN2) that are fully supported by the rBPMN Editor and not supported by Strelka editors.
CHAPTER VII
CONCLUSION

In this work, we designed and implemented a modeling tool for the new conceptually defined rBPMN language. Our goal is to provide tooling support for the rBPMN language, so that rBPMN models can be realized and the language be also evaluated. We use an MDE approach to develop the rBPMN Editor using GMF and EMF frameworks resulting in a set of plug-ins to be installed and executed in Eclipse IDE.

Our solution consists of a multi-diagram editor where a main diagram represents the business process using a BPMN extended notation, and sub-diagrams represent business rules and vocabularies using an URML extended notation. This multi-diagram approach supports the integration of business rules into business processes, but it avoids that rules details interfere with the overall business process visualization.

The rBPMN Editor was evaluated with three cases studies using the methodology for developing rule-enabled SOA. In addition, rBPMN diagrams of the business process patterns supported by the rBPMN language were also implemented, to evaluate that all patterns could be realized using this tool and to determine the differences between the models proposed by Milanović (2010) and those implemented using the rBPMN Editor. The rBPMN Editor supports the creation of both interaction and interconnected behavioral models to focus on the choreography and orchestration aspects. We showed how service orchestrations can be modeled in rBPMN, and the sequence flow be controlled by decisions defined as business rules in separate URML diagrams. This diagram separation allows dynamic updates of
business rules without changing the business process itself. In situations where the number of business rules is high for a single variability point of the business process, the readability of these rules could be significantly improved by separating them into multiple rule sets associated to the same rule gateway as demonstrated in the User Product Derby case study. The rBPMN Editor provides the necessary vocabulary elements to define data structures for rules conditions and conclusions, and message types. Thus, rBPMN models are created with complete descriptions of rules and messages that enables the generation of complete service descriptions. In all case studies and Service Interaction patterns, we demonstrate the creation of interaction models based on their respective interconnected behavioral models, and that they use the same business rules, thus showing the compliance between orchestrations and choreographies in the rBPMN language. We also demonstrate the creation of the conditions and constraints of these interactions by defining rule sets using URML (rule diagram) and associating them to rule gateways in the process diagram. The rBPMN Editor supports traceability between elements of the process and rules diagrams: i) a message in the process diagram can be associated with a message event type in the rules diagram to define its data structure; ii) a rule activity can be associated to a flow node element in the process diagram, e.g. a process activity or a start event as used in the Business Rule-Based Process Composition pattern implementation to dynamically compose a business process from business process fragments.

In addition to the tooling support for the rBPMN language, this project also contributes by defining the graphical representation of some R2ML elements that are not supported by URML such as events, complex events, and message event types. In comparison with the URML editor Strelka, the rBPMN Editor supports all four types of rules
and rule elements such as activity (*ActionEventExpression*) and *Invoke Activity Action*, business process modeling support (i.e., BPMN2), and business process and business rules integration, but it does not support rule interchange or WSDL transformation as Strelka versions do.

The future work involves addressing current limitations of the rBPMN Editor. The vocabulary support can be enhanced by supporting external vocabulary, so that rBPMN models can refer to other models and use external vocabulary elements. This also includes having an external vocabulary of primitive types (*String, Double, Boolean*, etc.) to be referenced by all rBPMN models so that they can be used to define attributes of classes and message event types. Another vocabulary enhancement would be supporting multiple graphical elements representing the same model element, where a class defined in a rule set diagram could be displayed on other rule set diagrams without having to duplicate its definition in the rBPMN model. Another reusability improvement is allowing multiple process diagrams to share same rule sets, which is desirable in scenarios showing different aspects (choreography and orchestration) of the same business process by creating two models (interaction and interconnected behavioral) that use the same set of business rules. We also would like to improve the readability of choreography models by fixing minor layout issues detected such as positioning pools side by side and attaching messages within any place along message flows; and improve the usability of rule sets within the process diagram by allowing expanding and collapsing their content. The rBPMN Editor could also support rule interchange by enabling rBPMN model transformations to R2ML XML and to the OMG standard RIF. Another desirable functionality is to generate complete service descriptions from rBPMN models by supporting model transformations to BPEL. We also
need to conduct empirical studies to investigate the quality characteristics (e.g.,
understandability, changeability, or analyzability) of the rBPMN Editor in different business
process modeling tasks. As an on-going effort, Petri net-based semantics for rBPMN are
being developed enabling the formal verification of rule-enhanced business process (e.g.,
deadlock detection) (Milanović et al., 2011), thus being necessary to support rBPMN models
to Petri nets transformations. Finally, the rBPMN Editor will be available as open source so
that this work can be shared and collaborated with the community.
REFERENCES


APPENDIX A – WORKFLOW PATTERNS

Figure 68. The Sequence pattern using the rBPMN Editor.

Figure 69. The Parallel Split pattern using the rBPMN Editor.

Figure 70. The Synchronization pattern using the rBPMN Editor.

Figure 71. The Exclusive Choice pattern using the rBPMN Editor.
Figure 72. The Simple Merge pattern using the rBPMN Editor.

Figure 73. The Multiple Choice pattern using the rBPMN Editor.
Figure 74. The Structured Synchronizing Merge pattern using the rBPMN Editor.

Figure 75. The Multiple Merge pattern using the rBPMN Editor.
Figure 76. The Discriminator pattern using the rBPMN Editor.

Figure 77. The Arbitrary Cycles pattern using the rBPMN Editor.
Figure 78. The Implicit Termination pattern using the rBPMN Editor.

Figure 79. The Multiple Instances without Synchronization pattern using the rBPMN Editor.
Figure 80. The Multiple Instances with a Priori Known Design Time Knowledge pattern using the rBPMN Editor (process diagram).

Figure 81. The Multiple Instances with a Priori Known Design Time Knowledge pattern using the rBPMN Editor (rule set diagram).
Figure 82. The Multiple Instances with a Priori Known Runtime Knowledge pattern using the rBPMN Editor (process diagram).

Figure 83. The Multiple Instances with a Priori Known Runtime Knowledge pattern using the rBPMN Editor (rule set diagram).
Figure 84. The Multiple Instances with no a Priori Runtime Knowledge pattern using the rBPMN Editor (process diagram).

Figure 85. The Multiple Instances with no a Priori Runtime Knowledge pattern using the rBPMN Editor (rule set diagram).
Figure 86. The Deferred Choice pattern using the rBPMN Editor.

Figure 87. The Milestone pattern using the rBPMN Editor.

Figure 88. The Cancel Activity pattern using the rBPMN Editor.
Figure 89. The Cancel Case pattern using the rBPMN Editor.
APPENDIX B – MESSAGE EXCHANGE PATTERNS

Figure 90. The In-Only pattern using the rBPMN Editor (process diagram).

Figure 91. The In-Only pattern using the rBPMN Editor (rule set 1 diagram).
Figure 92. The Robust In-Only pattern using the rBPMN Editor (process diagram).

Figure 93. The Robust In-Only pattern using the rBPMN Editor (rule set 1 diagram).
Figure 94. The In-Out pattern (with fault) using the rBPMN Editor (process diagram).

Figure 95. The In-Out pattern (with fault) using the rBPMN Editor (rule set 1 diagram).
Figure 96. The In-Optional-Out pattern using the rBPMN Editor (process diagram).

Figure 97. The In-Optional-Out pattern using the rBPMN Editor (rule set 1 diagram).
Figure 98. The Out-Only pattern using the rBPMN Editor (process diagram).

Figure 99. The Out-Only pattern using the rBPMN Editor (rule set 1 diagram).

Figure 100. The Out-Only pattern using the rBPMN Editor (rule set 2 diagram).
Figure 101. The Robust Out-Only pattern using the rBPMN Editor (process diagram).

Figure 102. The Robust Out-Only pattern using the rBPMN Editor (rule set 1 diagram).

Figure 103. The Robust Out-Only pattern using the rBPMN Editor (rule set 2 diagram).
Figure 104. The Out-In pattern using the rBPMN Editor (process diagram).

Figure 105. The Out-In pattern using the rBPMN Editor (rule set 1 diagram).

Figure 106. The Out-In pattern using the rBPMN Editor (rule set 2 diagram).
Figure 107. The Out-Optional-In pattern using the rBPMN Editor (process diagram).

Figure 108. The Out-Optional-In pattern using the rBPMN Editor (rule set 1 diagram).

Figure 109. The Out-Optional-In pattern using the rBPMN Editor (rule set 2 diagram).
APPENDIX C – SERVICE INTERACTION PATTERNS

Figure 110. The Send pattern using the rBPMN Editor.

Figure 111. The Send pattern using the rBPMN Editor (interaction model).
Figure 112. The Receive pattern using the rBPMN Editor.

Figure 113. The Receive pattern using the rBPMN Editor (interaction model).
Figure 114. The Send/Receive pattern using the rBPMN Editor.

Figure 115. The Send/Receive pattern using the rBPMN Editor (interaction model).
Figure 116. The Racing Incoming Messages pattern using the rBPMN Editor.

Figure 117. The Racing Incoming Messages pattern using the rBPMN Editor (interaction model).
Figure 118. The One to Many Send pattern using the rBPMN Editor.

Figure 119. The One to Many Send pattern using the rBPMN Editor (interaction model).
Figure 120. The One from Many Receive pattern using the rBPMN Editor.

Figure 121. The One from Many Receive pattern using the rBPMN Editor (interaction model).
Figure 122. The One to Many Send/Receive pattern using the rBPMN Editor.
Figure 123. The One to Many Send/Receive pattern using the rBPMN Editor (interaction model).
Figure 124. The Multi-Responses pattern using the rBPMN Editor.

Figure 125. The Multi-Responses pattern using the rBPMN Editor (interaction model).
Figure 126. The Contingent Requests pattern using the rBPMN Editor.

Figure 127. The Contingent Requests pattern using the rBPMN Editor (interaction model).
Figure 128. The Request with Referral pattern using the rBPMN Editor.
Figure 129. The Request with Referral pattern using the rBPMN Editor (interaction model).
Figure 130. The Relayed Request pattern using the rBPMN Editor.
Figure 131. The Relayed Request pattern using the rBPMN Editor (interaction model).
Figure 132. The Dynamic Routing pattern using the rBPMN Editor.
Figure 133. The Dynamic Routing pattern using the rBPMN Editor (interaction model).
APPENDIX D – BUSINESS RULES FOR AGILE BUSINESS PROCESS

PATTERNS

Figure 134. The Decision Logic Abstraction pattern using the rBPMN Editor (process diagram).

Figure 135. The Decision Logic Abstraction pattern using the rBPMN Editor (derivation rule set diagram).

Figure 136. The Decision Logic Abstraction pattern using the rBPMN Editor (production rule set diagram).
Figure 137. The Decision Node to Business Rule Binding pattern using the rBPMN Editor.

Figure 138. The Decision with Flexible Input Data pattern using the rBPMN Editor.
Figure 139. The Decision Flexible Output pattern using the rBPMN Editor (process diagram).

Figure 140. The Decision Flexible Output pattern using the rBPMN Editor (production rule set diagram).
Figure 141. The Constraints at Predefined Checkpoint pattern using the rBPMN Editor (process diagram).

Figure 142. The Constraints at Predefined Checkpoint pattern using the rBPMN Editor (integrity rule set 1 diagram).

Figure 143. The Constraints at Predefined Checkpoint pattern using the rBPMN Editor (integrity rule set 2 diagram).
Figure 144. The Constraints at Multiple Checkpoints pattern using the rBPMN Editor (process diagram).

Figure 145. The Constraints at Multiple Checkpoints pattern using the rBPMN Editor (integrity rule set 1 diagram).

Figure 146. The Constraints at Multiple Checkpoints pattern using the rBPMN Editor (integrity rule set 2 diagram).
Figure 147. The Constraints Enforced by External Data pattern using the rBPMN Editor (process diagram).

Figure 148. The Constraints Enforced by External Data pattern using the rBPMN Editor (reaction rule set diagram).

Figure 149. The Constraints Enforced by External Data pattern using the rBPMN Editor (integrity rule set diagram).
Figure 150. The Business Rule-Based Subprocess Selection pattern using the rBPMN Editor (process diagram).

Figure 151. The Business Rule-Based Subprocess Selection pattern using the rBPMN Editor (production rule set diagram).
Figure 152. The Business Rule-Based Process Composition pattern using the rBPMN Editor (process diagram).

Figure 153. The Business Rule-Based Process Composition pattern using the rBPMN Editor (derivation rule set diagram).

Figure 154. The Business Rule-Based Process Composition pattern using the rBPMN Editor (production rule set diagram).
APPENDIX E – RBPMN EDITOR SOURCE CODE

The rBPMN Editor will be available as an open source project at:

http://code.google.com/p/rbpmneditor/
APPENDIX F – RBPMN EDITOR SCREENCASTS

Screencasts created for the rBPMN Editor are available at:

http://www.youtube.com/user/rbpmn