DATA PERSPECTIVE IN BUSINESS PROCESS MANAGEMENT

THE ROLE OF DATA FOR PROCESS MODELING, ANALYSIS, AND EXECUTION

ANDREAS MEYER

BUSINESS PROCESS TECHNOLOGY
HASSO PLATTNER INSTITUTE, UNIVERSITY OF POTSDAM
POTSDAM, GERMANY

DISSERTATION
ZUR ERLANGUNG DES AKADEMISCHEN GRADES EINES
DOKTORS DER NATURWISSENSCHAFTEN
– DR. RER. NAT. –

June, 2015

Andreas Meyer: *Data Perspective in Business Process Management*, The Role of Data for Process Modeling, Analysis, and Execution, eingereicht an der Mathematisch-Naturwissenschaftlichen Fakultät der Universität Potsdam. Juni 2015.

Published online at the
Institutional Repository of the University of Potsdam:
URN urn:nbn:de:kobv:517-opus4-84806
http://nbn-resolving.de/urn:nbn:de:kobv:517-opus4-84806

ABSTRACT

Business process management (BPM) is a systematic and structured approach to model, analyze, control, and execute business operations also referred to as business processes that get carried out to achieve business goals. Central to BPM are conceptual models. Most prominently, process models describe which tasks are to be executed by whom utilizing which information to reach a business goal. Process models generally cover the perspectives of control flow, resource, data flow, and information systems.

Execution of business processes leads to the work actually being carried out. Automating them increases the efficiency and is usually supported by process engines. This, though, requires the coverage of control flow, resource assignments, and process data. While the first two perspectives are well supported in current process engines, data handling needs to be implemented and maintained manually. However, model-driven data handling promises to ease implementation, reduces the error-proneness through graphical visualization, and reduces development efforts through code generation.

This thesis addresses the modeling, analysis, and execution of data in business processes and presents a novel approach to execute data-annotated process models entirely model-driven. As a first step and formal grounding for the process execution, a conceptual framework for the integration of processes and data is introduced. This framework is complemented by operational semantics through a Petri net mapping extended with data considerations. Model-driven data execution comprises the handling of complex data dependencies, process data, and data exchange in case of communication between multiple process participants. This thesis introduces concepts from the database domain into BPM to enable the distinction of data operations, to specify relations between data objects of the same as well as different of types, to correlate modeled data nodes as well as received messages to the correct run-time process instances, and to generate messages for inter-process communication. The underlying approach, which is not limited to a particular process description language, has been implemented as proof-of-concept.

Automation of data handling in business processes requires data-annotated and correct process models. Targeting the former, algorithms are introduced to extract information about data nodes, their states, and data dependencies from control information and to annotate the process model accordingly. Usually, not all required information can be extracted from control flow information, since some data manipulations are not specified. This requires further refinement of the process model.

Given a set of object life cycles specifying allowed data manipulations, automated refinement of the process model towards containment of all data manipulations is enabled. Process models are an abstraction focusing on specific aspects in detail, e. g., the control flow and the data flow views are often represented through activity-centric and object-centric process models. This thesis introduces algorithms for roundtrip transformations enabling the stakeholder to add information to the process model in the view being most appropriate.

Targeting process model correctness, this thesis introduces the notion of weak conformance that checks for consistency between given object life cycles and the process model such that the process model may only utilize data manipulations specified directly or indirectly in an object life cycle. The notion is computed via soundness checking of a hybrid representation integrating control flow and data flow correctness checking. Making a process model executable, identified violations must be corrected. Therefore, an approach is proposed that identifies for each violation multiple, alternative changes to the process model or the object life cycles.

Utilizing the results of this thesis, business processes can be executed entirely model-driven from the data perspective in addition to the control flow and resource perspectives already supported before. Thereby, the model creation is supported by algorithms partly automating the creation process while model consistency is ensured by data correctness checks.

ZUSAMMENFASSUNG

Geschäftsprozessmanagement ist ein strukturierter Ansatz zur Modellierung, Analyse, Steuerung und Ausführung von Geschäftsprozessen, um Geschäftsziele zu erreichen. Es stützt sich dabei auf konzeptionelle Modelle, von denen Prozessmodelle am weitesten verbreitet sind. Prozessmodelle beschreiben wer welche Aufgabe auszuführen hat, um das Geschäftsziel zu erreichen, und welche Informationen dafür benötigt werden. Damit beinhalten Prozessmodelle Informationen über den Kontrollfluss, die Zuweisung von Verantwortlichkeiten, den Datenfluss und Informationssysteme.

Die Automatisierung von Geschäftsprozessen erhöht die Effizienz der Arbeitserledigung und wird durch Process Engines unterstützt. Dafür werden jedoch Informationen über den Kontrollfluss, die Zuweisung von Verantwortlichkeiten für Aufgaben und den Datenfluss benötigt. Während aktuelle Process Engines die ersten beiden Informationen weitgehend automatisiert verarbeiten können, müssen Daten manuell implementiert und gewartet werden. Dem entgegen verspricht ein modell-getriebenes Behandeln von Daten eine vereinfachte Implementation in der Process Engine und verringert gleichzeitig die Fehleranfälligkeit dank einer graphischen Visualisierung und reduziert den Entwicklungsaufwand durch Codegenerierung.

Die vorliegende Dissertation beschäftigt sich mit der Modellierung, der Analyse und der Ausführung von Daten in Geschäftsprozessen. Als formale Basis für die Prozessausführung wird ein konzeptuelles Framework für die Integration von Prozessen und Daten eingeführt. Dieses Framework wird durch operationelle Semantik ergänzt, die mittels einem um Daten erweiterten Petrinetz-Mapping vorgestellt wird. Die modellgetriebene Ausführung von Daten muss komplexe Datenabhängigkeiten, Prozessdaten und den Datenaustausch berücksichtigen. Letzterer tritt bei der Kommunikation zwischen mehreren Prozessteilnehmern auf. Diese Arbeit nutzt Konzepte aus dem Bereich der Datenbanken und überführt diese ins Geschäftsprozessmanagement, um Datenoperationen zu unterscheiden, um Abhängigkeiten zwischen Datenobjekten des gleichen und verschiedenen Typs zu spezifizieren, um modellierte Datenknoten sowie empfangene Nachrichten zur richtigen laufenden Prozessinstanz zu korrelieren und um Nachrichten für die Prozessübergreifende Kommunikation zu generieren. Der entsprechende Ansatz ist nicht auf eine bestimmte Prozessbeschreibungssprache begrenzt und wurde prototypisch implementiert.

Die Automatisierung der Datenbehandlung in Geschäftsprozessen erfordert entsprechend annotierte und korrekte Prozessmodelle. Als Unterstützung zur Datenannotierung führt diese Arbeit einen Algo-

rithmus ein, welcher Informationen über Datenknoten, deren Zustände und Datenabhängigkeiten aus Kontrollflussinformationen extrahiert und die Prozessmodelle entsprechend annotiert. Allerdings können gewöhnlich nicht alle erforderlichen Informationen aus Kontrollflussinformationen extrahiert werden, da detaillierte Angaben über mögliche Datenmanipulationen fehlen. Deshalb sind weitere Prozessmodellverfeinerungen notwendig. Basierend auf einer Menge von Objektlebenszyklen kann ein Prozessmodell derart verfeinert werden, dass die in den Objektlebenszyklen spezifizierten Datenmanipulationen automatisiert in ein Prozessmodell überführt werden können. Prozessmodelle stellen eine Abstraktion dar. Somit fokussieren sie auf verschiedene Teilbereiche und stellen diese im Detail dar. Solche Detailbereiche sind beispielsweise die Kontrollflusssicht und die Datenflusssicht, welche oft durch Aktivitäts-zentrierte beziehungsweise Objekt-zentrierte Prozessmodelle abgebildet werden. In der vorliegenden Arbeit werden Algorithmen zur Transformation zwischen diesen Sichten beschrieben.

Zur Sicherstellung der Modellkorrektheit wird das Konzept der „weak conformance" zur Überprüfung der Konsistenz zwischen Objektlebenszyklen und dem Prozessmodell eingeführt. Dabei darf das Prozessmodell nur Datenmanipulationen enthalten, die auch in einem Objektlebenszyklus spezifiziert sind. Die Korrektheit wird mittels Soundness-Überprüfung einer hybriden Darstellung ermittelt, so dass Kontrollfluss- und Datenkorrektheit integriert überprüft werden. Um eine korrekte Ausführung des Prozessmodells zu gewährleisten, müssen gefundene Inkonsistenzen korrigiert werden. Dafür werden für jede Inkonsistenz alternative Vorschläge zur Modelladaption identifiziert und vorgeschlagen.

Zusammengefasst, unter Einsatz der Ergebnisse dieser Dissertation können Geschäftsprozesse modellgetrieben ausgeführt werden unter Berücksichtigung sowohl von Daten als auch den zuvor bereits unterstützten Perspektiven bezüglich Kontrollfluss und Verantwortlichkeiten. Dabei wird die Modellerstellung teilweise mit automatisierten Algorithmen unterstützt und die Modellkonsistenz durch Datenkorrektheitsüberprüfungen gewährleistet.

PUBLICATIONS

Some ideas and figures have appeared previously in the following publications:

- Andreas Meyer, Luise Pufahl, Kimon Batoulis, Dirk Fahland, and Mathias Weske. Automating data exchange in process choreographies. *Information Systems*, 53:296–329, 2015.
- Andreas Meyer, Luise Pufahl, Dirk Fahland, and Mathias Weske. Modeling and Enacting Complex Data Dependencies in Business Processes. In *Business Process Management (BPM)*, pages 171–186. Springer, 2013.
- Andreas Meyer and Mathias Weske. Extracting Data Objects and their States from Process Models. In *Enterprise Distributed Object Computing (EDOC)*, pages 27–36. IEEE, 2013.
- Andreas Meyer and Mathias Weske. Weak Conformance between Process Models and Synchronized Object Life Cycles. In *Service-Oriented Computing (ICSOC)*, pages 359–367. Springer, 2014.
- Andreas Meyer, Luise Pufahl, Kimon Batoulis, Sebastian Kruse, Thorben Lindhauer, Thomas Stoff, Dirk Fahland, and Mathias Weske. Automating Data Exchange in Process Choreographies. In *Advanced Information Systems Engineering (CAiSE)*, pages 316–331. Springer, 2014.
- Andreas Meyer, Sergey Smirnov, and Mathias Weske. Data in Business Processes. *EMISA Forum*, 31(3):5–31, 2011.
- Luise Pufahl, Andreas Meyer, and Mathias Weske. Batch Regions: Process Instance Synchronization based on Data. In *Enterprise Distributed Object Computing (EDOC)*, pages 150–159. IEEE, 2014.
- Andreas Meyer and Mathias Weske. Activity-centric and Artifact-centric Process Model Roundtrip. In *Business Process Management (BPM) Workshops*, pages 167–181. Springer, 2013.
- Andreas Meyer, Artem Polyvyanyy, and Mathias Weske. Weak Conformance of Process Models with respect to Data Objects. In *Services and their Composition (ZEUS)*, pages 74–80. CEUR-WS, 2012.
- Andreas Meyer, Luise Pufahl, Dirk Fahland, and Mathias Weske. Enacting Complex Data Dependencies from Activity-Centric Business Process Models. In *Demo Sessions of the 11th International Conference on Business Process Management (BPM)*, pages 11–15. CEUR-WS, 2013.
- Andreas Meyer, Sergey Smirnov, and Mathias Weske. Data in Business Processes. Technical Report 50, Hasso Plattner Institute at the University of Potsdam, 2011.

- Andreas Meyer, Luise Pufahl, Dirk Fahland, and Mathias Weske. Modeling and Enacting Complex Data Dependencies in Business Processes. Technical Report 74, Hasso Plattner Institute at the University of Potsdam, 2013.
- Andreas Meyer, Luise Pufahl, Kimon Batoulis, Sebastian Kruse, Thorben Lindhauer, Thomas Stoff, Dirk Fahland, and Mathias Weske. Data Perspective in Process Choreographies: Modeling and Execution. Technical Report BPM-13-29, BPMcenter.org, 2013.
- Andreas Meyer and Mathias Weske. Activity-centric and Artifact-centric Process Model Roundtrip. Technical Report, Hasso Plattner Institute at the University of Potsdam, 2013.
- Andreas Meyer and Mathias Weske. Weak Conformance between Process Models and Synchronized Object Life Cycles. Technical Report 91, Hasso Plattner Institute at the University of Potsdam, 2014.
- Luise Pufahl, Andreas Meyer, and Mathias Weske. Batch Regions: Process Instance Synchronization based on Data. Technical Report 86, Hasso Plattner Institute at the University of Potsdam, 2013.

In addition to above publications as part of this thesis, I was also involved in the following research indirectly contributing to this thesis:

- Nico Herzberg, Andreas Meyer, and Mathias Weske. Improving business process intelligence by observing object state transitions. *Data & Knowledge Engineering (DKE)*, 98:144–164, 2015.
- Kimon Batoulis, Andreas Meyer, Ekaterina Bazhenova, Gero Decker, and Mathias Weske. Extracting Decision Logic from Process Models. In *Advanced Information Systems Engineering (CAiSE)*, pages 349–366. Springer, 2015.
- Andreas Meyer and Mathias Weske. Data Support in Process Model Abstraction. In *Conceptual Modeling (ER)*, pages 292–306. Springer, 2012.
- Josefine Harzmann, Andreas Meyer, and Mathias Weske. Deciding Data Object Relevance for Business Process Model Abstraction. In *Conceptual Modeling (ER)*, pages 121–129. Springer, 2013.
- Andreas Meyer, Nico Herzberg, Frank Puhlmann, and Mathias Weske. Implementation Framework for Production Case Management: Modeling and Execution. In *Enterprise Distributed Object Computing (EDOC)*, pages 190–199. IEEE, 2014.
- Rami-Habib Eid-Sabbagh, Marcin Hewelt, Andreas Meyer, and Mathias Weske. Deriving Business Process Data Architectures from Process Model Collections. In *Service-Oriented Computing (ICSOC)*, pages 533–540. Springer, 2013.

- Luise Pufahl, Nico Herzberg, Andreas Meyer, and Mathias Weske. Flexible Batch Configuration in Business Processes based on Events. In *Service-Oriented Computing (ICSOC)*, pages 63–78. Springer, 2014.
- Nico Herzberg, Andreas Meyer, and Mathias Weske. An Event Processing Platform for Business Process Management. In *Enterprise Distributed Object Computing (EDOC)*, pages 107–116. IEEE, 2013.
- Nico Herzberg, Andreas Meyer, Oleh Khovalko, and Mathias Weske. Improving Business Process Intelligence with Object State Transition Events. In *Conceptual Modeling (ER)*, pages 146–160. Springer, 2013.
- Nico Herzberg, Andreas Meyer, and Mathias Weske. Improving Process Monitoring and Progress Prediction with Data State Transition Events. In *Services and their Composition (ZEUS)*, pages 20–23, 2013.
- Susanne Bülow, Michael Backmann, Nico Herzberg, Thomas Hille, Andreas Meyer, Benjamin Ulm, Tsun Yin Wong, and Mathias Weske. Monitoring of Business Processes with Complex Event Processing. In *Business Process Management (BPM) Workshops*, pages 277–290. Springer, 2013.
- Michael Backmann, Anne Baumgrass, Nico Herzberg, Andreas Meyer, and Mathias Weske. Model-driven Event Query Generation for Business Process Monitoring. In *Service-Oriented Computing (ICSOC) Workshops*, pages 406–418. Springer, 2013.
- Anne Baumgrass, Nico Herzberg, Andreas Meyer, and Mathias Weske. BPMN Extension for Business Process Monitoring. In *Enterprise Modelling and Information Systems Architectures (EMISA)*, pages 85–98. GI, 2014.
- Rami-Habib Eid-Sabbagh, Matthias Kunze, Andreas Meyer, and Mathias Weske. A Platform for Research on Process Model Collections. In *BPMN*, pages 8–22. Springer, 2012.
- Michael Goderbauer, Markus Goetz, Alexander Grosskopf, Andreas Meyer, and Mathias Weske. Syncro - Concurrent Editing Library for Google Wave. In *Web Engineering (ICWE)*, pages 510–513. Springer, 2010.
- Ahmed Awad, Alexander Grosskopf, Andreas Meyer, and Mathias Weske. Enabling Resource Assignment Constraints in BPMN. Technical Report, Hasso Plattner Institute at the University of Potsdam, 2009.

ACKNOWLEDGMENTS

Looking back the past few years, I am very happy to have been surrounded by so many brilliant people helping me to pave my path towards this thesis. First and foremost, I like to thank Mathias Weske, my supervisor, for his continuous encouragement and support during my whole PhD. Especially, I am grateful to have gotten all the time I needed to make my way to the actual topic of this thesis while experimenting in very different fields of BPM. This freedom really made me enjoy the time working for and writing my thesis.

Furthermore, I really like to thank many colleagues from within the BPT group – where I saw many colleagues going and coming – as well as from the BPM community. In this respect, I am very happy to had the chance of working with Dirk Fahland that started during his stay in Potsdam and lasted until the end of my PhD. Dirk Fahland let me experience a very stringent and goal-oriented research approach with focusing on every detail and considering any single comment reviewers and colleagues provide to ideas. It was a lot of work – but also much more fun. From within the BPT group, I especially like to thank Luise Pufahl for countless discussions and all her support in many ways. Sometimes, I would not have known what to do without Luise. The same holds true for Anne Baumgrass for many reasons – thanks for keeping me safe especially in the final days of writing this thesis and I am really looking forward to successfully work with Anne for many more years. Mathias Kunze was my first source of targeting a question at, since Matthias literally knew always an answer and helped me a lot in very fruitful and goal-oriented discussions. My thanks also goes to Artem Polyvyanyy and Sergey Smirnov who supported me in finding my path towards data and bringing me to data consistency and data abstraction as two very interesting topics of my research work. And special thanks to Artem for taking me into plenty formalization challenges helping to better understand how to actually work formally.

Many more colleagues intensively worked with me and joined inspiring discussions; I am grateful for every single contact. Thank you!

Besides the research, I really enjoyed the time given to me working with bachelor's and master's students in projects and seminars. I am still very proud of the first students team I supervised together with Mathias Weske and Alexander Lübbe. The students worked on a cloud-based collaborative process modeling tool for Google wave and made it to the keynote of the Google IO for presenting their tool. But I will also never forget all other projects and seminars for many hours of fun.

Andreas Meyer
June, 2015

CONTENTS

I BACKGROUND 1
1 INTRODUCTION 3
1.1 The Essence of Business Process Management 4
1.2 Data in Business Processes 8
1.3 Problem Statement 11
1.4 Contributions 14
1.5 Structure of Thesis 17
2 PROCESS MODELS 21
2.1 Application of Process Models 22
2.2 Data Support in Process Models 25
2.3 Business Process Description Language: BPMN 26
2.4 Scenario: Build-to-Order and Delivery Process 28
3 FOUNDATION 37
3.1 Business Process Models 39
3.2 Business Process Relations 48
3.3 Net Systems 49
II HYBRID PROCESS MODEL FOR DATA AND CONTROL 57
4 PROCESS AND DATA VIEW INTEGRATION 59
4.1 Data Aspects 61
4.2 Business Process Models 69
4.3 Process Instance View 74
4.4 Business Processes 79
4.5 Process Choreographies 81
4.6 Conceptual Model 84
4.7 Formal Semantics 86
4.8 Related Work 94
4.9 Conclusion 97
5 EXTRACTION OF DATA NODES AND THEIR STATES 99
5.1 Extraction Algorithms for Generic Process Models 101
5.2 Application to Process Description Languages 111
5.3 Evaluation 113
5.4 Related Work 121
5.5 Conclusion 122
6 WEAK CONFORMANCE OF PROCESS SCENARIOS 123
6.1 The Notion of Weak Conformance 128
6.2 Computation via Soundness Checking 131
6.3 Correction of Process Scenarios 141
6.4 Related Work 149
6.5 Conclusion 154
7 MODEL TRANSFORMATIONS 155

7.1 Object-centric Process Model to Object Life Cycle 158
7.2 Object Life Cycle to Activity-centric Process Model 161
7.3 Activity-centric Process Model to Object Life Cycle 171
7.4 Object Life Cycle to Object-centric Process Model 173
7.5 Process Model Refinement 176
7.6 Object Life Cycle Tailoring 182
7.7 Related Work 190
7.8 Conclusion 191

III AUTOMATED PROCESS MODEL EXECUTION 193

8 MODEL-DRIVEN BUSINESS PROCESS EXECUTION 195
8.1 Complex Data Dependencies in Orchestrations 198
8.2 Patterns for SQL-Query Derivation 217
8.3 Process Data Handling 243
8.4 Automating Data Exchange in Choreographies 248
8.5 Correctness and Consistency Discussions 265
8.6 Evaluation 277
8.7 Related Work 302
8.8 Conclusion 307

9 CONCLUSIONS 309
9.1 Contributions of this Thesis 310
9.2 Relevance of Data in Business Process Management 313
9.3 Limitations & Future Research 316

IV APPENDIX 319

A INTER-VIEW TRANSFORMATION ALGORITHMS 321

BIBLIOGRAPHY 327

LIST OF FIGURES

Figure 1 Business process life cycle. 5
Figure 2 Structure of thesis. 18

Figure 3 Data support in processes. 25
Figure 4 BPMN example. 28
Figure 5 Running example: Computer retailer. 29
Figure 6 Running example: Collect orders. 30
Figure 7 Running example: Process order. 30
Figure 8 Running example: Prepare purchase order. 31
Figure 9 Running example: Request quotes. 31
Figure 10 Running example: Handle purchase order. 32
Figure 11 Running example: Handle payment. 32
Figure 12 Running example: Computer retailer detailed. 33
Figure 13 Running example: Customer. 34
Figure 14 Running example: Supplier. 35
Figure 15 Running example: Customer (XOR). 35
Figure 16 Running example: Supplier (XOR). 35

Figure 17 Process model example. 42
Figure 18 Marking of a process model. 43
Figure 19 Object-centric process model in CMMN notation. 46
Figure 20 Object life cycle example. 47
Figure 21 Petri net example. 50
Figure 22 State of Petri net after transition firing. 51
Figure 23 Short-circuited workflow net. 53

Figure 24 Correlation between data entities. 61
Figure 25 Example data node. 62
Figure 26 Data model for running example. 64
Figure 27 Object life cycle of data class CO. 65
Figure 28 Synchronized object life cycle. 68
Figure 29 Process model. 72
Figure 30 Activity life cycle. 74
Figure 31 Three process models with data dependencies. 76
Figure 32 Data marking of a process model. 76
Figure 33 Data view example. 78
Figure 34 Tree of data classes of a data model. 80
Figure 36 Process choreography. 83
Figure 37 Global data model for choreography. 83
Figure 38 Conceptual model: Process & data integration. 85
Figure 39 Mapping functions: Process & data integration. 85

Figure 40 Mapping a process orchestration to a Petri net. 88
Figure 41 Petri net with data dependency coverage. 89
Figure 42 Marking of *process order* subprocess. 91
Figure 43 Mapping a process choreoraphy to a Petri net. 93
Figure 44 Petri net mapping for process choreography. 94

Figure 45 Data extraction – before-after-comparison. 102
Figure 46 Screenshot: Result of automatic computation. 115
Figure 47 Automatic vs. manual data extraction. 117
Figure 48 User rating of annotation quality and usefulness. 119

Figure 49 *Process order* process model. 125
Figure 50 Synchronized OLC for four classes. 125
Figure 51 Examples for identifiable modeling errors. 127
Figure 52 Extract of Petri net representing a process model. 131
Figure 53 Place merging in Petri net. 132
Figure 54 Mapping of currently-typed synchronization edges. 132
Figure 55 Mapping of untyped synchronization edges. 132
Figure 56 Petri net extract of a synchronized OLC. 133
Figure 57 Handling of not occurring data states. 134
Figure 58 Internal structure of data state places. 134
Figure 59 Integration rules for process model and OLC. 135
Figure 60 Enabler and collector fragment addition. 137
Figure 61 Final enabler and collector fragment addition. 137
Figure 62 Extract of a workflow net system. 138
Figure 63 Workflow net system. 140
Figure 64 Concurrent data access examples. 145

Figure 65 Model transformations overview. 157
Figure 66 Synchronized OLC derived from OCP. 161
Figure 67 Example for SAD-6 application. 164
Figure 68 Iterative addition of join and merge gateways. 166
Figure 69 ACP derived from an OLC. 167
Figure 70 Data model. 170
Figure 71 Synchronized OLC derived from an ACP. 173
Figure 72 Resolving “gap” by process model refinement. 179
Figure 73 Resolving “jump” by process model refinement. 181
Figure 74 Data state consideration for model refinement. 184
Figure 75 Data dependencies restrict control flow. 184
Figure 76 OLC tailoring example with “jumps” kept. 187
Figure 77 OLC tailoring example with “jumps” reduced. 187
Figure 78 Initially tailored object life cycles. 190
Figure 79 Integrated tailored object life cycle. 190

Figure 80 Classical architecture of a WFMS. 196
Figure 81 Improved architecture of a WFMS. 197

Figure 82 Consolidated process model with m:n dependency. 198
Figure 83 Implicit OLC representation in activity. 201
Figure 84 Data node interactions with cardinalities. 202
Figure 85 CRUD operations for data nodes. 203
Figure 86 Extended data node. 204
Figure 87 Data model for modeling example. 206
Figure 88 Customer order collection. 206
Figure 89 Purchase order preparation. 208
Figure 90 Derived object life cycles. 209
Figure 91 Example process fragments for query derivation. 211
Figure 92 Setting missing foreign key for m:n relation. 216
Figure 93 Data model for case object. 219
Figure 94 Data model for dependent$^{1:1}$ objects. 222
Figure 95 Data model for dependent$^{1:n}$ objects. 226
Figure 96 Data model for dependent$^{m:n}$ objects. 231
Figure 97 Attribute consideration for activity reads and writes. 244
Figure 98 Explicit attribute annotation to data node. 245
Figure 99 Task *Receive quote*. 246
Figure 100 Form generation example. 247
Figure 101 Request for quote choreography. 248
Figure 102 Global choreography model. 248
Figure 103 Modeling guideline. 252
Figure 104 Collaboration example for *Computer retailer*. 254
Figure 105 Message class. 254
Figure 106 Schema mapping for *Computer retailer*. 255
Figure 107 Private process and local data model of *Supplier*. 257
Figure 108 Approach overview. 258
Figure 109 Example message flow (instance level). 260
Figure 110 Private process model of *Computer retailer*. 267
Figure 111 Workflow net extract for *Computer retailer*. 267
Figure 112 Integrated workflow net for *Computer retailer*. 268
Figure 113 Global collaboration diagram. 271
Figure 114 Workflow modules. 272
Figure 115 Workflow nets for process model consistency. 273
Figure 116 Petri net for correlation handling (initialization). 276
Figure 117 Petri net for correlation handling (init. error). 277
Figure 118 Process model for SQL derivation explanation. 279
Figure 119 Data handling in implementation. 279
Figure 120 Data model for patterns P1 to P4. 282
Figure 121 Pattern P1: send. 282
Figure 122 Pattern P2: receive. 282
Figure 123 Pattern P3: send/receive (participant A). 283
Figure 124 Pattern P3: send/receive (participant B). 284
Figure 125 Pattern P4: with message events. 285
Figure 126 Pattern P4: with receive tasks. 285
Figure 127 Data model for patterns P5 and P7. 286

Figure 128 Patterns P5 and P7: one-to-many send/receive. 287
Figure 129 Data model for pattern P6. 287
Figure 130 Pattern P6: running instance. 288
Figure 131 Pattern P6: create instance. 288
Figure 132 Data model for pattern P8. 289
Figure 133 Pattern P8: multi-responses (participant A). 290
Figure 134 Pattern P8: multi-responses (participant B). 290
Figure 135 Data model for pattern P9. 291
Figure 136 Pattern P9: contingent requests (participant A). 292
Figure 137 Data model for pattern P10. 293
Figure 138 Pattern P10: atomic multicast (participant A). 294
Figure 139 Pattern P10: atomic multicast (participant B). 295
Figure 140 Data model for pattern P11. 296
Figure 141 Pattern P11: request with referral (particip. A). 297
Figure 142 Pattern P11: request with referral (particip. B). 297
Figure 143 Pattern P11: request with referral (particip. C). 297
Figure 144 Data model for pattern P12. 298
Figure 145 Pattern P12: relayed request (participant A). 299
Figure 146 Pattern P12: relayed request (participant B). 299
Figure 147 Pattern P12: relayed request (participant C). 299
Figure 148 Data model for pattern P13. 301
Figure 149 Pattern P13: dynamic routing (participant A). 301
Figure 150 Pattern P13: dynamic routing (participant B). 301
Figure 151 Pattern P13: participants C and D. 302

LIST OF TABLES

Table 1 Rule-based object-centric process model. 45

Table 2 Comparison: conformance computation algorithms. 126
Table 3 Ranking of mechanisms for violation correction. 148

Table 4 Rule-based object-centric process model. 160
Table 5 Extract of OCP visualizing differences after roundtrip. 175

Table 6 SQL queries for patterns P1 to P3. 213
Table 7 Mapping and query for patterns P4 and P5. 213
Table 8 SQL queries for patterns P6 and P7. 215
Table 9 Pattern classification overview. 217
Table 10 Patterns for case object. 219
Table 11 Patterns for dependent$^{1:1}$ objects. 222
Table 12 Patterns for dependent$^{1:n}$ objects. 226
Table 13 Patterns for dependent$^{m:n}$ objects. 232
Table 14 Patterns for process and subprocess instantiation. 239
Table 15 Patterns for handling single attributes. 241
Table 16 Multi-instance task handling. 242
Table 17 Comparison: data-aware modeling techniques. 304
Table 18 Review requirements support. 305

ACRONYMS

ACP Activity-centric Process Model

AD Activity Diagram

BPA Business Process Architecture

BPEL Business Process Execution Language

BPM Business Process Management

BPMA Business Process Model Abstraction

BPMI Business Process Management Initiative

BPMN Business Process Model And Notation

BPMS Business Process Management System

CMMN Case Management Model And Notation

CSP Communicating Sequential Processes

CTL Computational Tree Logic

EPC Event-driven Process Chain

FSP Finite State Processes

GSM Guard-Stage-Milestone

IT Information Technology

KPI Key Performance Indicator

MDE Model-driven Engineering

OMG Object Management Group

OCP Object-centric Process Model

OLC Object Life Cycle

P2P Public-to-private

PCM Production Case Management

SOA Service-oriented Architecture

SQL Structured Query Language

UML Unified Modeling Language

WFMS Workflow Management System

WSDL Web Services Description Language

XML Extensible Markup Language

XQUERY XML Query Language

YAWL Yet Another Workflow Language

Part I

BACKGROUND

1 INTRODUCTION

All businesses are process-driven independently from the domain and their organizational background. Process-orientation is an organizational principle that emerged from business administration [113, 300] and organizational redesign [62, 126] and got established in organizations within the last 15 years [311]. However, the idea – although not conceptualized – dates back to 1776 where Smith describes the division of labor by separating the work into a set of simple tasks to be executed in sequence by different workers [310] implicitly representing the production process and 1932 where Nordsieck provides first thoughts about the necessity of processes-orientation [238].

Business process management (BPM) is a systematic and structured approach to analyze, improve, control, manage, configure, monitor, and execute business operations that are performed to achieve a business value [91, 370] also represented as business goals. Central to BPM are *process models* since they specify these business operations, i. e., they contain the business knowledge [78]. A model, e. g., a process model, follows a specific purpose it is created for [313]. Generally, models are an abstraction of the real world; i. e., a projection that reduces the amount of information displayed by filtering and only mapping relevant information into the model [155, 169].

Commonly, process models are seen as collection of activities – representing the business operations – along with their logical and temporal order [24, 370]. Enactment of process models comprises process execution and monitoring and thus, it allows the achievement of business goals and can be done manually as well as automatically with the use of information technology (IT). IT is one of the core enablers for BPM [62, 126, 341], especially for process enactment. IT supports people who actually execute business operations, increases performance, allows monitoring of business operations with real-time data, reduces replication and allows synthesis of process data, and allows part as well as full automation of business operations [82, 126, 184, 311, 320, 346].

Against this background, further perspectives represented in process models became important besides the logical and temporal ordering of activities – the *control flow perspective* [345]. These further perspectives are, most prominently, the *organizational perspective* and the *data perspective*. The former comprises resource management, i. e., who executes which activity when under which conditions and who has when access to which data. The data perspective strengthens the concept of entities being processed by activities, e. g., specifying which information are required to start activity execution or which results are expected after executing an activity. While the integration between control flow and resources is well researched in terms of general resource management [26, 39, 40, 43] including generic resource management patterns [208, 290], access control [11, 22, 41, 315], and role resolution [180, 299], the question of *how to handle data generically in process models as well as integrated with the control flow* remains open although it is fundamental for automated process enactment.

In this chapter, we detail this question by first reviewing the essentials of business process management in Section 1.1 followed by a discussion of the drivers for data consideration in business processes in Section 1.2 that gives raise to the problem statement of this thesis in Section 1.3. Section 1.4 summarizes our contributions before Section 1.5 outlines the structure of this thesis.

1.1 THE ESSENCE OF BUSINESS PROCESS MANAGEMENT

Process-orientation is an organizational principle that emphasizes *business processes* as source of value creation. Literature reveals manifold definitions on business processes, e. g., [62, 83, 109, 112, 126, 311, 341, 370], which commonly share the idea of activities being conducted to transform some input into some output being of value for the business. With respect to [370], we informally define a business process as follows.

Definition 1.1 (Business Process (informal))**.**
A *business process* is a collection of activities that are conducted in co-

ordination and that jointly realize a business goal in an organizational, technical, and informational environment. Each activity contains information about its input and its output and gets performed by some resource. The enactment of a business process within a single organization is referred to as *process orchestration* but it may interact with processes of multiple other organizations referred to as *process choreography*. ◂

Business process life cycle

BPM comprises the *design and analysis*, *configuration*, *enactment*, and *evaluation* of business processes [370] that describe the four logically interdependent phases each business process iterates through. These phases have been established in a life cycle model: the business process life cycle [346] which is represented in Figure 1. This conceptual model has seen wide acceptance and application, e. g., in [83, 109, 112, 125, 200, 341, 370]. The core entities, the business process life cycle is targeting at, are the process model and multiple artifacts at different levels of abstraction related to it, e. g., organizational data, execution data, technical data, and compliance rules. The process models as well as the additional artifacts need to be managed well in a structural fashion to efficiently apply business process management. Information technology as a core enabler for BPM [62, 126, 341] – represented in the center of Figure 1 – supports and combines the four phases and allows efficient BPM application. Next, we briefly introduce each of the four phases of the business process life cycle.

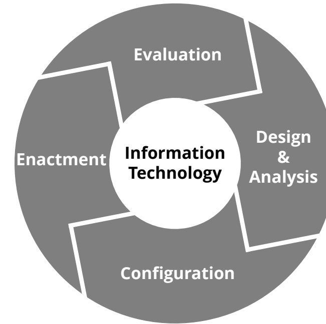


Figure 1: The business process life cycle consisting of four logically interdependent, cyclical phases with information technology, a core enabler for business process management, combining them.

DESIGN AND ANALYSIS. Since process models are central to BPM because of visualizing the operational information, the business process life cycle is typically entered in the design and analysis phase. In this phase, business processes are identified, reviewed, validated, and finally (after elicitation) represented by process models that are the basis for the remaining phases of the business process life cycle. Alternatively, existing process models can be improved through information captured in the other phases. Through the central role of process models, it is equally important to elicit them and to ensure their correctness and plausibility by analysis. Analysis comprises the validation and verification of process models. Validation refers to checking whether the model provides a precise representation (of the real world business process for the chosen purpose) and shows the activities actually intended

to be executed towards the business goal. Additionally, the consideration of given compliance rules, further business guidelines, governmental regulations, etc. is validated. Validation is often performed through workshops with the stakeholders or by simulation. Verification is typically distinguished into structural (or syntactical) and semantic (or behaviorally) correctness. For both types, large sets of analysis techniques have been developed in research and industry; e. g., [20, 254, 257, 331, 348, 369].

CONFIGURATION. Once the process model design and analysis is completed, it needs to be installed into the organizational, technical, and informational environment of the organization. The implementation allows process execution in the range from manual to full automation. The specification of policies, operational guidelines, and procedures to be followed by the employees can result in manual execution without dedicated support from a business process management system (BPMS), a software system supporting the execution and actually executing processes. An example BPMS is a workflow management system (WFMS). For automation, the process model must be enhanced with technical details required for execution and the system configuration must be specified. This includes most prominently the system as well as the user interfaces to allow interaction with employees and other legacy systems. Finally, the configuration phase also comprises integration and performance tests before the process model can be deployed into practice. Generally, organizational internal, the process model acts as blue print for its implementation independently from manual or automated execution.

ENACTMENT. This phase covers the actual run-time of business processes comprising their execution and monitoring [370]. Execution refers to the initiation and run of process instances (cases) to fulfill the business goals targeted by the corresponding process models. Monitoring refers to showing the status and progress of process instances based on the underlying process models and matching performance data to key performance indicators in near-real-time. In case of some automatic execution, the BPMS actively controls the execution by ensuring correct ordering of activities as well as utilization and creation of inputs and outputs of activities according to process model specifications [50, 82]. At run-time, execution data is captured. This is typically represented in logs.

EVALUATION. Based on the captured execution data and additionally available information, business processes, their models, and their implementations can be evaluated and improved [63, 130]. Evaluation may provide information about bottlenecks or unsuccessful respectively unintended results for some cases that then can be improved in a subsequent design phase. Process mining [82, 335,

336] is another mean to analyze the logs and, for instance, find deviations between the specified and the actually executed process model that, again, can be input to the next design phase for process model improvement.

In an iteration through the business process life cycle, phases can be skipped. For instance, if the configuration reveals issues with respect to the process model, the design and analysis phase can be re-entered without enacting and evaluating the business process. Likewise, the design and analysis phase can be directly entered from anywhere, if external factors, e. g., governmental regulations, company guidelines, and operational changes, require an adaptation of the business process. Referring to the business process life cycle, we define business process management as follows.

Definition 1.2 (Business Process Management).
Business process management (BPM) comprises concepts, methods, techniques, and systems to design, analyze, configure, enact (i. e., execute and monitor), and evaluate business processes, their models, and their implementations involving organizations, data, and technological infrastructure. ◂

Business process management provides an holistic view on business processes and makes the value creation of organizations explicit. Actual value creation is achieved by process execution which may use the full spectrum from fully manual to fully automated execution with different levels of information technology support. Automatic execution is performed by or at least in the context of some business process management system (BPMS) that coordinates activity execution.

Definition 1.3 (Business Process Management System).
A *business process management system (BPMS)* is a generic information system driven by explicit process representations, e. g., process models, to coordinate the enactment of business processes. ◂

In this thesis, we utilize the term WFMS as synonym to the term business process management system and consider a *process engine* as one part of a WFMS that actually executes a business process.

Classification of business processes

In business process management, different levels of control can be specified ranging from models for strategic goal representation or strategic decision taking to models for implementation or fully automated execution [271, 370]. Each level has different requirements on what needs to be specified. Additionally, on each level, models may contain different information, since the purpose of a model determines the information to be contained and the requirements to be matched. Thus, the content presented in different models differs in two dimensions. The differences within one level of control mostly relate to different views on the same real world process scenario by, for instance, putting

a specific participant, business goal, or the main execution path (also referred to as "happy path") in focus. Mostly, these different views combine to the overall picture on the process scenario. Conversely, considering models from different levels of control, they often differ significantly in content that may not be compatible even if considering the same process scenario. Most prominently, this is known as the "Business-IT-Gap" [35, 119, 282]. Utilizing information technology and providing an integration layer, BPM helps to close this gap.

Following [370], we distinguish four levels of control (LOC):

(LOC-1) Business goals and strategies,
(LOC-2) Organizational business processes,
(LOC-3) Operational business processes, and
(LOC-4) Implemented business processes.

The strategic level specifies high-level targets and long-term objectives to be achieved by the organization. For representation, usually plain text or goal models [178, 283, 355, 385] are used. These business goals get refined and clarified in organizational business processes that focus on the dependencies between business processes, their inputs, and their outputs. Organizational processes are often represented in textual form or by using semi-formal techniques. Operational processes contain activities and their logical and temporal order to achieve specified business goals. Typically, multiple operational processes are required to realize an organizational process referring to one or more business goals. Implemented process models additionally contain execution-specific information allowing the realization in its technical and organizational environment. For operational and implemented processes, process models as introduced above are utilized. In the remainder of this thesis, we focus on the two levels of control that utilize process models: operational and implemented business processes.

1.2 DATA IN BUSINESS PROCESSES

Data affects each phase in the business process life cycle. Data is modeled and added to the process models as well as analyzed with respect to, for instance, correctness in the *design and analysis* phase. Data information is utilized to determine which resource may access the required data (also utilizing information on access control and resource management), to implement the actual data access to allow manipulation of data, and to provide access to IT systems for the identified employees in the *configuration* phase. Data steers process execution by enabling activities upon existence of required information after the control flow reaches the activities and by storing execution results. Additionally, data builds the basis for process monitoring since it represents the actual happenings and results during process execution in the *enactment* phase. Data, read or written in activities or used for decision taking during process execution, can be captured, for instance, in log files.

This information is utilized for process improvements (e. g., reducing execution time and saving costs) and process controlling in the *evaluation* phase.

The relevance of data in business processes can be motivated along three areas contributing to different parts of the business process life cycle: (i) representation of organization's core assets (design), (ii) service-oriented architectures (execution), and (iii) process controlling (monitoring, evaluation).

An organization's core assets capture the essential properties without the value creation cannot take place. Organizations' value creation mainly base on information about their own value chain, customers, production, and research and development cycles [252, 370]. This information is captured in terms of data in different IT systems that combine the enterprise-wide data utilized in everyday work. All customer information, for instance, is usually stored in the customer relationship management (CRM) system. The organizations' actual value creation is performed by executing the organizations' business processes that depend on the information mentioned above. Thus, these business processes need access to the IT systems and its contained data to keep an organization operational.

The emergence of service-oriented architecture (SOA) [93] opened new horizons for *automated business process execution*, yet revealed new challenges [250]. SOA transforms enterprise landscapes slicing the functionality of large software systems into services. As services are capable of accomplishing atomic business tasks, they can be effectively used for task automation. Thereafter, SOA catered for automation of business processes. While control flow oriented business processes made the process routing logic explicit, data was still "hidden" inside IT systems. However, this data highly impacts process execution (also see above paragraph). For instance, many decisions in processes are data-driven. As a result, the role of data in process models grew significantly. It became essential to model the data and data flow within the process model.

For ensuring process quality by process controlling, key performance indicators (KPIs) [73, 106, 168] are measured and interpreted using business process intelligence [118, 234] techniques. KPIs reflect business goals of an organization. These goals are defined referring to data. Following, KPIs rely on data. For instance, one business goal might be to achieve the highest customer satisfaction in the market. This goal is reflected by several KPIs; one of them deals with delivery time. This number should be minimized for increasing the customer satisfaction. For process controlling, the activities contributing to this goal need to be identified for evaluating them. This is done by selecting the activities performing work on the appropriate data objects – in this case the order delivery. Thus, an explicit statement of data supports process controlling. Additionally, the data objects are considered for the evalua-

tion itself as well. State[1] and content changes provide, amongst others, insights with regard to the process progress and long lasting steps can be identified.

Summarized, making data in business processes explicit is required, since data

1. represents the actually manipulated artifacts during business process execution,
2. drives execution by influencing activity enablement and, in this context, describes pre- and postconditions of activities stating which information is required for and which information will be the result of activity execution,
3. appears in different states that describe (intermediate) processing results,
4. links business processes to IT systems, e. g., legacy systems,
5. is utilized for process control as core aspect of key performance indicators ensuring, for instance, process quality, and monitoring and evaluating business process execution,
6. shows core assets important for an organization's value creation, and
7. documents the actions undertaken.

Industry also identified the importance of data giving it some uptake by developing corresponding software systems and by adapting existing respectively creating new standards around data in business processes. The presentation of data in business processes especially helps to analyze business processes, to enact business processes, and to reason about business processes with process execution being the central application of data. In a 2009 Delphi study, automation – especially driven by process models – has been stated the *number one challenge* of process modeling in the future [145]. The increasing interest in process automation from industry and academia is justified because of the development of various BPMSs – from research institutions as well as large software vendors like SAP, IBM, Oracle – that may automatically enact the control flow of business processes based on model specifications. Since data is a core part of process automation, the data perspective gained severe attention in the remaining years.

An increased data awareness can also be observed considering the development of standards for BPM. For instance, the industry standard Business Process Model and Notation (BPMN) [243], established through the Object Management Group (OMG) and supported by many large companies, got heavily revised towards version 2.0 now providing "a standardized bridge for the gap between the business process design and process implementation" [243]. However, since few years, multiple paradigms exist regarding business process modeling. Besides the activity-centric paradigm – to which BPMN belongs –, the object-

1 A data state describes a specific business situation in the context of a data object. For more details, see Definition 4.4 on page 63.

centric – also referred to as artifact-centric – paradigm evolved driven by IBM research. An object-centric process model is modeled by its involved data objects; each object has a life cycle and multiple objects synchronize on their state changes. The actions performed to manipulate objects are "hidden" in state transitions specified in the object life cycles (OLCs) [154, 288]. These manipulations and finally also the synchronization between multiple objects are performed by some actors. Exploiting this idea, the Case Management Model and Notation (CMMN) standard [245] is developed through the OMG.

1.3 PROBLEM STATEMENT

Most existing process engines are activity-driven, e. g., [2, 31, 32, 45, 84, 149, 177, 340] and commercial products from major software vendors like SAP, Oracle, IBM, and Tibco. In these, the execution follows an explicitly prescribed ordering of domain activities not necessarily tied to data objects. Thus, the activity-centric paradigm is widely adopted in industry often utilizing – according to a study from 2013 [129] – the ARIS EPC notation [82, 157] or the business process execution language (BPEL) [240], ISO 9000 [148], Unified Modeling Language (UML) [244], BPMN [243] standards. In contrast, for the object-centric paradigm, only few implementations exists to date: an extension to IBM web sphere in a pharmaceutical context [28], Siena [55], Barcelona [132], and PHILharmonic Flows [48]. This is a strong indicator that the object-centric paradigm is not yet widely adopted. Additionally, it lacks major empirical evidence regarding applicability and usability. The benefits are shown for few individual cases only; e. g., for industrializing discovery processes in the healthcare domain [28] and for manufacturing processes where the process flow follows from process objects [172, 230]. Although the object-centric paradigm obviously has its benefits for several domains, in many domains, e. g., accounting, insurance handling, municipal procedures, and logistics, the processes are rather activity-driven. In addition, the activity-centric paradigm has been applied generically in the area of BPM.

Considering this fact in addition with the automation challenge [145] stated above already, the main research question of this thesis is formulated as follows.

> HOW TO AUTOMATICALLY EXECUTE THE DATA PERSPECTIVE FROM ACTIVITY-DRIVEN PROCESS MODELS?

Automatically executing process models requires (i) the representation of control flow and data information within the process model, (ii) a formalization of the model in conjunction with operational semantics, and (iii) correct process models. Thus, elaborating on this research question naturally gives raise to multiple sub-research-questions (SRQ).

Execution of a business process requires a multitude of information. Analysis of existing BPMSs shows that such information comprises, for instance, the ordering of activities, referenced automated services, utilized data objects and sources, the allocation of a user to a task – referred to as task performer or process participant – who actually performs the work during execution and user forms allowing interaction between the task performer and the system. Since the main research question tackles the data perspective, it is essential to know (SRQ-1) WHICH INFORMATION IS REQUIRED TO EXECUTE THE DATA PERSPECTIVE OF A PROCESS MODEL. Thereby, we differentiate between information already available in some model in the area of BPM and information that needs to be added with the goal of reusing – where appropriate – established concepts from other areas of computer science and software engineering.

Simple data dependencies can already be enacted from a process model by some activity-centric process engines, e. g., an activity can only be executed if a particular data object is in a particular state. However, when *m:n relationships* exist between processes and data objects, modeling and enactment becomes more difficult; e. g., in build-to-order scenarios, a computer manufacturer receives orders from multiple customers and purchases the parts to build the ordered computers from multiple suppliers such that there exist m:n relationships between customer orders and purchase orders through the computer parts. To these m:n relationships, we refer as *complex data dependencies*. Widely accepted process descriptions languages such as BPMN do not provide sufficient modeling concepts for capturing these m:n relationships. As a consequence, actual data dependencies are often not derived from a process model. They are rather implemented manually in services and application code. This yields high development efforts and may lead to errors during the configuration phase of the business process life cycle. An entirely model-driven approach, i. e., specifying data dependencies and the utilization in the process model, helps in reducing these efforts. In the ideal case, the configuration phase can be skipped and the process model directly be executed with respect to data. This goal of creating an entirely model-driven solution directly leads to the question: (SRQ-2) HOW CAN THE REQUIRED INFORMATION BE VISUALIZED IN PROCESS MODELS TO ALLOW GRAPHICAL MODELING OF THE DATA PERSPECTIVE?

Graphical visualization of data dependencies reduces error-proneness and entrance barriers for non-IT users such that actual process experts and domain experts collectively create the process models. This, in turn, increases the probability that executed business processes and the corresponding process models comply to each other. However, data additions usually increase model complexity that must be handled. Thus, for the sake of increasing process model quality but also for ensuring proper process execution, we question (SRQ-3) HOW TO SUPPORT STAKE-

HOLDERS IN CREATION OF PROCESS MODELS CONSIDERING THE DATA PERSPECTIVE. The previous research questions directly lead to the question on (SRQ-4) HOW TO ACTUALLY EXECUTE THE DATA PERSPECTIVE INCLUDING COMPLEX DATA DEPENDENCIES. Thereby, we aim at a generic solution to be applicable to at least most activity-centric process description languages. The execution requires a formal framework including operational semantics describing (SRQ-5) THE INTERPLAY OF CONTROL FLOW AND DATA PERSPECTIVES GENERICALLY. Currently, often either only one perspective is considered in an organization or both perspectives exist side by side without integration although they are *"[...] two sides of the same coin"* [277].

As sketched above, referring to technological support in BPM, there exist many different BPMSs and process description languages used in industry. The data perspective is generally important but often not supported satisfactorily [215] indicated, for instance, by missing support of workflow data patterns [289, 291, 374] presenting various ways how data is represented and utilized in business processes. Allowing stakeholders to decide on the usage of the BPMS as execution platform and the process description language, we aim on a generic support of the data perspective. Thus, the question arises (SRQ-6) WHICH TECHNOLOGY SHALL BE USED TO ENABLE INDEPENDENCE OF PLATFORM AND PROCESS DESCRIPTION LANGUAGE including thoughts on how generic the approach could be.

Proper execution requires a quality management to ensure that the automated execution works flawlessly. Especially due to the goal of an entirely model-driven support of the data perspective, the structural and behavioral correctness must be ensured. Such quality management is usually specified by correctness constraints leading to the next sub-research-question: (SRQ-7) HOW TO CHECK AND ENSURE CORRECTNESS OF THE PROCESS MODELS WITH RESPECT TO THE DATA PERSPECTIVE?

Derived from the business process life cycle, process models are used in a large variety of situations, e. g., analysis, execution, automation, communication, and monitoring. Further details on these fields of application follow in Chapter 2. While the first three mentioned aspects are (partially) targeted by the above sub-research-questions, the latter two require some discussion. Communication is an important factor in business process management [24, 144, 370]. Communication on business processes usually involves stakeholders requiring different views on the business process. The difference may occur on a horizontal or vertical scale.

Vertical refers to process refinement, i. e., adding additional information to the process model, or business process model abstraction (BPMA), i. e., reducing the level of detail by combining or removing detailed information. We tackle process refinement by adding required data information to a process model (see SRQ-2). BPMA is a reverse

operation not leading to an executable process model. Thus, we omit detailed discussions on data support in this thesis.

Horizontal refers to the representation of different parts of the business process, e. g., one model per major milestone to be reached towards the ultimate business goal of the process. Thereby, each model may utilize a different process description language or modeling paradigm. Additionally, horizontal also refers to the representation of the same information through different techniques providing deeper insights, e. g., showing the same business process by some activity-centric process model and some object-centric process model to see both perspectives. Representation of views on a process model or the underlying business processes strengthens communication capabilities. Depending on the use case and stakeholder, one view may be more suitable than an other. Thus, the question arises (SRQ-8) HOW CAN WE SUPPORT AND ENCOURAGE THE UTILIZATION OF VARIOUS VIEWS ON DATA?

Finally, process monitoring is a large research field in its own utilizing a multitude of information sources to provide insights on process execution. Thus, monitoring is closely related to process execution – also indicated by sharing the same phase in the business process life cycle: the enactment phase. Process monitoring also utilizes the data perspective since, for instance, specified key performance indicators usually refer to data objects. For instance, the creation, deletion, and manipulation (state changes) of data objects provide insights on their usage and timestamps between those changes help reasoning about the time passed. In this thesis, we focus on the modeling, design-time analysis, and execution of the data perspective. Utilization of data objects for process monitoring is subject to additional work.

1.4 CONTRIBUTIONS

To answer the main research question and its sub-questions formulated in the previous section, this thesis provides a novel approach on integrating the control flow and data flow perspectives for activity-centric business processes and allows their execution with respect to these two perspectives from the corresponding process models. Process models may be accompanied by a data model and object life cycle models globally, i. e., on business process level, specifying the data dependencies on a structural and behaviorally level respectively while process models may specify parts of the overall business process. All information required to execute a business process is derived and generated from explicitly visualized modeling concepts. Being on the edge between operational and implemented process models (see LOC-3 and LOC-4), we contribute in reducing the gap between these levels of control, since deriving execution information from the process models minimizes the actions to be taken in the configuration phase of the business process life cycle. Parts of this thesis have been published in two journal papers,

six conference papers, two workshop papers, one demo paper, and six technical reports, e. g., [212, 213, 220, 223, 224]; for a full list of publications, we refer the reader to page vii and following. Each chapter begins with stating the publications where parts of the chapter have already been published before. Complementary, the thesis summarizes and extends the results of these works and presents them on a higher level by linking the individual contributions. Below, we explain the contributions of this thesis in more detail and relate them to the research questions presented in Section 1.3.

(1) Model-driven Business Process Execution

This thesis provides an entirely model-driven approach to execute control flow and data flow perspectives of activity-driven processes. Targeting sub-research-questions SRQ-1 and SRQ-2, we determine the required information and provide a modeling guideline to incorporate this information into process models utilizing concepts mainly from the business process management and database domains. Given such enriched process model, we specify operational semantics allowing the execution of the represented business process (see SRQ-4). Thereby, we utilize the semantics from common activity-centric process engines as basis and add features for handling complex data dependencies, retrieving and storing data, and allowing communication between different processes. For these data aspects, we utilize standard technology, e. g., Extensible Markup Language (XML) and XML Query Language (XQuery), and encapsulate them in isolated modules that exist orthogonally to the existing activity-driven process technology and thus, can easily be added to standard process engines at well defined and standardized locations. In this thesis, we provide extensions that are generically defined such that, for application, the process description language can be chosen freely. The utilization of standard technologies and generic process descriptions lead to a quasi independence of platform and process description language (see SRQ-6).

(2) Formal Framework for Process and Data Integration

Specification of operational semantics requires formal underpinning of the utilized concepts to unambiguously state the behavior. Thus, we introduce a formal framework on the integration of process and data perspectives in business process management. The framework formally represents all information required for process execution including different types of models, e. g., process model and data model, tackling sub-research-questions SRQ-1, SRQ-2, SRQ-4, and SRQ-5. The operational semantics is specified through a mapping to Petri nets [253] that extends an existing mapping of control flow concepts [80].

(3) Data Flow Correctness

To ensure the correctness of process models (see SRQ-7) with respect to their data specifications, we introduce the notion of *weak conformance* that generalizes the concept of object life cycle conformance [176]. Weak conformance ensures that each data manipulation specified in a process model refers to some set of data state transitions (data manipulations) specified in the object life cycles accompanying the process model. Thus, the OLCs act as reference for data manipulations in process models. In this thesis, we show how to identify violations and how to correct them. An additional benefit of our data flow correctness approach is that we build on concepts for control flow correctness and thus, we allow to finally check for control flow and data flow correctness at once.

(4) Data Extraction from Control Flow

Execution requires sufficiently annotated process models. This especially includes the data manipulations performed by activities since they are usually not modeled in process models. Instead, in practice, process experts concentrate on the control flow and postpone data handling to the configuration phase of the business process life cycle. In this thesis, we offer support to the process modelers by annotating the process models with data information that is extracted from the modeled control flow, i. e., activities and their logical and temporal ordering, (see SRQ-3). The resulting, data-annotated process model provides – depending on the process quality – some insights of data utilization and helps in specifying the data perspective in the process model graphically (see SRQ-2).

(5) Model Transformations

The first step towards encouraging stakeholders utilizing the benefits from different views on business processes (SRQ-8) is the provision of these multiple views. Therefore, we provide two types of model transformations affecting views: *inter-view* and *intra-view* transformations summarized in contributions (5a) and (5b) respectively.

(5a) View transition. Inter-view transformations derive one view on a business process from another one by preserving the given model information. Two extremes in this regard for process modeling are the activity-centric and the object-centric paradigms. Both represent similar information with completely different views. We provide a roundtrip transformation between both extremes through a mediator: object life cycles (OLCs). This allows to change the paradigm, the process model is created and visualized with, multiple times during the process modeling process. Additions and modifications can be applied in any of these views allowing stakeholder utilizing the appropriate method depending on their current task; e. g., specifying the order of activities is

usually best done with the activity-centric view while data manipulations can easily be added by, for instance, adapting the OLCs.

(5b) View adaptation. In addition to changing the paradigm a view bases on, adaptations to the current view provide new insights or help in understanding the process. We provide two adaptation options: First, we allow reduction of large OLCs that may cover organization-wide data manipulations towards the information required for a given process model highlighting the important data manipulations. In contrast to deriving the OLC view from the process model, the current structure and given labels of the OLC are preserved during the view adaption. Second, we allow refinement of a given process model towards coverage of the data manipulations given in an OLC by extending the process model again preserving control flow structure and labels. This supports the process designer in the creation process towards implementable process models also targeting sub-research-question SRQ-3.

1.5 STRUCTURE OF THESIS

We conclude this chapter with an outline of this thesis and the correspondences between contributions stated in Section 1.4 and the chapters in this thesis. Figure 2 provides an overview about the structure of this thesis with rectangled numbers referring to the chapter the corresponding concept or technique is described in. The thesis consists of three parts which contain three, four, and two chapters respectively. Note that Chapters 5 to 8 are self-contained and mostly independent with few but explicitly stated inter-dependencies. Thereby, Chapter 8 presents the actual solution for the main research question while Chapters 5 to 7 present techniques and concepts supporting in creating the models for model-driven process execution. Next, we discuss each part's contents and relate them to the contributions.

Part I: Background

The first part of the thesis focuses on preliminaries. Chapter 2 introduces several fields of application for process models, discusses the support of data in chosen process description languages, and introduces BPMN as example language used for process model visualization throughout this thesis. Afterwards, we present the running example: a build-to-order and delivery process. Chapter 3 lays the foundation of this thesis by briefly discussing concepts from literature that are important in the course of this thesis.

Part II: Hybrid Process Model for Data and Control

The second part first presents the conceptual framework to integrate process and data views in Chapter 4 providing the solution for contribution (2). Process model and control flow refer to activity-centric process

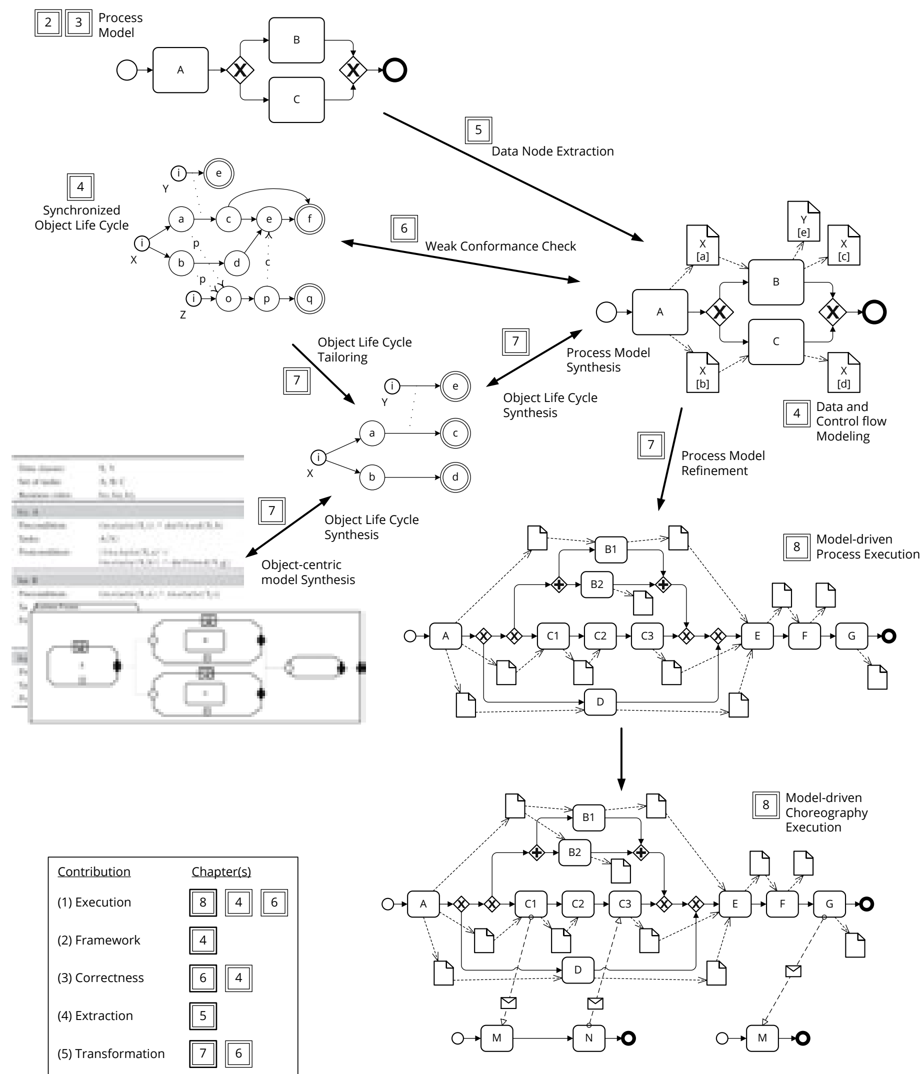


Figure 2: Structure of this thesis with numbers in double-bordered squares indicating the chapter where the corresponding concept or technique is discussed.

models throughout the entire thesis if not stated otherwise. We introduce the data side and the process side of process models formally and provide integrated execution semantics through a Petri net mapping as part of contribution (1). Thereby, we introduce the concept of *synchronized object life cycles*; these are OLCs containing inter-dependencies restricting the execution of some data state transitions until the dependent OLC reaches the required state.

In Chapter 5, we provide generic algorithms that allow to extract data information from control flow structures and activity labels and to annotate this information to the activities in terms of data nodes (see contribution (4)). In addition, we also show how to adapt the generic algorithms towards a chosen process description language to make use

of language-specific control flow information strengthening the results of data extraction.

Afterwards, Chapter 6 introduces the notion of weak conformance, an approach to compute weak conformance by utilizing the Petri net mapping from Chapter 4 and soundness checking, and means to correct identified violations to ensure correct process models from the control flow as well as data perspectives (see contribution (3)).

Chapter 7 presents algorithms to transform process models and object life cycles. On the one hand – targeting contribution (5a) – we provide algorithms allowing a roundtrip between activity-centric and object-centric process models via object life cycles that act as mediator by entirely synthesizing one representation from another one. On the other hand – targeting contribution (5b) – we provide algorithms allowing a refinement (in terms of increasing the level of detail) of a process model based on OLC information and tailoring, i. e., reducing, a given OLC based on process model information. Both algorithms require the notion of weak conformance (see Chapter 6) as pre-step to ensure alignment between the process model and the object life cycle.

Part III: Automated Process Model Execution

Chapter 8 introduces the solution to contribution (1) and thus, the solution answering the main research question of this thesis. Summarized, Chapter 8 introduces modeling guidelines and operational semantics (basing on the ones described in Chapter 4) for both process orchestrations (internal processes) and process choreographies (process interactions) to execute the corresponding business processes entirely model-driven. Therefore, we allow code generation (i) to handle the existence, retrieval, and storage of data including complex dependencies and (ii) to handle automatic message exchange between multiple process participants. In addition to the modeling guideline and the operational semantics, we also provide an extensive overview about correctness of process orchestrations and process choreographies ensuring proper automatic execution from sufficiently annotated process models. Here, we collect a large body of existing research and explain how to utilize the correction mechanisms in the context of this chapter. One of the correction mechanisms is the notion of weak conformance introduced in Chapter 6.

Finally, Chapter 9 concludes the thesis by summarizing its contributions, discussing the application of data in further BPM areas, summarizing limitations and open problems, and providing future research capabilities.

2 PROCESS MODELS

PROCESS MODELS are an abstraction of the real world [169] describing the logical and temporal partial order of activities [24, 370]. Process models represent business processes with specific focuses. Thus, their creation follows a defined purpose [313]. Additionally, creating a process model is influenced by the used modeling technique that consists of a process description language and a modeling methodology. A process description language is specified by syntax (a grammar specifying the usage of modeling concepts), semantics (describing the behavioral properties), and a notation (visualization of modeling concepts) [128, 362]. A modeling methodology defines a procedure on how to use the modeling language [362].

Thereby, a process description language may follow multiple modeling paradigms with activity-centrism and object-centrism as two extremes. Activity-centric process modeling origins from traditional workflow modeling and centers around the logical and temporal partial order of activities. Object-centric modeling mainly emerged from the need of enhanced process flexibility and centers around the manipulation of objects. Currently, activity-centric process models (ACPs) are widely used in practice while object-centric process models (OCPs) start emerging. In this thesis, we build on the activity-centric paradigm and incorporate ideas from other areas of the spectrum to finally move towards an integrated modeling. In this respect, we informally define a process model according to [370] as follows.

Definition 2.1 (Process Model (informal))**.**
A *process model* specifies a set of actions and their execution constraints consisting of the actions' logical and temporal partial ordering, data that gets manipulated, and resources performing the actual work. Additionally, the model's embedding in the technological environment is specified. A process model acts as blueprint for a set of cases and is used for business process execution. ◂

In the remainder of this chapter, we first discuss the application of process models for different purposes in the field of business process management (BPM) in Section 2.1. Afterwards, we review the data support in process models in Section 2.2 and provide a brief introduction to the process description language Business Process Model and Notation (BPMN) that is used for representation purposes throughout the thesis in Section 2.3. Finally, in Section 2.4, we introduce the running example of this thesis: a build-to-order and delivery scenario.

2.1 APPLICATION OF PROCESS MODELS

Application of process models

Process models are created and used for a multitude of purposes. Next, we review some of them exemplarily that are of interest with respect to data utilization. Thereby, we do not aim for completeness but intend to illustrate the spectrum of potential applications.

PROCESS COMMUNICATION. One key purpose of process models is to document the way business operations are conducted to reach a consistent understanding and thus to improve communication among different stakeholders [24, 370]. This comprises various perspectives on the process model including control flow, resource, and data perspective. Different stakeholders require different views on the business process. Process models help here to bridge this gap and allow a discussion across multiple levels of responsibility. Such communication is mainly of importance during the design and analysis phase of the business process life cycle, since communication and subsequently a consistent understanding finally supports proper process execution. Thereby, the process models are used as means for knowledge management. Furthermore, process understanding and communication are among the most perceived benefits of business process management due to a study published in 2009 [144].

PROCESS ARCHITECTURES. Business process architectures (BPAs) provide an overview on the interactions, i.e., dependencies, between process models and adds to communication on business process level (multiple process models may represent one business process) and organizational level instead of process model level as discussed above. Dependencies between process models are due to control flow, also referred to as event flow, [79, 86, 287] or data [87] flow. In the latter, data objects show the partial ordering of process models, e.g., one model produces a result that is than the input to another process model leads to a sequence dependency.

PROCESS VIEWS. As already mentioned, communication follows different purposes and includes multiple stakeholders on different

levels of responsibility. Each purpose and each stakeholder requires different information in the process model while communicating about the same business process – referred to as views. Views can be created in multiple ways. Abstraction [96, 255, 309] and refinement [162, 164] are means to change the level of detail of a process model for the same (part of the) business process. Additionally, transformations between different process description languages [190, 247, 367, 371] are a classical mean to cater a view with respect to, amongst others, the utilized information technology (IT) system, the knowledge of the interested stakeholder, the degree of formalism in the representations.

PROCESS ANALYSIS. Process analysis is an important step during process design and preparation for process execution to ensure proper execution of the business process. Thereby, formal process descriptions have a higher potential for precise analysis [346], since their syntax and semantics is well defined and violations can be identified by checking for deviations from specification. Formal analysis is usually divided into *verification* and *validation*. Verification checks for structural and behavioral consistency tackling properties like correctness [18, 331], compliance [1, 9, 15, 117, 279], and risk management [161, 388]. Validation checks whether the process model meets the requirements and needs of all stakeholders. Common validation methods are conducting workshops and performing process simulation [192, 297, 370]. Simulation is additionally used for quantitative analysis and forecast to analyze newly created or adapted business processes before deployment in practice [83]. Therefore, the corresponding process models get annotated with run-time specific data, e. g., execution costs and time, available resources, and instantiation frequencies. From these simulation results, needs for process improvement are derived.

PROCESS EXECUTION. Process models act as blue print for process execution [82, 311]. They specify which activities are executed next based on the current status of the process execution, they specify which data objects are utilized for activity execution and which will be the result upon termination, and they specify who may execute an activity from which then one is finally chosen to actually execute the activity. Process execution can be done manually without IT system interaction, manually with IT system support or guidance, manually while actively using IT systems, and completely automatically through an IT system, e. g., a workflow or process engine [2, 45, 184, 320]. From these, especially process automation bridges the gap between requirements specified in the process models and the system specification utilized for automated process execution [35, 119, 282]. Thereby, the process

model predefines the behavior the process engine then orchestrates. Data is one of the main drivers during execution, since it represents the artifacts being actually processed (also see Section 1.2); process automation from process models is considered the number one challenge in the future [145].

PROCESS COLLABORATION. During execution, participants from different business processes (from probably different organizations) may require to interact to, for instance, synchronize execution or exchange information. These interactions are done through message exchanges. Thereby, messages contain data that is sent from one participant to the other – often referred to as process choreographie [69, 350]. A cross-organizational integration can be supported by IT systems as discussed above for process orchestrations. Additionally, in industry, process collaboration is also used to outsource single activities or complete business processes leading to distributed value chains that need to be coordinated [38, 61].

PROCESS MONITORING. Process monitoring is about observing the execution. It can be done for manually as well as automatically executed business processes. For completely manual ones, additional effort must be performed since monitoring systems cannot directly monitor activity execution. Instead, the process participant has to manually report activities or systems have to be installed that observe the manual execution [134]. If IT systems are involved, the actions can be tracked automatically and stored in a log. Additionally, information can be retrieved from side effects observed in IT system as, for instance, manipulated data objects or exchanged messages [136]. Generally, monitoring allows the provision of real-time information to customers and process stakeholders about current the state of the process and the involved data objects. Continuous monitoring allows early identification of and reaction to issues and inconsistencies, e. g., delays in process execution, usually identified through event processing [17, 23, 36, 42, 229].

PROCESS IMPROVEMENT. The process improvement is considered the top benefit of process modeling in practice [144]. It targets many criteria that are supposed to result in, most prominently, reduced costs, time, and resource consumption as well as increased quality and flexibility. Often, the basis for process improvement are results from performance measurements where data plays a major role, since most criteria base on data information; e. g., the delivery time of an order is measured by knowing the time the order is received by the company and the time the ordered products actually reach the customer while measuring the order processing time utilizes the time the package leaves the company – in

both cases, information related to the data object order are used. Process improvement leads to *to-be* process models. One procedure to do so is process redesign which takes the *as-is* state of a process model as input [63, 271, 272]. Continuously improving the process models allows quick reactions on changes in the environment, e. g., law changes and new guidelines. Thereby, small but frequent improvement iteration cycles require also quick implementation Otherwise, the changes may be obsolete before they were put to action. Thus, process implementation shall follow process modeling [311], for instance, by enacting the business process from the process model.

PROCESS MINING. Process mining [335, 336] "extract[s] knowledge from event logs recorded by an information system" through provision of "techniques and tools for discovering process, control, data, organizational, and social structures" [82]. Often, process mining is used to construct process models from the log information and to compare the constructed process models with the actually modeled ones for identifying deviations; e. g., identifying compliance violations. The event logs can also be used to determine performance measures which then may be input for process improvement to another analysis and design phase in the business process life cycle.

2.2 DATA SUPPORT IN PROCESS MODELS

Data is relevant for multiple aspects in business processes (see Section 1.2 and fields of application of process models. Thus, data should be presented in process models. In this section, we briefly review an in-depth analysis on data support in multiple process description languages. Details on the evaluation framework and the corresponding results are given in [216].

Process description languages can be classified according to their data support at both design-time and run-time ranging from *control-flow-driven* to *data-driven* models with three major levels in-between those extremes. Figure 3 visualizes the five levels scale. We consider the *control-flow-and-data-driven level* as goal state of a process description language indicated by a star, since it then equally captures control flow as well as data information required for (automated) process execution.

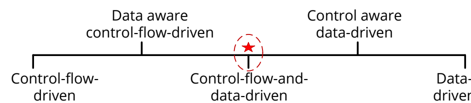


Figure 3: Scale for the level of data support in process description languages.

Only few process description languages are classified on one of the extreme levels. On the data side, these are, for instance, document-

driven workflows [363] and case handling [347]. On the control flow side, these are, for instance, workflow nets [332] although there exist approaches introducing data into workflow nets, e. g., so-called WFD-nets [325]. Most of the approaches evolved in the context of object-centric processes are data driven and also aware of basic control flow information. Few, e. g., Corepro [232] and PHILharmonic Flows [173], can be classified on the goal level. From process description languages specifying activity-centric process models, most are classified as *data aware control-flow-driven* and none is classified on the goal level. Thereby, the data awareness remains shallow usually only supporting simple data dependencies as 1:1 or 1:n relationships if at all and missing integration of object life cycles as the object-centric process description languages usually do. Considering the contributions of this thesis and applying them to activity-centric process description languages such as BPMN, the classification changes and such activtiy-driven process description languages would then be classified on the goal level *control-flow-and-data-driven*. Indeed, the goal level can be reached from both sides as shown by, for instance, PHILharmonic Flows that comes from the object-centric, i. e., data-driven, side. However, reaching it from the activity-centric, i. e., control-flow-driven, side allows reuse of established concepts and proven analysis techniques from traditional workflow modeling.

2.3 BUSINESS PROCESS DESCRIPTION LANGUAGE: BPMN

Next, we briefly introduce BPMN [243] in its current state of data support as example process description language, since we utilize BPMN for representing process models throughout this thesis. An exhaustive discussion is beyond the scope of this chapter. Thus, for introductions to BPMN, we refer the reader to [49, 110, 306, 370, 372]. We chose BPMN for representation because of its widespread application and interest in practice. According to a recent study from 2013, 60% of the over 300 participants stated that BPMN is of interest for adoption in their company; in total BPMN received more votes then the standards placed two and three together (multiple selections where allowed).

BPMN was initially proposed by the Business Process Management Initiative (BPMI) and got later standardized by the Object Management Group (OMG). Currently, BPMN is available in version 2.0. BPMN contains multiple diagram types for business process modeling. *Conversation diagrams* are used to model high-level interactions between multiple organizations by defining communications, i. e., sets of message exchanges, between the participants. Conversation diagrams can be detailed by *choreography diagrams* by adding tasks responsible for a certain message exchanges. Third, BPMN allows modeling of global and local *collaboration diagrams*. Global collaboration diagrams present the information of choreography diagrams in a different style focusing

on the control flow integration of the exchanged messages. Local collaboration diagrams show the internals of business process execution and are usually not shared across organization borders. They can be seen as the core of business process modeling in practice. In this thesis, we concentrate on collaboration diagrams and there, mostly on the local ones.

BPMN collaboration diagrams provide an exhaustive set of modeling elements. *Activities* capture single steps within business operations. An activity is represented as rounded rectangle in the business process model. They can be atomic – called task – or a hierarchically structured collection of atomic tasks – called subprocess and indicated by a *plus* at the bottom middle of an activity. An activity may have multiple types as indicated by special markers in the top left, e. g., automatic (service) task with two gear wheels, a manual task with a hand icon, a send task with a white envelope, and a receive task with a black envelope. *Events* allow modeling the occurrence of real-world happenings represented as circle with one or two lines depending on the event type. An event may be of type start, intermediate (often interrupting process execution), and end. As second property, an event consists of a trigger, e. g., send or a receive a message, a timeout, and an exception. *Gateways* allow modeling of routing behavior including decision structures represented by diamonds with special markers. Thereby, BPMN allows conjunctions (plus marker), exclusive disjunctions (no marker or cross marker), inclusive disjunctions (circle marker), event-based exclusive disjunction realizing a deferred choice [345] (pentagon marker), and arbitrary routing behavior specified for each gateway individually (star marker) referred to as complex gateway. Activities, events and gateways realize the control flow perspective of a BPMN model.

In BPMN, *pools* and *lanes* are used to model organizations and roles that are assigned to activities taking care of the resource perspective. Tackling the data perspective, BPMN allows modeling of *data objects* (document shape) that are associated to activities representing read and write operations and *data stores* (cylinder shape) representing access to storage that exists independently from process instances. Data objects can have assigned *data states* representing business situations and indicated by string surrounded by [and] below the data object name. Messages (in global collaboration diagrams) are represented by envelopes. Data objects and activities can be multiplied (multi-instance, list) indicated by three bold dashes at the bottom middle of the corresponding element.

Control flow edges connect two control flow constructs (activity, event, gateway). *Data flow edges* connect a control flow construct and a data flow construct (data object, data store). *Message flow edges* connect send events/tasks contained in one pool with receive events/tasks in another pool representing message exchange. Thus, BPMN collaboration diagrams have a graph structure.

With respect to operational semantics, BPMN provides informal token flow semantics basing on further formalization efforts, e. g., [80, 359].

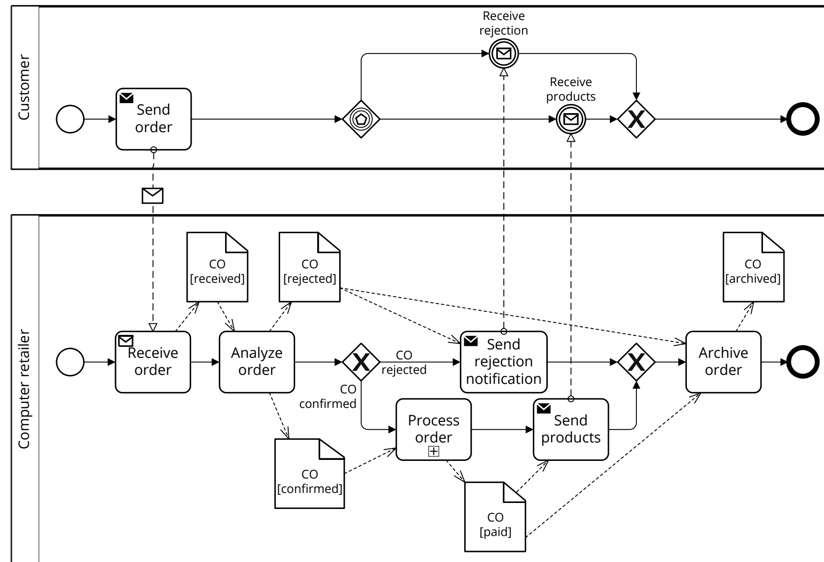


Figure 4: Example representing a build-to-order and delivery process as BPMN collaboration diagram.

Figure 4 visualizes the notation of BPMN by a condensed example of a build-to-order and delivery process. The figure shows two pools (the *Customer* and the *Computer retailer*) acting as the participants in this collaboration diagram. The *Customer* sends an order to the *Computer retailer* and waits for a response indicating an order rejection or the arrival of the ordered products. Since the corresponding path in the process model is triggered upon retrieval of an event, the disjunction is handled by an event-based gateway. The computer retailer receives the order and based on analysis results, the order is either rejected or confirmed and thus processed. In the former, a rejection notification is sent to customer. In the latter, after processing the order, the products are eventually sent to the to customer. The order processing is hidden in a subprocess. The remaining activities are all atomic tasks.

2.4 SCENARIO: BUILD-TO-ORDER AND DELIVERY PROCESS

In this thesis, we utilize a build-to-order and delivery process that bases on an abstract real-world business process. Three participants are involved in this build-to-order and delivery process with different perspectives: A *Customer* may order custom-built products from a *Computer retailer* who in turn purchases the required components from various *Suppliers*. Generally, these components or the built products are not held in stock. But due to returns and order cancellations after procurement of components, few items may be held in stock and

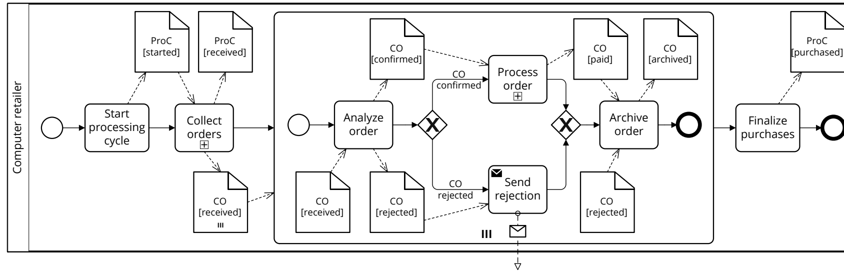


Figure 5: Running example: Build-to-order and delivery process from *Computer retailer*'s point of view.

can then be used for order processing without purchasing them from the supplier. Figure 5 shows this process from the computer retailer's point of view and gets detailed through expanding the contained subprocesses in Figures 6 to 11. In the remainder of this thesis, we utilize this process orchestration or parts of it as running example to explain and visualize the introduced concepts. For completeness reasons, we briefly show the process orchestrations from the customer's and the supplier's point of views in Figure 13 and Figure 14 respectively. In the scope of this thesis, we will utilize the interactions between the different participants to show enactment of data exchanges (cf. Section 8.4). For visualization of process models in this thesis, we utilize BPMN. However, many other process description languages may be used alternatively, e. g., Unified Modeling Language (UML) activity diagrams, event-driven process chains (EPCs), and Yet Another Workflow Language (YAWL).

Since the computer retailer's view is the driving one for this thesis, we concentrate on the introduction of the computer retailer's actions to achieve the goal of successfully selling custom-built computers and receiving the respecting payment. First, the computer retailer needs to *start* a new *processing cycle* (*ProC*) indicating that the shop is open and she is ready to receive new orders from customers. To reduce costs, component purchases are bundled for multiple customer orders. Thus, secondly, *customer orders* are collected in a loop structure as detailed in Figure 6 until a sufficient number of orders was received. After retrieval of a *customer order CO*, a subprocess is started that analyzes the received order and extracts the *components CP* required to build the computer and determines for each component from which supplier it may be purchased. The last task within the loop structure checks whether the number of received orders reached the required minimum. If so, the order retrieval is stopped for the current processing cycle and the corresponding data object advances to the corresponding data state[1] *received*. Otherwise, the next loop iteration is triggered to receive additional customer orders.

1 A data state describes a specific business situation in the context of a data object. For more details, see Definition 4.4 on page 63.

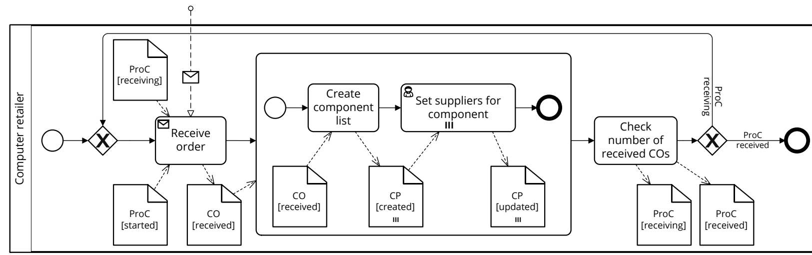


Figure 6: *Collect orders* subprocess from Figure 5.

After collecting the customer orders, a set of actions is undertaken for each received customer order *CO* indicated by the multi-instance subprocess in Figure 5. First, a customer order gets analyzed. This results in a decision whether the order is *confirmed* or *rejected*. In case of rejection, the corresponding message is sent to the customer. In case of confirmation, the customer order *CO* gets processed as detailed in Figure 7 resulting in a *paid* customer order *CO*.

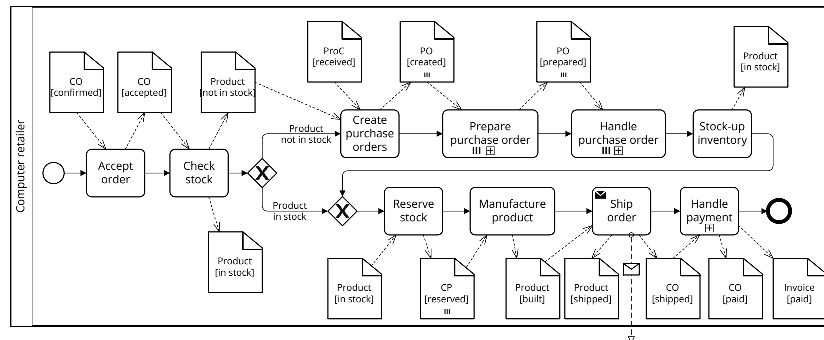


Figure 7: *Process order* subprocess from Figure 5.

During customer order processing, the customer order *CO* gets accepted. In practice, this step may also trigger a rejection due to, for instance, an untrustworthy customer. This step considers the context of an order while the order analysis executed before checks the order itself. Thus, both activities focus on different aspects to ensure satisfactory customer order handling. In the scope of this thesis, we abstract from these details and induce an acceptance of the customer order as only option. Then, we check the stock whether the components required to build the ordered product are in stock since these may exist sometimes as discussed in the beginning of this section. If so, four activities can be skipped. They are only executed if the components are not in stock indicated by data object *Product* to be in data state *not in stock*. These activities allow the *creation of the purchase orders* required to procure the components, the *preparation of the purchase orders*, their *handling*, and finally *stocking-up the inventory* with the corresponding components such that the product to be built is in data state *in stock*. A purchase order contains information about the component and its quantity to be delivered. This leads to one *purchase order* per *component* required to

build the product of some received *customer order*. The preparation and handling of the purchase orders is described in Figures 8 and 10 respectively. Since both are multi-instance subprocesses, the process models comprise the actions undertaken for each *purchase order PO*.

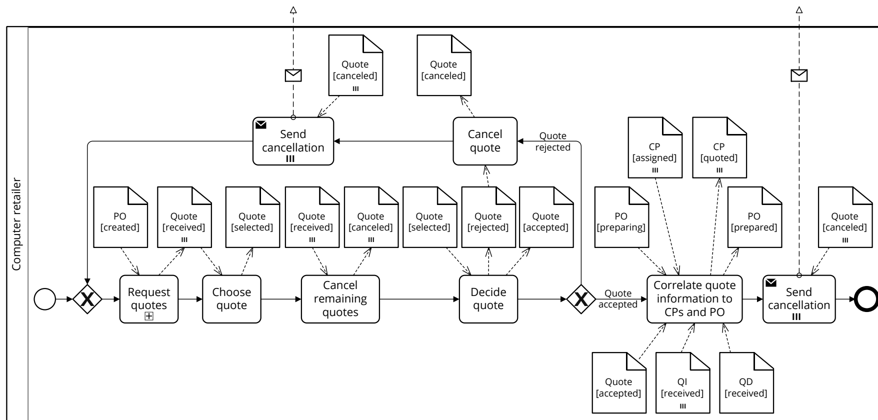


Figure 8: *Prepare purchase order* subprocess from Figure 7.

The purchase order preparation comprises a request for quote followed by a decision which quote to take. Therefore, the quotes are requested in the first subprocess that is detailed in Figure 9. For the given *purchase order PO*, a set of requests is created – one for each supplier a quote is expected from. The request preparation is done in the subsequent multi-instance subprocess in which first the targeted supplier is to be specified locally to the subprocess before the corresponding component is assigned to the request. To identify the correct component, a not yet assigned component which contains the specified supplier as specified in task *Set suppliers for component* in Figure 6 is randomly chosen. After preparing all requests for one purchase order, they are sent to the respective suppliers from which the corresponding quotes are received afterwards. This is again managed within a multi-instance subprocess. Each response consists of the *Quote* itself, additional *quote details QD* and respecting *quote items QI*. Thereby, the quote contains meta information, e. g., state of the quote and customer status as gold customer, the quote details contain general information, e. g., total price

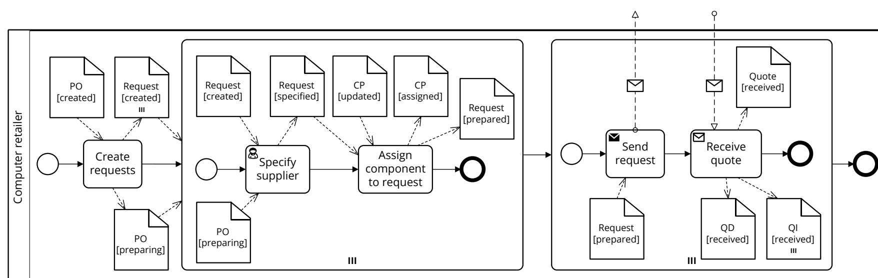


Figure 9: *Request quotes* subprocess from Figure 8.

and delivery date, and the quote items contain component specific information, e. g., type, quantity, and component price.

Next, the received quotes are analyzed and one of them gets chosen. The remaining quotes get canceled and for the selected one, a decision needs to be taken. Either this quote gets rejected such that it is canceled as the others before. Or this quote gets accepted. Then, the quote information gets correlated to the component and purchase order objects to indicate which quote shall be utilized during handling of the purchase order. In both cases, suppliers get notified about a quote rejection in terms of a cancellation message. This loop structure is iterated until one quote was accepted.

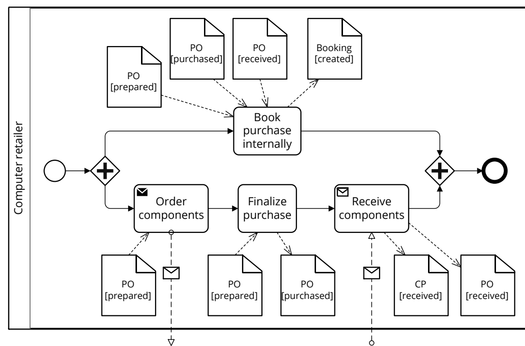


Figure 10: *Handle purchase order* subprocess from Figure 7.

The multi-instance subprocess *Handle purchase order* is detailed in Figure 10 and consists of a single AND block with two interleaving paths. On the one hand, the components are ordered following the given purchase order *PO*. Afterwards, this paths *finalizes the purchase* and waits until the ordered components are received indicated by data objects *CP* and *PO* each in data state *received*. Meanwhile, at any point in time, the purchase is booked internally for accounting reasons.

After component retrieval and internal booking of the purchase, these components are put into stock resulting in data object *Product* being in data state *in stock* (see Figure 7). This concludes the four optional activities to be executed if the *Product* is not in stock while checking. Next, the corresponding components *CP* get reserved before they are utilized to *Manufacture* the *product*. If the *Product* is successfully *built*, it can be shipped to the customer. Finally, the payment of the order must be handled. This is detailed in Figure 11.

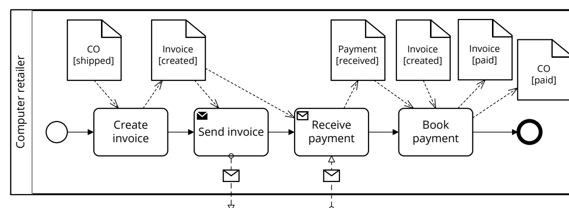


Figure 11: *Handle payment* subprocess from Figure 7.

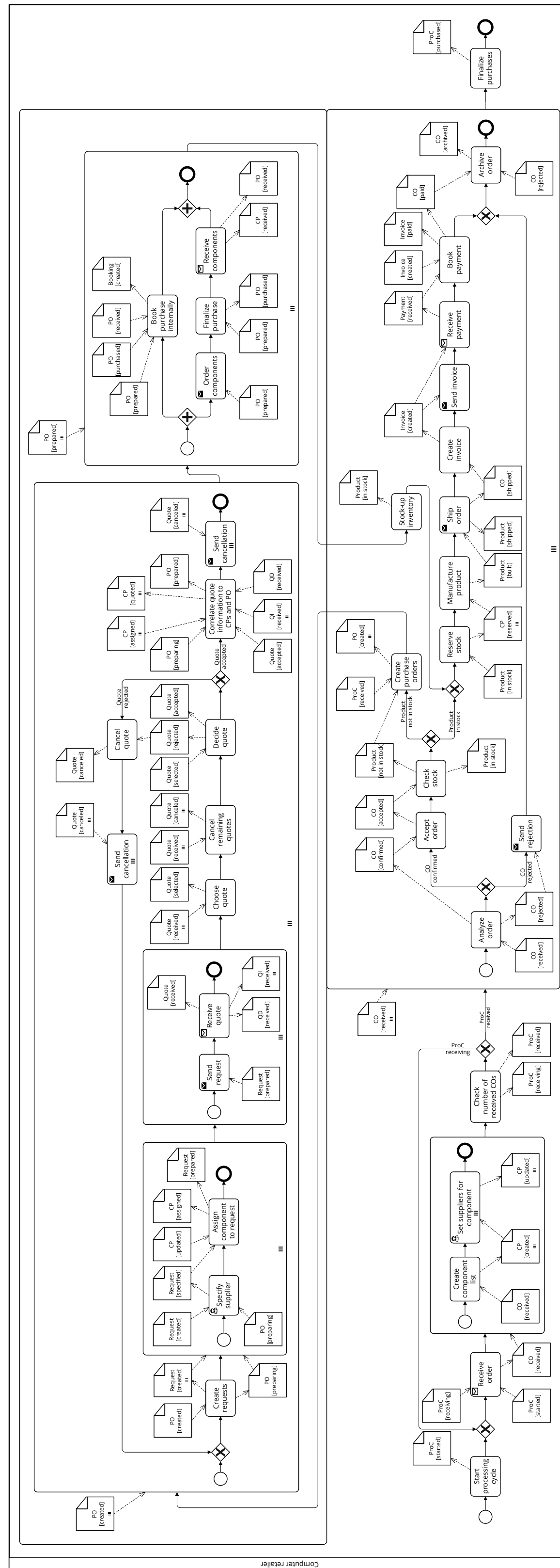


Figure 12: Running example: Complete detailed process model from computer retailer's point of view omitting message flow for readability reasons.

First, the *Invoice* is *created* based on the *shipped* customer order *CO*. Then, the invoice is sent to the customer followed by receiving the corresponding payment. In practice, several additional activities are involved in the payment handling as, for instance, re-sending the invoice and sending dunning letters. For this scenario, we abstract from these details. Upon retrieval of the payment, it is booked and the corresponding data objects *Invoice* and *CO* are both set to data state *paid*. This concludes the *Process order* subprocess in Figure 5. Afterwards, depending on the path taken, the *paid* or the *received* customer order *CO* gets archived followed by activity *Finalize purchases* which concludes the build-to-order and delivery process by setting data object *ProC* to data state *purchased* indicating that each customer order has been rejected or processed successfully.

A business process can be represented by set of process models as done above or by a single process model as presented in Figure 12 for the build-to-order and delivery process from the *Computer retailer*'s point of view. Both views show the same business operations but for increasing the single model's readability, we omitted the representation of message flow.

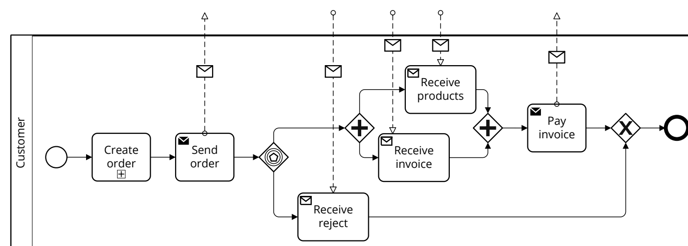


Figure 13: Running example: Order and delivery process from *Customer*'s point of view.

Figure 13 shows the build-to-order and delivery process from the *Customer*'s point of view. For readability reasons and to reduce model size, we omitted the data objects. First, the *Customer* creates the order within a subprocess and then sends the result to the *Computer retailer*. Based on actions taken on the retailer's side and the type messages sent, either the upper or the lower branch is triggered by the event-based gateway. The lower branch receives the reject notification while the upper branch receives the ordered product and the invoice in any order. After receiving products and invoice, the *Customer* pays for the order.

Figure 14 shows the build-to-order and delivery process from the *Supplier*'s point of view. Upon retrieval of a request for quote from the *Computer retailer*, the quote is created based on given information and then send back to the *Computer retailer*. Afterwards, the *Supplier* waits for a response whether the quote is accepted. Receiving a confirmation in terms of the actual purchase order, the *Supplier* produces the articles and sends them to the *Computer retailer*.

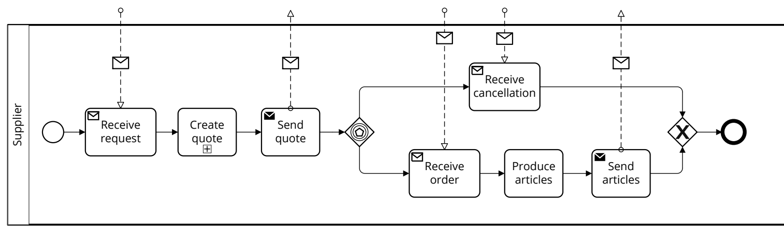


Figure 14: Running example: Build-to-order and delivery process from *Supplier*'s point of view.

Please note, the formal framework introduced in this thesis does not include event-based gateways since they are not widely supported in different process descriptions languages and since they can often be re-modeled with exclusive disjunctions with corresponding branching conditions – Figures 15 and 16 show this for the customer and supplier process models. Thereby, messages can be interpreted as read data objects that are required for process execution. The concepts and techniques described in this thesis are compatible with event-based gateways; they are handled analogously to exclusive gateways.

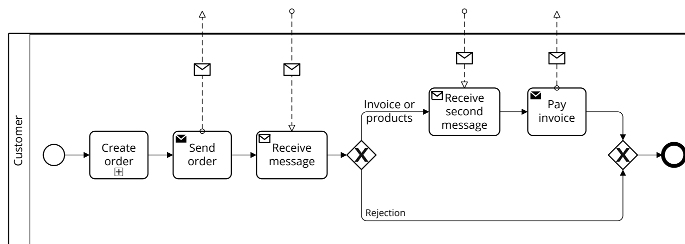


Figure 15: Running example: Order and delivery process from *Customer*'s point of view re-modeled with XOR gateways (compare with Figure 13).

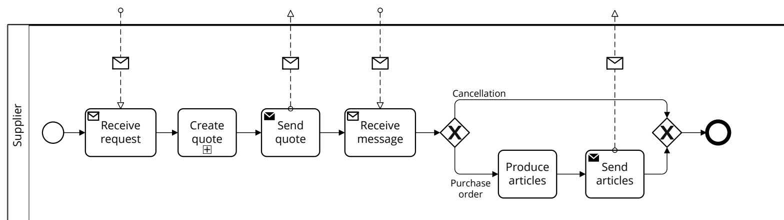


Figure 16: Running example: Build-to-order and delivery process from *Supplier*'s point of view re-modeled with XOR gateways (compare with Figure 14).

3 FOUNDATION

BEFORE we proceed with the discussion of the integration of data and control flow aspects of business processes and its application in several phases of the business process management life cycle, we formally introduce fundamental concepts in this chapter. Readers familiar with business process management (BPM) fundamentals can move to Chapter 4. However, we recommend to read at least the introduction to this chapter since we set some terminology and discuss the utilization of various letters within formulas. Starting from page 359, we provide a list of symbols summarizing the symbols utilized in this thesis and providing their explanation.

First, starting with Section 3.1, we start with the discussion of process models – their syntax and execution semantics – as widely used in literature, e. g., [370], before discussing behavioral relations within process models. Afterwards, we introduce net systems as formalism to verify various properties of process models, e. g., structural and behavioral correctness. Finally, we discuss the formal concepts generally used for model checking that verifies a business process' consistency to a given set of rules and regulations. As prerequisite, we recall basic mathematical notions used throughout all formalizations within this thesis.

$\mathbb{N}$ denotes the set of natural numbers including zero. $\mathbb{N}^+$ denotes the set of positive natural numbers excluding zero. Analogously, $\mathbb{R}$ and $\mathbb{R}^+$ denote the set of real numbers and positive real numbers excluding zero respectively. $\mathbb{R}_0^+$ denotes the set of positive real numbers including zero. A closed interval of these sets is denoted by $[a, b]$ where $a, b \in \mathbb{N}$ (or $\mathbb{N}^+, \mathbb{R}, \mathbb{R}^+, \mathbb{R}_0^+$ respectively). For some value $x \in [a, b]$ holds that $a \leqslant x \leqslant b$. Analogously, the open interval (a, b) is defined. Here, $x \in (a, b)$ denotes that $a < x < b$. Both types of intervals can also be combined; e. g., $x \in [a, b)$ denotes that $a \leqslant x < b$. Boolean algebra operations utilize $\wedge$ to denote the conjunction of two statements and $\vee$ for their disjunction. An implication is denoted by $\Rightarrow$ and the equivalence of statements is denoted by $\Leftrightarrow$.

Notation 3.1 (Set). A *set* $S = \{s_1, s_2, \dots, s_n\}$ is a collection of distinct objects with $|S|$ denoting the cardinality (size) of the set. A set is finite if its cardinality is bounded. We refer to the empty set by $\emptyset$. For two sets A, B, $=$ denotes their equivalence ($A = B \Rightarrow \forall x[x \in A \leftrightarrow x \in B]$), $\subseteq$ denotes their inclusion ($A \subseteq B \Rightarrow \forall x[x \in A \rightarrow x \in B]$), $\cup$ denotes their union ($A \cup B = \{x : x \in A \vee x \in B\}$), $\cap$ denotes their intersection ($A \cap B = \{x : x \in A \wedge x \in B\}$), $\times$ denotes the Cartesian product over both sets ($A \times B = \{(a, b) | a \in A \wedge b \in B\}$), and $\setminus$ denotes the set-theoretic difference of both sets ($B \setminus A = \{x \in B | x \notin A\}$). A pair of sets A, B is disjoint, if and only if $A \cup B = \emptyset$. ◀

Notation 3.2 (Power Set). The set of all subsets including $\emptyset$ of a set A is called the *power set* of A and is denoted by 2^A or $\mathfrak{P}(A)$. A set A' is a subset of a set A if and only if all objects contained in A' are also contained in A, i. e., $A' \subseteq A$. ◀

Notation 3.3 (Relation). An *n-ary relation* $R \subseteq S_1, S_2, \dots, S_n$ ($n \in \mathbb{N}^+$) denotes a set of n-tuples such that each element consists of n components and the kth component ($k \in \mathbb{N}$, $k \in [1, n]$) is an element of S_k. For *binary relations*, $n = 2$ holds. ◀

Let R be a binary relation on a set X. Then, R^+ denotes the *transitive closure* of R that is a transitive relation on set X such that R^+ contains R and R^+ is minimal; R^+ can be seen as the intersection of all transitive relations containing R. A binary relation R is transitive if and only if $(x, y) \in R \wedge (y, z) \in R \Rightarrow (x, z) \in R$. A binary relation $R = f \subseteq (S_1 \times S_2)$ is a function that maps elements of S_1 to elements of S_2.

Notation 3.4 (Function). A *function* f with domain S_1 and codomain S_2 is commonly denoted by $f : S_1 \rightarrow S_2$. Thereby, (i) $\forall x \in S_1 \exists y \in S_2 : (x, y) \in f$ holds and (ii) $\forall x \in S_1 \wedge \forall y_1, y_2 \in S_2 : (x, y_1) \in f \wedge (x, y_2) \in f \Rightarrow y_1 = y_2$ holds. We write $f(x) = y$ if $(x, y) \in f$. We refer to a *partial function* $f : S_1 \nrightarrow S_2$ with domain S_1 and codomain S_2 if $f : S_1' \rightarrow S_2$ with $S_1' \subset S_1$ is a function. ◀

Notation 3.5 (Sequence). A *sequence* is a function $\sigma\{1, 2, \dots, n\} \rightarrow S$ over a finite set S with $n \in \mathbb{N}$ also referred to as $\sigma : < s_1, s_2, \dots, s_n >$. The length of a sequence is denoted by $|\sigma| = n$. $<>$ denotes the empty sequence with $| <> | = 0$. ◀

In this thesis, sets are generally represented by uppercase letters and by calligraphic letters; number sets are represented by blackboard bold letters. Sequences are mainly represented by uppercase letters. Single entities, usually being part of a set or sequence, are represented by lowercase letters. Relations are generally represented by Fraktur letters. Functions are generally represented by Greek letters and, due to the lack of sufficient letters, functions are exceptionally also represented by Latin letters.

Generally, we use subscripts to denote the relationships between these notions and omit subscripts where the context is clear. For instance, let $a = (M, \mathfrak{X}, T_S, \alpha)$ be a tuple consisting of set M, relation $\mathfrak{X}$, sequence T_S, and function α. Then, M_a,$\mathfrak{X}_a$,$T_{S,a}$, and α_a denote the relationships of M,$\mathfrak{X}$,T_S, and α respectively to a.

3.1 BUSINESS PROCESS MODELS

As mentioned in the previous chapter, process modeling follows either the activity-centric or the object-centric paradigm in practice as well as science. Following the goal of integrating data and further concepts of the object-centric process modeling paradigm into traditional workflow modeling with control flow being the main driver, we focus on the utilization of the activity-centric process modeling paradigm in this thesis. Therefore, we introduce the syntax and semantics in an activity-centric fashion. Accordingly, the syntax of a basic activity-centric process model (ACP) is formally defined as follows with *basic* referring to the set of concepts commonly used for process modeling [171, 389].

Definition 3.1 (Basic Activity-centric Process Model)**.**
A *basic activity-centric process model* $pm^B = (N, D, Q, R, \mathfrak{C}, \mathfrak{F}, type_a, type_t, type_g, \mu, \beta)$ consists of a finite non-empty set $N \subseteq A \cup G \cup E$ of control flow nodes being activities A, gateways G, and event models E (A, G, and E are pairwise disjoint), a finite non-empty set D of data nodes, a finite non-empty set Q of activity labels, and a finite set R of resources actually executing the activities (N, D, Q, and R are pairwise disjoint). $\mathfrak{C} \subseteq N \times N$ is the control flow relation defining a partial ordering of control flow nodes and $\mathfrak{F} \subseteq (D \times A) \cup (A \times D)$ is the data flow relation representing read and write operations of activities with respect to data nodes. function $type_a : A \rightarrow \{task, subprocess, multiInstanceTask, multiInstanceSubprocess\}$ gives each activity a type, partial function $type_t : A \nrightarrow \{user, service, send, receive, unspecified\}$ specifies the type of each (multi-instance) task, function $type_g : G \rightarrow \{XOR, AND\}$ assigns to each gateway a type, function $\mu : A \rightarrow Q$ assigns to each activity a label, and function $\beta : A \rightarrow R$ assigns to each activity a resource, which executes the corresponding activity. We refer to a basic activity-centric process model as process model if the context is clear. ◀

Following the mentioned schema, we use subscripts, e.g., A_{pm}, D_{pm}, and $\mathfrak{F}_{pm}$, to denote the relation of sets, relations, and functions to process model pm and omit subscripts where the context is clear. Tasks and multi-instance tasks can be executed manually (*user task*) or automatically (*service task*), they may send respectively receive messages, or their execution type is unspecified usually referring to a mixture of manual and automatic execution. Both types of subprocesses are own process model definitions, i. e., a subprocess contains a set of control flow and data nodes as well as the corresponding relations and

functions as specified for an activity-centric process model. Further, an activity is executed by a human *resource*. In case of service tasks, the human resource does not actively execute the activity but acts as responsible person to monitor the automatic execution. The specific human resource is assigned to an activity either directly by name calling or indirectly by specifying a set of requirements, where each human fulfilling these requirements may allocate the activity. Therefor, multiple allocation procedures exist, e. g., role-based, capability-based, and history-based resource assignment [290], where allocation depends on the role a human resource has in the organization the business process belongs to, the capabilities and knowledge a human resource possesses, or the tasks the resource worked on earlier respectively.

Each activity requires to have a *label* specifying the work to be done during activity execution on an abstract level. The corresponding activity description details this specification. The structure of such activity label is defined as follows.

Definition 3.2 (Activity Label)**.**
An *activity label* is an ordered list of words represented by a string describing an action, a data object an action is performed upon, and an optional fragment providing further details (e.g., locations, resources, or regulations) [204]. ◀

Each activity label follows a predefined natural language grammar as follows.

Definition 3.3 (Natural Language Grammar for Process Modeling)**.**
A context-free natural language used for activity labels in process models consists of terminal symbols representing verbs of any form (VB), singular nouns (NN), plural nouns (NNS), adjectives (JJ), adverbs (RB), determiner (DT), coordinating conjunctions (CC), prepositions and subordinating conjunctions (IN), and the word *to* (TO). A set of ten axioms can be used to build the natural language NLL of an activity label with NLL, NP, and PP being non-terminal symbols.

1. NLL → VB NP | NP NP | NP VB NP
2. NP → NN | NNS
3. NP → JJ NP | JJ NNS
4. NP → NN RB | NNS RB
5. NP → DT NP | DT NNS
6. NP → NP NP
7. NP → NP CC NP
8. NP → NP PP
9. PP → IN NP
10. PP → TO NP

◀

With regards to [183], the first axiom allows three different labeling styles for an activity label: verb-object labeling (e. g., analyze order), action-noun labeling (e. g., order analysis), and descriptive labeling (e. g., computer retailer analyzes order). Practically, there might also exist activity labels not conforming to the grammar introduced above, e. g., a single noun as in *analysis*. In the scope of this thesis, we assume that all activity labels do conform to the introduced grammar.

For a control flow node $x \in N$, $\bullet x \subseteq N$ denotes the set of preceding control flow nodes of x, $x\bullet \subseteq N$ denotes the set of succeeding nodes of x, and $\bullet x \bullet \subseteq N$ denotes the set of connected, i. e., preceding and succeeding, nodes of x. $|\bullet x|$ returns the number of preceding control flow nodes of x; analogously, the number of the other sets can be determined. Then, node $x \in N$ is a *source* node, if x has exactly one preceding and no succeeding node ($|\bullet x| = 0 \wedge |x \bullet| = 1$), or a *sink* node, if x has exactly one preceding and no succeeding node ($|\bullet x| = 1 \wedge |x \bullet| = 0$).

Structural correctness criteria

Further, we expect each process model pm^B to satisfy a set of basic structural correctness criteria (SCC):

(SCC-1) each activity of pm has exactly one preceding and exactly one succeeding control flow node, i. e., $\forall\, a \in A : |\bullet a| = 1 \wedge |a\bullet| = 1$,

(SCC-2) each gateway of pm has at least three connected control flow nodes with at least one preceding and at least one succeeding node, i. e., $\forall\, g \in G : |\bullet g \bullet| \geqslant 3 \wedge |\bullet g| \geqslant 1 \wedge |g\bullet| \geqslant 1$,

(SCC-3) each gateway of pm acting as split ($type_g = XOR$) or fork ($type_g = AND$) denoted by the set G_i has exactly one preceding control flow node, i. e., $\forall\, g \in G_i \subseteq G\, |\bullet g| = 1$,

(SCC-4) each gateway of pm acting as join ($type_g = XOR$) or merge ($type_g = AND$) denoted by the set G_o has exactly one succeeding control flow node, i. e., $\forall\, g \in G_o \subseteq G : |g\bullet| = 1$,

(SCC-5) each event model of pm is either a source node or a sink node, and

(SCC-6) process model pm is structurally sound, i. e., pm has exactly one *source* and one *sink* node and every node of pm is on a path from the source to the sink node.

Visualization

For visualization of process models, we utilize Business Process Model and Notation (BPMN) [243]. Therefore, event models are drawn as untyped start and end events. An activity is visualized as a rectangle with rounded corners and its label inside. A collapsed, i. e., hidden, subprocess is indicated by a marker shaped like "+" within a square at the bottom of the activity while an expanded subprocess is visualized as activity with other nodes inside instead of a label. The task type is indicated by markers shaped like a human head (manual), two gearwheels (service), a black envelope (send), or a black envelope (receive) in the upper left corner of the activity. Unspecified tasks do not contain any marker. Gateways are drawn as diamonds. To differentiate the above introduced types of gateways, we use a marker shaped like "×" for an *XOR* gateway and a marker shaped like "+" for an *AND*

gateway inside the diamond shape. A data node in a particular data state is visualized as a BPMN data object having its name followed by its state in square brackets. A data node $d \in D_{pm}$ can appear multiple times in the visualization of the process model (also when in a particular data state). Control flow and data flow edges are drawn as solid and dashed directed edges, respectively. Although using BPMN for visualizing process models in this thesis, the concepts explained can be generally applied to all business processes complying to the introduced formalization.

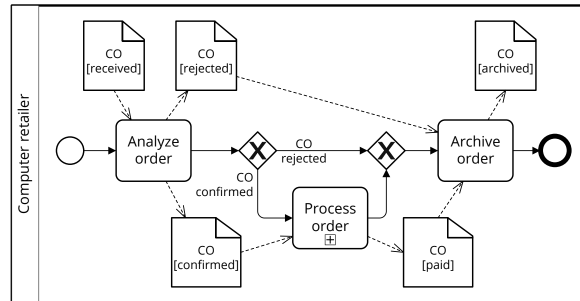


Figure 17: Process model example.

Figure 17 shows a basic activity-centric process model showing an extract of the build-to-order and delivery process from Section 2.4. This extract shows three untyped activities (rectangle shapes) connected by control flow edges (solid edges) with the second one being only executed if the customer order *CO* gets confirmed by the first activity *Analyze order*. This choice is represented by the diamond-shaped XOR gateways. The information about decision taking is often not modeled explicitly in an activity-centric process model. For clarification, we added this information in reference to data nodes (document shapes). In this figure, five data nodes are annotated to the activities by data flow edges (dashed edges) where the direction indicates read or write operations. If a control flow edge targets a data node, it is written from the source activity. If a control flow edge origins from a data node, it is read by the targeted activity. The start of the process visualized by a start event (circle with light border) and the termination is visualized by an end event (circle with bold border). Finally, the rectangle comprising the remaining process model indicates the resource assigned to the activities. Here, all activities are executed by resource *Computer retailer*.

Process execution semantics

The execution semantics of a process model follows Petri net semantics [233]. A corresponding Petri net [253] can be derived from a process model by the mapping introduced by Dijkman et al. [80]. Termination of a node triggers the control flow to move to the next node on the appropriate paths through the process models. Presenting the process progress with respect to control flow on process model level instead of the Petri net level can be done by utilizing the process state or marking of a process model, defined as follows. Thereby, the term marking

refers to the assignment of tokens to control flow edges and the semantics is presented as token game.

Definition 3.4 (Process State and Marking)**.**
A *process state* (or *marking*) m of a process model describes the currently enabled control flow nodes during process execution. A process state is represented by tokens on its control flow edges. Given process model pm, the process state of pm is a mapping $m : \mathfrak{C}_{pm} \to \mathbb{N}$ for each control flow edge of pm. ◀

M_{pm} denotes the set of all markings of a process model pm, where each marking $m_{pm} = [m(c_1), m(c_2), \ldots, m(c_n)]\ \forall c \in \mathfrak{C}_{pm}$. If the control flow edges are totally ordered, a marking m can be visualized by an array. Let the control flow edges be totally ordered by their identifier visualized as annotation the control flow edges, then the marking of the process model given in Figure 18 after executing activity *Analyze order* and before taking the decision is $m = [0, 1, 0, 0, 0, 0, 0]$. Thereby, the number in the array indicates the number of tokens on a control flow edge. In the example, only edge 2 contains a token.

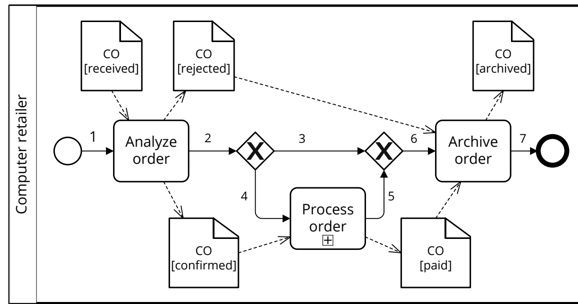


Figure 18: Process model example from Figure 17 with annotated control flow edges showing an edge's index. Based on these indices, the marking of the given process model is $m = [0, 1, 0, 0, 0, 0, 0]$ after execution of activity *Analyze order*.

Formal markings

Let m and m' be two markings of process model pm and let $\mathbb{N}$ denote the set of natural numbers including zero. We write $m \xrightarrow{x} m'$ to denote that the process state changes from m to m' by executing node x of pm. If $\sigma_A = a_1 a_2 \ldots a_n$, $n \in \mathbb{N}$, is a list of nodes of pm, $m \xrightarrow{\sigma_A} m'$ denotes the fact that there exists a sequence of process states $m_1, m_2, \ldots, m_{n-1}$ such that $m \xrightarrow{a_1} m_1 \xrightarrow{a_2} \ldots m_{n-1} \xrightarrow{a_n} m'$. We call σ_A an *execution sequence* of pm, which starts with m. $|\sigma_A|$ denotes the number of node executions involved in the execution sequence. Let a and a' be nodes of pm. With $a \Rightarrow_{pm} a'$, we denote the predicate, which evaluates to `true` if $a = a'$ or if there exists an execution sequence of pm, which starts with the initial marking and executes a before a'; otherwise $a \Rightarrow_{pm} a'$ evaluates to `false`.

Based on execution sequences of markings, we define a path through a process model. Thereby, we refer to the process state, where the

control flow edge originating from the source node is marked while no other control flow edge is marked, as *initial marking* of a process model. Further, we refer to the process state, where all control flow edges targeting a sink node are marked while no other control flow edge is marked, as *final marking* of a process model.

Definition 3.5 (Path (Trace))**.**
A *path* (or *trace*) through a process model is an execution sequence σ_A of control flow nodes starting from the initial marking leading to one final marking. ◀

Process instance

Execution of business processes leads to the process instance view, because these executions are represented by process instances with each instance belonging to exactly one process model. At all points in time, a process instance has a current *process instance state* which is represented by a marking of the corresponding process model. A sequence of process instance states describes a process instance which we define as follows.

Definition 3.6 (Process Instance)**.**
A *process instance* $i = (pid, T_Z, pm)$ consists of a process instance identifier pid and a sequence $T_Z = \langle z_1, z_2, \ldots, z_n \rangle$ of process instance states and references a process model pm. Each $z_k \in T_Z$ refers to one marking of process model pm. ◀

Given a set of process models PM representing one business process, I denotes the set of process instances of all process models $pm \in PM$. Auxiliary function $\rho_I : I \to PM$ assigns each process instance $i \in I$ to its corresponding process model $pm \in PM$. All process instances referring to the same process model pm are comprised in set $I'_{pm} \subseteq I$.

Object-centric processes

After discussing activity-centric process models above, we briefly introduce the second major modeling paradigm next starting with the corresponding definition, which presents one option to formalize object-centric process models (OCPs). The definition follows the proposal from Yongchareon et al. [384]. However, an extension to the Guard-Stage-Milestone (GSM) approach [141] as presented in the case management standard Case Management Model and Notation (CMMN) [245] can also be used for representation; both utilize the same concepts but present them differently. While CMMN presents an OCP as a graph, Yongchareon et al. utilize a rule-based approach visualized with tables. In this thesis, we utilize the table representation for OCPs, since it visualizes the dependencies between tasks on a more detailed level; task enablement can be based on states and further attributes of data objects. The rule-based definition is as follows.

Definition 3.7 (Object-centric Process Model)**.**
Let S be a set of data states and let $\mathcal{J}$ be a set of data attributes. Then, an *object-centric process model* $ocp = (AS, U, BR)$ consists of a schema

$AS = (C, \mathtt{instate}, \mathtt{defined})$ with a finite non-empty set of data classes C, the *in-state* function $\mathtt{instate} : C \times S \rightarrow \{true, false\}$, and the *defined* function $\mathtt{defined} : C \times \mathcal{J} \rightarrow \{true, false\}$, a finite set U of tasks, and a finite set BR of business rules. A data class consists of a name, a set of attributes $\mathcal{J}$, and a set of data states S. Let a data object be the instance representation of a data class, then functions $\mathtt{instate}$ and $\mathtt{defined}$ evaluate to true or false depending on the existence of a data object of class $c \in C$ being in a data state $s \in S$ ($\mathtt{instate}$) or containing a value for an attribute $j \in \mathcal{J}$ ($\mathtt{defined}$) respectively. A task $u = (label, CS)$, $u \in U$, consists of a label and a finite set $CS \subseteq C$ of data classes referring to data objects being read or written by this task. A business rule $br = (pre, post, SU)$ consists of a precondition pre, a postcondition $post$, and a finite set $SU \subseteq U$ of tasks manipulating data objects to meet the postcondition. A pre- respectively a postcondition comprises a set of in-state and defined functions connected by operators $\wedge$ and $\vee$. Thereby, a defined function may only contain a data class, which is used in at least one in-state function. ◂

Table 1 shows the ACP from Figure 17 as OCP visualized following the rule-based approach [384]. First, the data classes, tasks, and business rules are specified. The example utilizes only data class *customer order CO*. Objects of that class may get manipulated by the three tasks *analyze*, *processOrder*, and *archive*. The interrelation of tasks and data classes is specified by business rules *br1*, *br2*, and *br3*. Business rule *br1* is correlated to task *analyze* and is triggered if and only if the corresponding customer order *CO* is in data state *received* while, additionally, attribute

Table 1: Process model from Figure 17 as object-centric process model following visualization of [384].

Data classes:	customer order CO
Set of tasks:	analyze; processOrder; archive
Business rules:	b1;b2;b3
b1: Computer retailer analyzes customer order with respect to completeness and validity	
Precondition:	$\mathtt{instate}(CO, received) \wedge \mathtt{defined}(CO, CustomerNumber)$
Tasks:	analyze(CO)
Postcondition:	$(\mathtt{instate}(CO, confirmed) \vee \mathtt{instate}(CO, rejected)) \wedge \mathtt{defined}(CO, AnalysisDescription)$
b2: Computer retailer processes order	
Precondition:	$\mathtt{instate}(CO, confirmed)$
Tasks:	processOrder(CO)
Postcondition:	$\mathtt{instate}(CO, paid) \wedge \mathtt{defined}(CO, Price) \wedge \mathtt{defined}(CO, DeliveryDate) \wedge \mathtt{defined}(CO, PaymentDate)$
b3: Computer retailer archives order in information system	
Precondition:	$(\mathtt{instate}(CO, paid) \wedge \mathtt{defined}(CO, PaymentDate)) \vee \mathtt{instate}(CO, rejected)$
Tasks:	archive(CO)
Postcondition:	$\mathtt{instate}(CO, archived) \wedge \mathtt{defined}(CO, ArchivalDate)$

CustomerNumber of the *CO* is specified. This task is working on objects of class *CO* as indicated through the class in parentheses following the task name. Executing task *analyze* moves the object either into state *confirmed* or into state *rejected* while the attribute *AnalysisDescription* must get specified during task execution. On confirmation of the object, business rule *br2* gets triggered and moves the object into data state *paid* with three attributes getting specified (*Price*, *DeliveryDate*, and *PaymentDate*). Business rule *br3* is triggered if the customer was processed and attribute *PaymentDate* got specified by *br2* or if the customer order was rejected by *br1*. The execution of the corresponding task *archive* sets the customer order *CO* into data state *archived* and specifies attribute *ArchivalDate*.

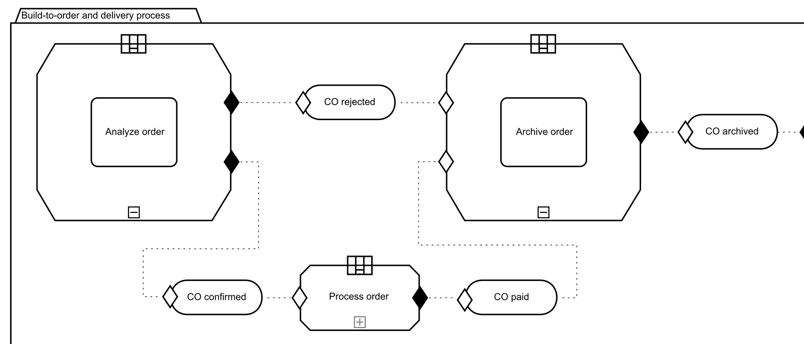


Figure 19: Process model example from Figure 17 in CMMN [245].

Following CMMN [245], this example can be represented as shown in Figure 19. We utilized three stages from which two are expanded and contain the tasks *Analyze order* and *Archive order* respectively. The third task is hidden in the stage labeled *Process order*. The stages and tasks refer to the tasks in the OCP definition. The white diamonds indicate the preconditions upon which a stage is enabled while the black diamonds indicate results (postconditions). Configuring them, also attributes of data objects could be considered but they are not supposed to be visualized. The stages are connected via milestones (ellipses) that visualize the postconditions of the preceding stage. For instance, the upper output connector (refers to a control flow edge) of the first stage connects to milestone *CO rejected* indicating that the customer must have been rejected to follow this path. The successor of this milestone is then targeted to such customer orders such that they can get archived directly. The black diamond on the overall container – called *case plan model* – terminates the process.

The execution semantics of OCPs is represented by means of object life cycles (OLCs) which are state transition nets describing the partial ordering of data states and specifying the tasks inducing this order. In OLC terminology, we refer to tasks as *actions*. In detail, an OLC is defined as follows.

Definition 3.8 (Object Life Cycle)**.**
An *object life cycle* $l = (S, s_i, S_F, \mathfrak{T}_S, \Sigma, c)$ consists of a finite non-empty set S of data states, an initial data state $s_i \in S$, a non-empty set $S_F \subseteq S$ of final data states, and a finite set Σ of actions representing the manipulations on data objects (S and Σ are disjoint). $\mathfrak{T}_S \subseteq S \times \Sigma \times (S \backslash \{s_i\})$ is the data state transition relation through which an object life cycle describes the dependencies between the data states of a data class. ◀

L denotes the finite set of all object life cycles in a specified scope. A scope usually comprises a business process but may also be extended towards multiple business processes or reduced towards single process models. If the scope is not explicitly defined, we assume that it is set to the business process. Auxiliary function $\eta : L \to C$ implements the data class reference by mapping a given object life cycle $l \in L$ to the corresponding data class $c \in C$ such that $\eta(l) = c$. As aforementioned, we use subscripts, e. g., S_l and $\mathfrak{T}_{S,l}$ to denote the relation of sets and functions to the object life cycle l and omit subscripts where the context is clear.

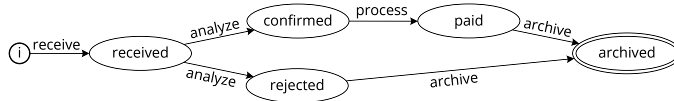


Figure 20: Object life cycle example.

Figure 20 presents the object life cycle of data class customer order *CO* utilized in the process model in Figure 17. Within this part of the overal build-to-order and delivery process, a *CO* may be in data states *received*, *confirmed*, *paid*, *archived*, or *rejected* – in one state at a time. Additionally, we add an initial state *i* leading to state *received* via state transition *receive* that is not covered in Figure 17. The remaining data state transitions correspond to the activities shown in the process model in Figure 17; e. g., state transition process *process* changes the state of a *CO* object from *confirmed* to *paid*. The final data state of this OLC is data state *archived*.

The semantics for OLCs follows and extends state machine semantics from [127] as follows. Let $\mathbb{N}$ denote the set of natural numbers including zero. For $s, s' \in S$ and $a \in \Sigma$, we denote by $s \xrightarrow{a} s'$ the fact that $(s, a, s') \in \mathfrak{T}_S$. If $\sigma_S = a_1 a_2 \dots a_n$, $n \in \mathbb{N}$, is a sequence of actions, $s \xrightarrow{\sigma_S} s'$ denotes the fact that there exists a sequence of data states $s_1 s_2 \dots s_{n-1}$ such that $s \xrightarrow{a_1} s_1 \xrightarrow{a_2} \dots s_{n-1} \xrightarrow{a_n} s'$. We call σ_S an *execution sequence* of l, which starts with s, and s' *is a reachable data state from data state* s *via* σ_S. $|\sigma_S|$ denotes the number of action instances involved in the execution sequence. With $s \Rightarrow_l s'$, we denote the predicate, which evaluates to `true` if $s = s'$ or if there exists an execution sequence of l, which starts with i and reaches s before s'; otherwise $s \Rightarrow_l s'$ evaluates to `false`. We require each data state $s \in S \backslash (s_i \cup S_F)$ being part of execution sequences $i \xrightarrow{\sigma_S} s$ and $s \xrightarrow{\sigma_S} s'$ with $s' \in S_F$, i. e.,

OLC semantics

each data state is part of an execution sequence $i \xrightarrow{\sigma_S} s'$ leading from the initial to some final data state. Further, we require each data state $s' \in S_F$ is reachable from $s_i \in S$.

3.2 BUSINESS PROCESS RELATIONS

Relations describing the order of nodes are of major importance. Usually, they are defined on activity level such that event models and gateways are excluded from these relations. As the relations between activities are sufficient for this thesis, we follow this approach and define the relations accordingly. Two activities $a_1, a_2 \in A$ are in *weak order relation* [95, 368, 369] $a_1 \succ a_2$ if there exists one path in the process model, where a_1 is succeeded by a_2 with any number of other control flow nodes between them. If the weak order relation between activities a_1 and a_2 holds in one direction for all paths through the process model where both activities occur, then both activities are in *strict order relation* $\rightsquigarrow$, i. e., $a_1 \rightsquigarrow a_2$ implies that there does not exist a path which executes a_2 before a_1. If the weak order relation between activities a_1 and a_2 does not hold in any direction in all paths trough the process model, both activities are in an *exclusiveness relation* $+$ to each other; i. e., there exists no path executing both activities in any order. If the weak order relation for activities a_1 and a_2 holds in both directions in one or among multiple paths through the process model, both activities are in a *concurrency relation* $\|$. Definition 3.9 summarizes above introduced relations.

Definition 3.9 (Control Flow Relations)**.**
Let $a_1, a_2 \in A$ be two activities of process model pm and let $\mathfrak{C}^+$ denote the *transitive closure over the control flow relation* of process model pm such that $n_1 \mathfrak{C}^+_{pm} n_2$ respectively $(n_1, n_2) \in \mathfrak{C}^+_{pm}$ indicate that there exists a path from n_1 to n_2 in pm for $n_1, n_2 \in N_{pm}$. Further, let $\succ$ denote the weak order relation, let $\rightsquigarrow$ denote the strict order relation, let $+$ denote the exclusiveness relation, and let $\|$ denote the interleaving relation between two activities a_1 and a_2. Then,

- $a_1 \succ a_2$ holds if $(a_1, a_2) \in \mathfrak{C}^+$,
- $a_1 \rightsquigarrow a_2$ holds if $a_1 \succ a_2 \wedge a_2 \not\succ a_1$,
- $a_1 + a_2$ holds if $a_1 \not\succ a_2 \wedge a_2 \not\succ a_1$, and
- $a_1 \| a_2$ holds if $a_1 \succ a_2 \wedge a_2 \succ a_1$.

◀

Considering the process model in Figure 17, activity *Analyze oder* is in *strict order* relation to both other activities and activity *Process order* is in *strict order* relation to activity *Archive order*. Exclusiveness and interleaving relations are not part of this process model. Considering the more detailed process model in Figure 5 on page 29, activities *Process order* and *Send rejection* are in *exclusiveness* relation since in one process execution only one of them can be executed. In the process model in

Figure 10 on page 32, each activity of the lower path is in *interleaving* relation with activity *Book purchase order internally*. Finally, considering the process model in Figure 8 on page 31, activities *Cancel remaining quotes* and *Decide quote* are in *weak order* relation but not in *strict order* relation since in case some number of quotes get rejected, the second activity would have been executed before the first one (although in another loop iteration) while in the optimal case (first quote gets accepted and none rejected), the first activity is executed before the second one.

The concept of *behavioral profiles* [368] utilizes the weak order relations between all activities of a process model to identify the sets of strict order, exclusiveness, and concurrency relations. The set of these three relations for one process model pm is the behavioral profile of pm.

3.3 NET SYSTEMS

Process models following one of the process description languages introduced in the last chapter are widely used in industry. However, these lack formal semantics and analysis techniques for checking, for instance, behavioral consistency. Therefore, we utilize a well-established formalism for the analysis: Petri nets. Most existing process description languages can be transformed into Petri nets dealing with the tradeoff of information loss for complex structures as the non-interrupting intermediate events in BPMN. So can process models following the definitions given in this thesis be transformed into Petri nets (cf. Sections 3.1 and 4.7). This section is dedicated to the formal definition of net systems. First, we define the syntax and semantics of net systems before we elaborate on their structural classes and discuss their behavioral properties which finally are used for behavioral analysis.

Petri nets are graphs that represent the behavior of dynamic systems. They are based on ideas of Carl Adam Petri he presented in his seminal doctoral thesis [253] 1962, a rather philosophical discussion about concurrency and synchronization as theory for the asynchronous communication of systems. Thereby, he proposed a behavioral formalism extending automata concepts by concurrency and introduced the basic notions of Petri nets: places and transitions (see below) describing local states and local actions respectively. Since then, Petri nets gained a large uptake in the computer science community and a large body of literature shows influential properties and analysis techniques, e. g., [233]. Comprehensive introductions to Petri nets can be found in [74, 122, 274, 275, 276]. Next, we summarize these introductions and formally define net systems.

Syntax and Semantics

A Petri net, or *net* for short, is a bipartite graph of places and transitions which are connected by directed edges called flow.

Definition 3.10 (Petri Net)**.**
A *Petri net* is a tuple $pn = (P, \mathcal{T}, \mathfrak{E})$ consisting of finite disjoint sets P and $\mathcal{T}$ representing places and transitions respectively and a flow relation $\mathfrak{E} \subseteq (P \times \mathcal{T}) \cup (\mathcal{T} \times P)$. ◀

We use subscripts, e.g., P_{pn} and $\mathcal{T}_{pn}$ to denote the relation of the sets to Petri net pn and omit subscripts where the context is clear. We refer to the set $P \cup \mathcal{T}$ as nodes of the Petri net. The set of all input nodes of a node $x \in P \cup \mathcal{T}$ is denoted as *preset* $\bullet x = \{y \in pn \cup \mathcal{T} | \mathfrak{E}(y, x) = 1\}$. Analogously, the set of all output nodes of a node $x \in P \cup \mathcal{T}$ is denoted as *postset* $x\bullet = \{y \in P \cup \mathcal{T} | \mathfrak{E}(x, y) = 1\}$. A node $x \in P \cup \mathcal{T}$ is an input (output) node of a node $y \in P \cup \mathcal{T}$ if and only if $x \in \bullet y$ ($x \in y\bullet$). A place $p \in P$ is a source place referred to as p_i if $\bullet p = \emptyset$ and a sink place referred to as p_o if $p\bullet = \emptyset$. $\mathfrak{E}^+$ denotes the irreflexive transitive closure of the flow relation.

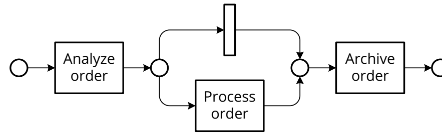


Figure 21: Petri net example presenting the same behavior as the process model in Figure 17.

Figure 21 shows a Petri net with the same behavior as the process model introduced in Figure 17. First, the order is analyzed, second, the order is optionally processed, and finally, the order is archived. Rectangles represent transitions and circles represent places that can get marked with tokens. Transitions usually contain a label indicating the action performed by them. Additionally, for syntactical reasons, *no operation (nop)* transitions do exist. Thy are unlabeled and have a smaller size then labeled transitions. Usually, no operation transitions are used to keep the bipartiteness of a Petri net. For instance, in Figure 21, a no operation transition is required to cover the optionality of transition *Process order* ensuring a bipartite graph.

The state of a Petri net is represented by its marking, i. e., distribution of tokens across its places. Therefore, based on the syntax, we define Petri net semantics in terms of markings.

Definition 3.11 (Petri Net State, Marking, and System)**.**
Let $pn = (P, \mathcal{T}, \mathfrak{E})$ be a Petri net. The *marking* (or *state*) f of pn is a function $f : P \to \mathbb{N}$ mapping the set of places onto the natural numbers $\mathbb{N}$ including zero such that $f(p)$ returns the number of tokens of place p. Let $P_i \cup P_o \subseteq P$ be the source and sink places of a Petri net such that $\forall p_i \in P_i : \bullet p_i = \emptyset \wedge |p_i \bullet| \geqslant 1$ and $\forall p_o \in P_o : p_o\bullet = \emptyset \wedge |\bullet p_o| \geqslant 1$. Then, f_i is the initial marking with tokens in each source place p_i and no token in any other place of the net. f_o is the final marking with tokens in each sink place p_o and no token in any other place of the net.

Let F denote the set of all markings of a Petri net and let $f, f' \in F$ be two markings. Then $f \geqslant f'$ if and only if $\forall p \in P : f(p) \geqslant f'(p)$ and $f > f'$ if and only if $\forall p \in P : f(p) > f'(p)$. $f + f'$ denotes a marking f'' such that $\forall p \in P : f''(p) = f(p) + f'(p)$. A *net system* is a tuple $\mathcal{S} = (pn, f_i)$, where pn is a Petri net and f_i is its initial marking. ◀

F_{pn} denotes the set of all markings of a Petri net pn, where each marking $f_{pn} = [f(p_1), f(p_2), \ldots, f(p_n) \; \forall p \in P_{pn}]$. If the places are totally ordered, a marking m can be visualized by an array. Let the places be totally ordered by their identifier, then the marking of the Petri net given in Figure 22 after execution (firing, see below) of transition *Analyze order* is $f = [0, 1, 0, 0]$, because $f(p_1) = f(p_3) = f(p_4) = 0$ and $f(p_2) = 1$ where the natural number indicates the number of tokens in the corresponding place.

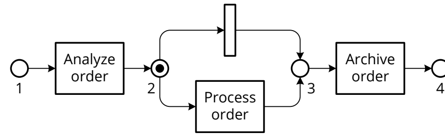


Figure 22: State of Petri net from Figure 21 after firing of transition *Analyze order*. Assuming a total ordering of places (from left to right in this example), the marking f of this Petri net can be represented as array such that $f = [0, 1, 0, 0]$.

Tokens are visualized as black dots within places in the graphical representation and represent execution conditions for the transitions. If all input places of a transition carry one token, the transition is enabled. Upon firing, it consumes the tokens and produces one in each of its output places, i. e., the state of the Petri net is changed.

Definition 3.12 (Enabling, Firing, Reachability)**.**
Let $pn = (P, \mathcal{T}, \mathfrak{E})$ be a Petri net and $f \in F$ a marking. A transition $t \in \mathcal{T}$ is *enabled* in f, if and only if $\forall p \in \bullet t : [f(p) \geqslant 1$. If $t \in \mathcal{T}$ is enabled in f, then it can fire. *Firing* of t leads to a new marking f' such that $\forall p \in \bullet t : f'(p) = f(p) - 1$ and $\forall p \in t\bullet : f'(p) = f(p) + 1$. An execution sequence of transitions $\sigma_{\mathcal{T}} = t_1, t_2, \ldots, t_{n-1}$ is a *firing sequence* of pn denoted by $f_1 \xrightarrow{\sigma_{\mathcal{T}}} f_n$, if and only if there exist markings $f_1, f_2, \ldots, f_n \in F$ such that for all $1 \leqslant i < n$ it holds $f_i \xrightarrow{t_i} f_{i+1}$. For any two markings $f, f' \in F$, f' is *reachable* from f, if and only if $f \xrightarrow{\sigma_{\mathcal{T}}} f'$. ◀

As a marking of a net system represents it state, the set of markings reachable from the initial marking represents the *state space* of a net system. Identifying whether a transition can be executed requires to construct the state space of the corresponding net system and to detect a reachable marking enabling this transition [122]. In the context of this thesis, we assume firing semantics to be sequential referred to as trace semantics [77, 138, 196]. This means, only one transition can fire at a time, i. e., firing a transition is transaction, leading to *firing sequences*. If

multiple transitions are enabled simultaneously, only one of them is arbitrarily chosen to fire. Subsequently, the other still enabled transitions may follow. Thus, firing of concurrent transitions is interleaving. Following trace semantics, the behavior of a net system is defined in terms of all distinct firing sequences starting in the initial marking. Each firing sequence refers to a trace of the net system.

Definition 3.13 (Traces of a Net System)**.**
Let $\mathcal{S} = (pn, f_i)$ be a net system with $pn = (P, \mathcal{T}, \mathfrak{E})$ and $f_i \in F$. The set of traces of a net system is defined by $T_P(pn, f) = \{\sigma_\mathcal{T} \in \mathcal{T}^* | \exists f \in F : f_i \xrightarrow{\sigma_\mathcal{T}} f\}$. ◀

Following this definition, a trace does not necessarily represent a complete firing sequence as the reached marking may not be final and thus, may not be a termination state of the net system.

Structural and Behavioral Properties

Structural properties

Net systems adhere to various structural properties based on which they are classified within structural classes. Each net system may belong to multiple of these classes. The *free-choice* property [27, 74, 321] is an important restriction on the structure of a net as a large theory of behavioral analysis with comparably efficient algorithms is based on this class. In fact, free-choice net systems are usually cited as good compromise between expressiveness and analyzability of net systems [331]. In free-choice net systems, structure and behavior is closely coupled [159] such that each choice within the net can be made freely, i. e., all transitions involved in a choice have the same knowledge about the enablement of their enabling; the sets of input places of transitions involved in a choice are either identical or disjoint. There must be no place that is input to one transition and no input to another one involved in the same choice. Historically, this notion was introduced as *extended free-choice* [27] but *classical* free-choice nets [121] and extended ones are behaviorally equivalent [27].

Workflow nets [331, 332], another important restriction on a net's structure, were explicitly designed for modeling and analyzing business processes. A net is called a workflow net, if and only if it contains a dedicated source place and a dedicated sink place as well as the short-circuited net is strongly connected, i. e., each node lies on a path from the source to the sink place. A short-circuited net is obtained by connecting the sink place with the source place via a new nop transition.

Definition 3.14 (Workflow Net)**.**
A net $pn = (P, \mathcal{T}, \mathfrak{E})$ is a *workflow net*, if and only if pn has a single source place $p_i \in P$ with $\bullet p_i = \emptyset$, pn has a single sink place $p_o \in P$ with $p_o \bullet = \emptyset$, and the short-circuited net $pn' = (P, \mathcal{T} \cup \{t^*\}, \mathfrak{E} \cup \{(p_o, t^*), (t^*, p_i)\})$ of pn is strongly connected. ◀

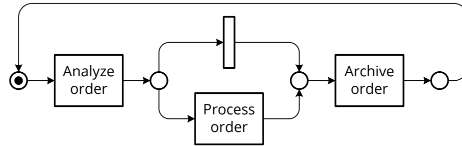


Figure 23: Short-circuited workflow net for Petri net from Figure 21 with marking in the initial place.

A net system $\mathcal{S} = (pn, f_i)$ is called a workflow system if pn is a workflow net. The Petri net given in Figure 21 is a workflow net since it has single source and sink places and the short-circuited net as given Figure 23 is strongly connected, since each node of the net is reachable from each other node.

Behavioral properties

Besides structural properties, net systems may also adhere to behavioral properties being proposed for verification implicating the absence of some behavioral anomaly. In the next paragraphs, we recall some of them being of relevance to this thesis.

Boundedness restricts the behavior of a net system to a set of finite markings. In unbounded net systems, infinitely many tokens are produced by some transitions. This is commonly seen as erroneous behavior [331] as the state space of the corresponding net grows infinitely. Thus, verification of some properties is not decidability any more [122]. Avoiding this, the boundedness property ensures that the upper bound, i. e., the number of tokens that may be carried by one place at any time, is finite – and subsequently the number of reachable markings as well. Secondly, the *safeness* property further restricts boundedness as safeness requires that the analyzed net system's upper bound is 1, i. e., no place carries more than one token at any point in time.

Definition 3.15 (Boundedness and Safeness)**.**
Let $\mathcal{S} = (pn, f_i)$ be a net system with $pn = (P, \mathcal{T}, \mathfrak{E})$ and $f_i \in F$. $\mathcal{S}$ is *bounded*, if and only if there exists for each place $p \in P$ an upper bound $n \in \mathbb{N}^+$ such that $\forall f \in F : f(p) \leqslant n$ with $\mathbb{N}^+$ denoting the set of positive natural numbers excluding zero. $\mathcal{S}$ is *safe*, if and only if $\mathcal{S}$ is bounded and the upper bound of each place $p \in P$ equals 1. ◀

The *soundness* property has been introduced as correctness criterion for business process models [331] and bases on a set of requirements a process specification needs to follow. Initially, soundness was specified for workflow nets [74] but this property can be applied to a wide range of process description languages either by transformation to workflow nets, if net-based formalisms are available [357], or by considering the corresponding execution semantics and checking the behavior according to the requirements recalled below. Further, process models can be transformed into workflow nets for soundness checking as for instance, with the BPMN to Petri net mapping given in [80]. Based on the structural restrictions of workflow nets, soundness checking requires a single source and a single sink node of the corresponding process

model. Then, soundness ensures that (i) the final marking of a net system is always reachable starting from the initial marking, (ii) a token in the sink place implies the absence of tokens in all other places, and (iii) each transition can be enabled in some marking of the net system. (i) infers that there does not exist any deadlock in the net system. A deadlock refers to a marking except the final one, where no transition is enabled (*dead marking*). (ii) infers that a sound workflow net is bounded and that there do not exist lifelocks within the net system. A lifelock describes a situation, where some transitions are trapped in an infinite loop. Combining requirements (i) and (ii) implies *proper termination* of the net system. (iii) infers that there do not exist dead transitions within the net system, i. e., for each transition t, there exists a marking reachable from the initial marking in which t is enabled and can get fired. Based on markings of a net system, we formally define the soundness property as follows.

Definition 3.16 (Soundness)**.**
A workflow system (pn, f_i) with the source place p_i and the sink place p_o is *sound*, if and only if (i) for every marking f reachable from initial marking f_i, there exists a firing sequence leading from f to the final marking f_o, i. e., $\forall f : (f_i \xrightarrow{*} f) \Rightarrow (f \xrightarrow{*} f_o)$, (ii) marking f_o is the only marking reachable from the initial marking with a token in the sink place, i. e., $\forall f : (f_i \xrightarrow{*} f, f \geqslant f_o) \Rightarrow (f = f_o)$, and (iii) every transition of the net can be enabled, i. e., $\forall t \in \mathcal{T} : \exists f, f' : f_i \xrightarrow{*} f \xrightarrow{t} f'$. ◀

Further soundness definitions relaxing above requirements have been introduced for various purposes. For instance, process choreographies (see Section 4.5 for details) are realized by several process orchestrations interacting with each other, i. e., several process models are connected by message flow edges. Such compositions may still show expected behavior without executing all activities specified in the single process orchestrations. In other words, there may exist dead transitions in the net system. To cope with these situations, *weak soundness* [195] was introduced. In contrast to the soundness property from above, single activities may not be executed in the course of process execution, but the process still terminates properly (no deadlocks and no tokens left in the Petri net after reaching the final marking; cf. (i) and (ii) from Definition 3.16). In this thesis, we apply weak soundness in cases, where enablement of transitions does not base on the process model definition only but also on context data as the current state of data objects utilized during process execution. Details follow in Chapters 4 and 6.

Definition 3.17 (Weak Soundness)**.**
A workflow system (pn, f_i) with the source place p_i and the sink place p_o is *weak sound*, if and only if (i) for every marking f reachable from initial marking f_i, there exists a firing sequence leading from f to the final marking f_o, i. e., $\forall f : (f_i \xrightarrow{*} f) \Rightarrow (f \xrightarrow{*} f_o)$ and (ii) marking f_o

is the only marking reachable from the initial marking with a token in the sink place, i. e., $\forall f : (f_i \xrightarrow{*} f, f \geqslant f_o) \Rightarrow (f = f_o)$. ◂

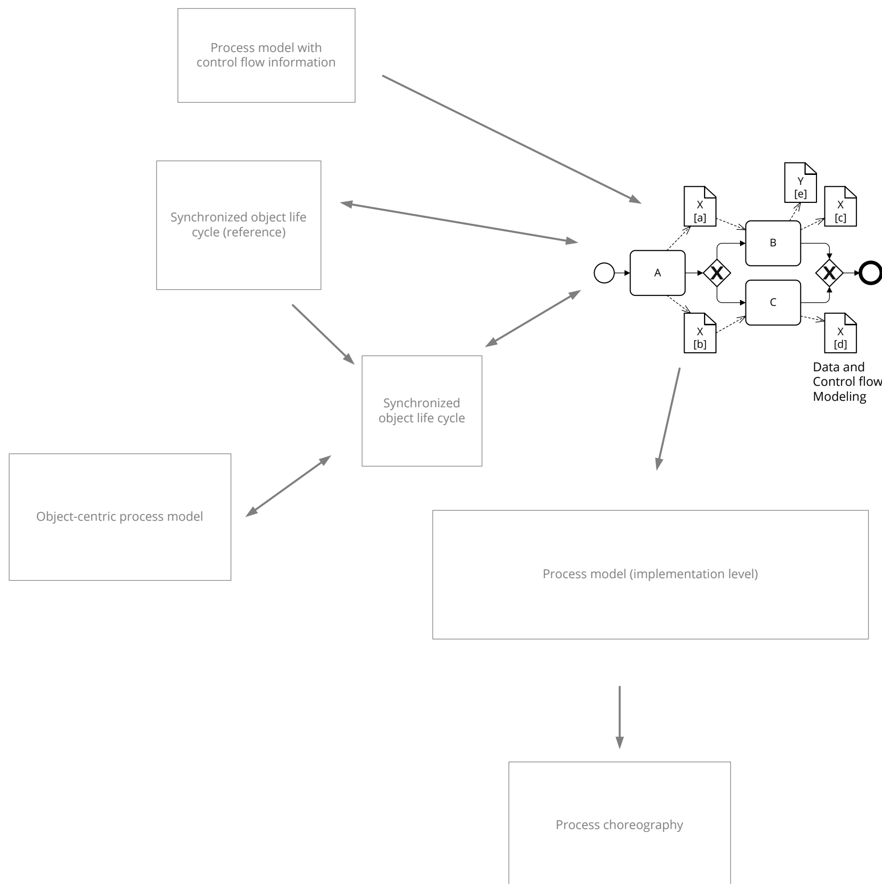
Besides these two soundness definitions, various complementary ones have been introduced for different purposes. A comprehensive discussion and comparison of major variants and further definitions regarding the correlation between soundness and further properties is given in [351].

Part II

HYBRID PROCESS MODEL FOR DATA AND CONTROL

4 PROCESS AND DATA VIEW INTEGRATION

This chapter is based on results published in [210, 211, 212, 213, 214, 215, 216, 217, 218, 219, 220, 221, 223, 224, 260, 261].



BUSINESS PROCESS MANAGEMENT (BPM) is an important approach to manage work in organizations. Many organizations face competitive markets, where process orientation enables quick reactions on market changes. In the early years, business process management focused on the design and documentation of business processes in process models that captured activities and their ordering necessary to achieve business goals. Thus, control flow was the dominant aspect in process model. Being widely adopted by industry, BPM faced new challenges and opportunities such that in recent years, further perspectives perspectives received attention – most prominently the data perspective. Section 1.2 discusses the importance of data for business process management.

Data plays a major role in process automation, process controlling, and process documentation. Data objects are the actual entities being manipulated during business process execution – manual as well as automated execution both with information system support. Data objects describe which objects need to exist prior task execution because they get utilized and they describe which objects will exist after execution. Information about the progress of a process can be retrieved from the actual processing of objects in case there exists a data object awareness, for instance, implemented through control flow and data integration. Thereby, data states describe processing results of specific data objects. Additionally, data is often used to take decisions during process execution, e. g., which path to follow. However, for information system supported process execution, many processes are not sufficiently supported from these information systems due to a missing integration of control flow and data aspects [30, 174, 231, 267, 268, 273, 278, 343, 347, 356]. These information systems usually focus on tasks and their flow of control (as in BPM for a long time) while data objects are not considered as, for instance, in the well-known and wide-spread process engine *Activiti* [2]. If at all, data is stored outside these systems and must be integrated manually during process execution.

For process controlling, data objects are generally used to specify key performance indicators which are then used to evaluate and ensure process quality. Data documentation allows to explicitly visualize the data manipulations and to show the utilized information systems and helps to analyze their inter-dependencies as usually done in master data management [52, 191]. Furthermore, representation of an organization's core assets is essential as these capture the core properties for value creation. Organization's value creation mainly bases on information about their value chain, customers, production, and research and development cycles. This information is captured in terms of data in information systems and gets used during process execution.

Summarized, explicit representation of data in process models helps to analyze processes, to execute processes, and to document the link to legacy systems as well as to visualize the core assets for an organization's value creation. In fact, the control flow describes *how* a business goal is to be achieved while the data perspectives describes *what* has to be achieved in this regard. Thus, control flow and data are "two different sides of the same coin" [269, 277] and require some integration.

Following, a business process consists of two parts: (i) process models, which orchestrate the execution of activities, and (ii) a data model, which describes the data structure within the business process, i. e., the relationships between the utilized data objects. We start with the definition of the data aspects before we motivate and describe extensions to the process model definition presented in Definition 3.1.

4.1 DATA ASPECTS

Analogously to object orientation [288], data entities are distinguished into data classes (type level) and data objects (instance level). A data class defines the attributes for which a data object may get assigned values and the data states an object may be in during process execution. Data nodes, used in process models, are associated to exactly one data class and represent data objects at model level such that they define values for a subset of attributes of the corresponding data class and the data states expected during process execution; each data node my refer to multiple data objects. In turn, each data class and data object may refer to various data nodes while each data object refers to exactly one data class. Figure 24 visualizes these relations. Speaking of data at execution level refers to data objects while data at process model level refers to data nodes; data classes are used on data type level. Formally, we define a data class as follows.

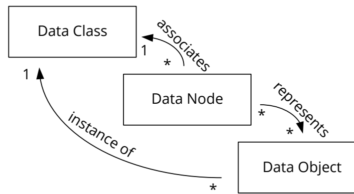


Figure 24: Correlation between data entities.

Definition 4.1 (Data Class)**.**
A *data class* $c = (name, \mathcal{J}, S)$ has a $name$ and consists of a finite set $\mathcal{J}$ of run-time attributes and a finite non-empty set S of data states ($\mathcal{J}$ and S are disjoint). ◀

Data classes

Following the schema introduced in Chapter 3, we use subscripts, e. g., $\mathcal{J}_c$ and S_c to denote the relation of sets (and functions) to data class c and omit subscripts where the context is clear. C denotes the finite set of all data classes in a specified scope. A scope usually comprises a business process but may also be extended towards multiple business processes or reduced towards single process models. If the scope is not explicitly defined, we assume that it is set to the business process. Each attribute $j \in \mathcal{J}$ is fully qualified and allows to determine the actual attribute and the corresponding data class. For instance, referring to the data model in Figure 26, *CP.supplier* indicates that attribute *supplier* of data class *Component (CP)* is referenced. Attributes we require to exist for each data class are the class' unique name used as identifier and – from the set of run-time attributes – a primary key used to distinguish data objects during process execution. The set of foreign keys is optional depending whether there exists another data object the current one refers to. For instance, in the process model in Figure 6, each object of class component *CP* refers to one object of class customer order *CO* as indicated in the data model by the foreign key attribute *CP.co_id*. Thereby, several components may reference the same customer order. The set of data states acts as enumeration such that each data object

respectively data node of this class has exactly one of these states and no other. A data node is defined as follows.

Definition 4.2 (Data Node)**.**
A *data node* $d = (name, c, s, J, pk, FK, FK^*, type_d, type_{op})$ has a $name$, refers to a data class c, and consists of a data state $s \in S_c$ and a finite set J of attributes relevant for the data node being either of type mandatory (J_M) or of type optional (J_O); $J = J_M \cup J_O$ and $J_M \cap J_O = \emptyset$. Further, a data node consists of a primary key pk, a finite set FK of foreign keys, and a set $FK^* \subseteq FK$ of all-quantified foreign keys (J, $\{pk\}$ and FK are pairwise disjoint). Sets J, $\{pk\}$, and FK denote different types of attributes which union refers to the set of run-time attributes $\mathcal{J}_c$ specified for data class c (cf. Definition 4.1); $J \cup \{pk\} \cup FK \subseteq \mathcal{J}_c$. Function $type_d : D \rightarrow \{singleInstance, multiInstance\}$ defines the instance multiplicity property and function $type_{op} : D \rightarrow \{new, delete, \perp\}$ assigns to each data node a data operation type. ◀

Auxiliary function $\varphi_D : D \rightarrow C$ assigns each data node $d \in D$ to exactly one data class $c \in C$ it refers to such that $\varphi_D(d) = c$. In a set of data objects referring to a multi-instance data node in the process model, various elements of this set may reference objects of the same class but different actual objects; e. g., the *components* in the subprocess in Figure 9 or at activity *Correlate quote information to CPs and PO* in Figure 8. To such sets of foreign keys, we refer as all-quantified foreign keys (FK^*). Figure 25 shows an example data node from the running example in Section 2.4 enriched with all annotations as introduced above.

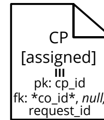


Figure 25: Example data node from running example with all annotations. The given data node is read by activity *Correlate quote information to CPs and PO* (see Figure 8). The data node is of data class component *CP*, has the corresponding label, is in data state *assigned*, has the primary key *cp_id*, has the foreign keys *co_id*, *po_id*, and *request_id*, where *co_id* is all-quantifying, is of type *multi-instance*, and the data operation type is empty indicating a read operation.

Data object

A data object is the actual entity or piece of information being processed, manipulated, or worked with during business process execution. Data states represent the results of processing a data object in the process context. Thereby, each data state describes a specific situation of interest to the organization from the data object's point of view. In detail, the set of mandatory attributes being defined, i. e., attributes containing a non-null value, describes the state of the data object. However, the object itself cannot distinguish between mandatory and optional attributes. Formally, a data object consists of a sequence of data states and is defined as follows.

Definition 4.3 (Data Object)**.**
A *data object* $o = (name, T_S, c)$ refers to its $name$. During its life time, an object traverses a sequence $T_S = \langle s_1, s_2, \ldots, s_n \rangle$ of data states. The data class c describes the object's structure. Let S_c denote the set of data states given by the corresponding data class c. Then, $\forall\, s_k \in T_S, 1 \leqslant k \leqslant n : s_k \in S_c$ holds. ◀

Given the set O of all data objects in a specified scope, auxiliary function $\varphi_O : O \to C$ assigns each data object $o \in O$ to exactly one data class $c \in C$ such that $\varphi_O(o) = c$. Auxiliary functions δ_D and δ_O return for a given data node all data objects it represents respectively for a given data object all data nodes the object refers to such that $\delta_D : D \to 2^O$ and $\delta_O : O \to 2^D$ respectively.

A data object may represent, but is not limited to, documents, forms, database fields or tables, variables, messages, and products. In the healthcare domain, for instance, a data object may also represent a patient, because the doctor is examining her, i. e., the doctor works with or manipulates (in terms of medication) the patient. In the logistics and sales domains, an activity *Ship order* hands over the *Product*s (multiple entities) referring to a *customer order* (an entity) to a logistics service provider who processes the respecting package (an entity) by sending it to the receiver (cf. Figure 7). A data object does not represent full information systems or databases; these are represented by data stores in the corresponding process models. At all points in time, a data object is in exactly one data state. The state may change over time through updates by activities being represented with multiple data nodes. A *customer order* may be, amongst others, in data states *received* indicating the retrieval of an order, in state *confirmed*, indicating that analysis succeeded successfully, and *shipped* indicating that the products referred to by the *customer order* have been handed over successfully to the logistics service provider. As the defined attributes of a data object determine its state, each data state refers to a set of values for its attributes; we define a data state as follows.

Definition 4.4 (Data State)**.**
A data state comprises a valuation of all attributes $\mathcal{J}_c$ of a data class c. Let V be a universe of attribute values. Then, the *data state* $s : \mathcal{J}_c \to V$ is a function which assigns each attribute $j \in \mathcal{J}_c$ a value $v \in V$ that holds in the current state of the corresponding object o. Thereby, data state $s \in T_{s,o}$ of object o and $c = \varphi_O(o)$ holds. ◀

At all points in time, each attribute can get assigned a value. If it is not defined, the value is set to $\perp$. To the content of a data object, i. e., its attributes and the corresponding values, we refer as *process data*.

Data model

The relationships between data classes used for data nodes' foreign key specifications are represented in a *data model* following the specifications of Unified Modeling Language (UML) class diagrams [244]. We

generally assume that the scope for classes in the data model is a business process. The subsequent definition specifies the subset of concepts from UML class diagrams required to describe the data dependencies between data classes in business processes.

Definition 4.5 (Data Model)**.**
A data model $dm = (C, \mathfrak{R})$ consists of a finite non-empty set C of data classes (cf. Definition 4.1) with data relations $\mathfrak{R} \subseteq C \times C$ between them. Data relations are either undirected associations $\mathfrak{R}_{Assoc}$ or directed parental data relations $\mathfrak{R}_P$. Parental data relations are of types composition ($\mathfrak{R}_{Comp}$), aggregation ($\mathfrak{R}_{Aggr}$), or generalization ($\mathfrak{R}_{Gen}$) such that $\mathfrak{R}_P = \mathfrak{R}_{Comp} \cup \mathfrak{R}_{Aggr} \cup \mathfrak{R}_{Gen}$. The UML concept of generalization sets is extended to relation clusters RC such that clustering is available for any type of the introduced parental data relations $\mathfrak{R}_P$. ◀

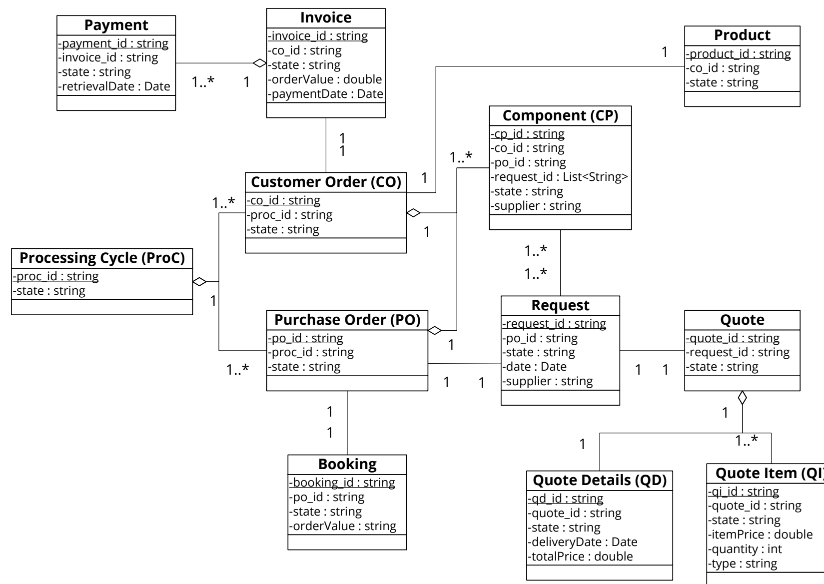


Figure 26: Data model for the running example from Section 2.4 from the *Computer retailer* point of view.

Figure 26 shows a data model applicable for the example introduced in Section 2.4 from the computer retailer point of view. A *Processing Cycle (ProC)* contains an arbitrary positive number of *Customer orders (CO)* and *Purchase orders (PO)* which in turn contain *Components (CP)*. *Components (CP)* and *Requests* are in an m:n relationship and each *Request* refers to exactly one *Purchase Order (PO)*. Furthermore, each *Request* is referred to by a *Quote* that contains exactly one representation of *Quote Details (QD)* and an arbitrary positive number of *Quote Items (QI)* representing the list items of the *Quote*. For each *Purchase Order (PO)*, one internal *Booking* exists while each *Customer Order (CO)* is referred to by an *Invoice* that in turn is composed of at least one *Payment*. Finally, a *Product* summarizes the actual items ordered by a customer and is therefore associated to the *CO* and has the corresponding foreign key *co_id*. In the given exam-

ple, generalization sets are not required. However, generalization sets may, for instance, be used to indicate that a *Quote* either consists of one representation of *Quote Details (QD)* and an arbitrary number of *Quote Items (QI)* (see above) or consists of an arbitrary number of *Quote Items (QI)* and a *warranty agreement*.

Object life cycles

While a data model defines the structure of data classes, an *object life cycle (OLC)* describes the behavior of data objects by specifying the manipulations allowed to be performed on an object by actions in which situation represented by the data object's state, i. e., an OLC describes the partial ordering of data states and specifies the actions inducing this order. In a business process acting on a data object, each activity comprises any number of consecutive actions of the corresponding life cycle. This connection is enabled by assigning exactly one OLC to each data class in the respecting data model; function $\eta : L \rightarrow C$ provides this mapping. An OLC is a state transition net and is formalized as in Definition 3.8.

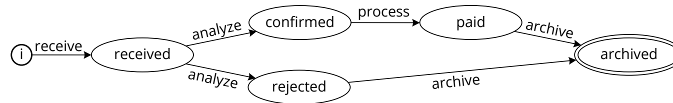


Figure 27: Object life cycle of data class *Customer Order (CO)* for the process model in Figure 5 on page 29 consisting of six data state transitions and six data states where *i* is the initial one and *archived* the final one.

Figure 27 shows the object life cycle of data class *Customer Order (CO)* from the running example in Section 2.4 for Figure 5 hiding some details of the subprocesses. For instance, the transition from data state *confirmed* to *paid* actually consists of three additional intermediate data states: *accepted*, *shipped*, and *invoiced* that are passed in this execution order. Details will follow in the remainder of this section. The OLC given in Figure 27 allows two execution sequences to reach the final state *archived* from the initial state *i*. Depending on the order analysis, the *CO* gets either confirmed or rejected. In both cases, the traces lead to the same final data state.

Generalizing the OLC semantics from Chapter 3, an activity may change a data state s to some state s' preceding s, i. e., data state s may be changed to data state s' if there exists an execution sequence σ_S such that $s \stackrel{\sigma_S}{\longmapsto} s'$. Further, a data object is in exactly one data state at each point in time. Following, parallelism must not be present in an OLC.

OLC synchronization

Additionally, a manipulation performed on one data object often does not only rely on the current data state of this data object but on the data states of further data objects as well. For instance, orders may only be received from customers after a new processing cycles was started or a processed customer order may only reach data state *paid* if the invoice is paid correctly. To handle these inter object life cycle dependencies, we

introduce the concept of *object life cycle synchronization*. In this context, we define active data states – a prerequisite – as follows.

Definition 4.6 (Active Data State)**.**
A data state in an object life cycle is *active* at a specific point in time, if it is one of the data states the corresponding data object may currently be in at this specific point in time. ◂

Deciding whether a certain data state is active requires information about manipulations done to that data object. This information can be derived, for instance, from process models. All data states on a path from the one accessed last to the one accessed next in the corresponding object life cycle including these two states are considered active. Thereby, the next accessed data state is only considered active, if it read. In case, it gets written, this data state is excluded from the set of active data states. If an activity reads (writes) multiple data nodes of the same data class, all of them are considered accessed last respectively accessed next. In case, the activity representing the current point in time only reads (writes) nodes of the specific data class, activities succeeding (preceding) this activity are inspected until a match is found. If there exists no last accessed data state, the initial one is considered last accessed. If there exists no next accessed data state, all reachable final data states are considered next accessed. Assume the OLC of data class *Customer order (CO)* as given in Figure 28; the extension of the OLC in Figure 27 by data states *accepted*, *shipped*, and *invoiced*. The latter two are introduced between data states *confirmed* and *paid* while data state *accepted* can be reached from state *received* and may lead either to state *confirmed* or to state *rejected* splitting the decision taking about order confirmation and rejection into two steps. Relating this object life cycle to the process model in Figure 5 on page 29 (the overview process model of the build-to-order and delivery process from the computer retailer's point of view), after execution of activity *Start processing cycle*, only the initial data state *i* and is considered active, since there exists no last accessed state and the next accessed state *received* is written. After termination of activity *Analyze order* and before execution start of activity *Process order*, the active state is *confirmed* since it was last accessed and will be next accessed. During execution of activity *Process order*, the set of active data states comprises states *confirmed*, *shipped*, and *invoiced*. State *paid* as next accessed one is excluded. After introducing active data states, we proceed with the concept of object life cycle synchronization resulting in a *synchronized object life cycle*.

Definition 4.7 (Synchronization Edge)**.**
A synchronization edge $se = (src, tgt, dep)$ consists of a source src, a target tgt, and an optional dependency type dep to connect multiple object life cycles synchronizing the data state transitions between

them. Thereby, it either connects two data states or two data state transitions of two object life cycles l_1, l_2 defining preconditions towards data state transitions or ensuring joint execution of the connected transitions respectively. An undirected or untyped synchronization edge $se_{\mathfrak{T}_S} = (t_1, t_2)$ connects data state transitions $t_1 \in \mathfrak{T}_{S,l_1}$ and $t_2 \in \mathfrak{T}_{S,l_2}$ ($l_1 \neq l_2$). The third attribute, dep, is not used. A directed or typed synchronization edge $se_S = (s_1, s_2, dep)$ connects data states $s_1 \in S_{l_1}$ and $s_2 \in S_{l_2}$ ($l_1 \neq l_2$) with s_1 being the source data state, s_2 being the target data state, and $dep = \{currently, previously\}$ describing the type of dependency between these data states. ◂

For typed synchronization edges connecting data states, currently means that the source data state must be active in the corresponding OLC if a transition to the target data state shall occur in another OLC. Previously relaxes this requirement such that the source data state must have been active some time in the past to allow the data state transition to the target data state. Two data state transitions connected by an untyped synchronization edge get combined such that they are executed together. This property is transitive. To ensure proper specification of synchronization edges, typed ones shall not impose circular dependencies, e. g., data state a requires b and state b requires a with a and b belonging to different OLCs. For untyped synchronization edges, two edges sharing one transition must not have a transition of the same OLC as second parameter; e. g., for untyped synchronization edges $se_{\mathfrak{T}_S,1} = (A, B)$ and $se_{\mathfrak{T}_S,2} = (B, C)$, transitions A and C must belong to different OLCs. SE denotes the finite set of all synchronization edges in a specified scope.

Putting above concepts together, we define a synchronized object life cycle as follows.

Definition 4.8 (Synchronized Object Life Cycle)**.**
A *synchronized object life cycle* $\mathcal{L} = (L, SE)$ consists of a finite non-empty set L of object life cycles and a finite set SE of synchronization edges connecting various object life cycles. ◂

Sync. OLC visualization

Visualization of synchronization edges is achieved by dotted directed edges between the data states and a label with respect to the type of dependency respectively by dotted undirected edges between the data state transitions stated in each tuple. Labeling-wise, a *c* represents type *currently* while a *p* represents type *previously*.

Sync. OLC state reachability

A data state s within an OLC is only reachable if the dependencies described by the synchronization edges are fulfilled, i. e., all transitions connected with the one leading to s are enabled and all data states in other OLCs targeting s with a synchronization edge hold or held once depending on the dependency type. Multiple edges with the same target data state are handled with respect to the origin of the source data state. If they belong to the same OLC, the described dependencies are disjunctions. If they belong to different OLCs, the described dependencies are conjunctions. Each process model is associated to exactly

Figure 28: Synchronized object life cycle for data classes *ProcessingCycle*, *Customer Order (CO)*, and *Invoice* data classes.

one synchronized object life cycle through data class and single OLCs relationships.

Figure 28 shows the synchronized OLC for the build-to-order and delivery process given in Section 2.4 containing object life cycles of classes *Customer Order (CO)*, *Processing Cycle (ProC)*, and *Invoice*. For readability reasons, we omitted the transition labels in this figure. Each single OLC is labeled with the corresponding class name. The customer order life cycle is the one from Figure 27 extended by three data states replacing the transition from state *confirmed* to state *paid*. These states are *accepted*, *shipped*, and *invoiced* that are passed in this execution order. Additionally, data state *rejected* cannot only be reached directly from data state *received* but via state *confirmed* as well. The processing cycle OLC consists of four data states in sequence: *i*, *started*, *received*, and *finalized*. Data state *finalized* can only be reached if the corresponding *CO* is *currently* in data state *archived*. The occurrence of data state *started* of a *ProC* object allows a dependent *Customer Order (CO)* to get *received* independently from the state the *ProC* is in. The invoice OLC has a more complex structure than the other two OLCs and allows infinitely many traces. The transitions connecting data states *shipped* and *invoiced* as well as *created* and *sent* are executed together, i. e., the data state transitions reference the same activity in a process model, ensuring that objects of classes *CO* and *Invoice* respectively are manipulated together. However, both state transitions *may* be ordered during activity execution. The synchronization edge of type *previous* connecting state *sent* of an *Invoice* and state *invoiced* of a *Customer Order (CO)* ensures that, although being executed in one transaction, first data state *sent* is set. Furthermore, a *Customer Order (CO)* may only reach data state *paid* if the *Invoice* is currently in state *paid*. Data state *archived* may only be reached by a *CO* if the *Invoice* currently is either in state *i* for initial, in state *canceled*, or in state *paid*.

When a certain data state transition shall take place, the synchronization validation function has to be executed for each affected synchronization edge. If all these validation functions evaluate to true, the transition takes place. Otherwise, the transition must not execute.

Definition 4.9 (Synchronization Validation Function)**.**
Given a synchronization edge $se = (src, tgt, dep)$, the *synchronization validation function* $\Pi : SE \rightarrow \{true, false\}$ evaluates to true, if both data state transitions are enabled or if the data state src is active (either dependency type) or was active earlier (dependency type previously) in the corresponding object life cycle. Otherwise, Π evaluates to false. ◀

4.2 BUSINESS PROCESS MODELS

In Definition 3.1, we introduced an activity-centric process model (ACP) with the most common modeling concepts [171, 389]. With respect to data and the contributions of this thesis, the process model definition needs to be extended. Next, we describe and motivate the extensions before we present the revised definition in Definition 4.10.

As discussed in Chapter 3, each activity is executed by human resources (user tasks) or in the responsibility of human resources (service, send, and receive tasks). This is notated by a resource being assigned to an activity. Although, patterns have been identified describing the execution of one activity by many resources [208, 290], we restrict to exactly one resource being assigned to one activity. However, an activity may be offered to multiple human resources, where one is chosen to finally execute the activity. Handling a team (multiple resources) as single resource is allowed and reduces the impact of our limitation.

Process execution

During process execution, each activity works on a set of data objects represented as data nodes in the process model. Following, the resource executing the activity requires rights to access the corresponding data objects and a data object is potentially used by multiple activities. On model level, we therefore define for each data node a set of resources, which get granted access to it. For service tasks, we assume that this right is handed over to the information system running the corresponding service. A global access rights assignment works on class level allowing a resource to access all data nodes referring to the particular class of data objects. We restrict to the data node level, because access management is not in the focus of this thesis. Summarized, an activity can only be executed by or in the context of a specific human resource, if the preconditions (input data nodes) specified by the data flow hold and if the resource entitled to execute the activity gets granted access to the objects referenced as data nodes in the pre- and postconditions (input respectively output data nodes).

Probabilities

Only some of the data nodes specified in the process model may get accessed during activity execution such that there exists an access prob-

ability for each of the nodes. To identify data nodes with an probability higher than zero, we also need the capability to determine the data nodes that may be accessed by a specific activity. The objects processed during process execution need to be persisted to retain intermediary and final results. Therefore, one the model level, we utilize *data stores*, which are connected to data nodes indicating that all information of the corresponding data object is stored in this location. A data store can represent any information system or database.

Data From business artifacts [237], we adopt the concept that each process is "driven" by one class of data objects, which links and relates the other classes to each other. We call the objects of this class *case object*. XOR-gateways with multiple outgoing control flow edges require guidance to choose the correct path for further process execution. We assign to each such control flow edge an expression which evaluation determines the path choice. To these gateways, we also refer as *data-based XOR gateway*. The expression contains a data class with a respecting data state which has to be matched. As taking a choice implies that not each control flow edge is taken for each process execution, there exist probabilities indicating how often one control flow edge is chosen. These probabilities can be assigned to control flow edges manually or they can be derived from process logs via process mining. We also allow assignment of an expression to data flow edges with a semantically different meaning. These expressions indicate the number of data nodes being affected by the corresponding data flow. This plays a major role in the context of decomposing one object into multiple others all being of the same class.

Utilization Finally, each process model belongs to exactly one business process. The extensions regarding labels and decision taking at XOR-gateways are mainly required due to the contributions described in this chapter as well as in Chapters 5, 6, and 7. Persistence, case object, and data flow expression extensions result from Chapter 8. The probability extensions as well as the assignment of resources to data nodes for specifying access rights result from work in additional areas of BPM being considered for more general application opportunities of this framework; namely, these are the areas of business process monitoring and business process model abstraction (see Section 9.2). Next, we define the fully specified ACP in two parts; first the actual process model followed by data-specific configurations specified at process model level.

Definition 4.10 (Activity-centric Process Model)**.**
An *activity-centric process model* $pm = (N, D, DS, Q, R, bp, \mathfrak{C}, \mathfrak{F}, type_a, type_t, type_g, \mu, \beta, \kappa, DCF)$ consists of a finite non-empty set $N \subseteq A \cup G \cup E$ of control flow nodes being activities A, gateways G, and event models E, a finite non-empty set D of data nodes, a finite set DS of data stores used for object persistence, a finite non-empty set Q of activity labels, and a finite set R of resources actually executing the activities

and accessing the objects of the process model (N, D, DS, Q, and R are pairwise disjoint). Further, it is part of one specific business process bp. $\mathfrak{C} \subseteq N \times N$ is the control flow relation and $\mathfrak{F} \subseteq (D \times A) \cup (A \times D)$ is the data flow relation representing read and write operations of activities with respect to data nodes.

Function

- $type_a : A \rightarrow \{task, subprocess, multiInstanceTask, multiInstanceSubprocess\}$ gives each activity a type,
- $type_t : A \nrightarrow \{user, service, send, receive, unspecified\}$ specifies the type of each (multi-instance) task,
- $type_g : G \rightarrow \{XOR, AND\}$ assigns to each gateway a type,
- $\mu : A \rightarrow Q$ assigns to each activity a label,
- $\beta : A \rightarrow R$ assigns to each activity a resource, which executes the corresponding activity,
- $\kappa : \mathfrak{C} \rightarrow [0, 1]$ defines the probability a control flow edge is chosen for execution.

For each process model, a data-specific configuration DCF may be defined and applied to the process model. We refer to an activity-centric process model as process model if the context is clear. ◀

Definition 4.11 (Data-specific Configurations).
A data-specific configuration $DCF_{pm} = (\mathfrak{P}, \lambda, \gamma, \xi, case, \varkappa)$ for a process model pm allows specifying in which database a data object is stored respectively from which database it is retrieved via the persistence relation $\mathfrak{P} \subseteq (DS \times D) \cup (D \times DS)$.

Function

- $\lambda : A \times D \rightarrow [0, 1]$ determines the probability a data node is accessed by an activity and
- $\gamma : D \rightarrow 2^R$ assigns to each data node a set of resources being allowed to access it.

Partial function

- $\xi : G \times N \nrightarrow D$ assigns data conditions in terms of one data class with one data state to control flow edges having an XOR gateway as source,
- $case : A \cup \{pm\} \nrightarrow C$ defines the case object for the given process model and each activity $a \in A_{pm}$ where $type_a(a) \neq task$, and
- $\varkappa : \mathfrak{F} \nrightarrow exp$ optionally assigns an expression exp to a data flow edge.

◀

Function explanation

Function $\rho : PM \rightarrow BP$ assigns each process model $pm \in PM$ to the corresponding business process $bp \in BP$. $\lambda(d_1, a)$, the probability data node $d_1 \in D$ is accessed by activity $a \in A$, evaluates to a value greater 0, if data node $d_1 \in D$ is read or written by activity $a \in A$; $\lambda(d_1, a) > 0 \Leftrightarrow ((a, d_1) \in \mathfrak{F} \vee (d_1, a) \in \mathfrak{F})$. $\lambda(d_1, a)$ evaluates to 1, if and only if d_1 is read (written) by activity a and if

there does not exist another data node $d_2 \in D$ of the same class, i. e., $\varphi_D(d_1) \not\approx \varphi_D(d_2)$, read (written) by a. Otherwise, i. e., if no read or write association exists, λ evaluates to 0. $\varphi_D : D \to C$ is a function mapping a data node $d \in D$ to the corresponding data class $c \in C$. Let $c \in C$ be the data class of multiple data nodes $d_i \in D$ such that $\forall\, d_i : \varphi_D(d_i) = c$ read (written) by activity a, the probability λ is determined by the execution probability of preceding (succeeding) branches, where the corresponding objects are accessed – or equal distribution if no information is given. The probabilities of data nodes of one class read (written) by one activity have to sum up to one: $\forall\, d_i \in D$ where $\varphi_D(d_i) = c \wedge (d_i, a) \in \mathfrak{F} : \sum_{d_i} \lambda(d_i, a) = 1$ for read operations and $\forall\, d_i \in D$ where $\varphi_D(d_i) = c \wedge (a, D_i) \in \mathfrak{F} : \sum_{d_i} \lambda(d_i, a) = 1$ for write operations. Similarly, the probabilities for all outgoing control flow edges of a node $n \in N$ must sum up to one: Let $n_i \in N$ denote all control flow nodes directly preceding n, i. e., $(n, n_i) \in \mathfrak{C}$, then $\forall\, n_i : \sum_{n_i} \kappa(n_i) = 1$. Utilizing functions λ, β, and γ, data node $d \in D$ is accessed by resource $r \in R$ during execution of activity $a \in A$, if $\beta(a) \in \gamma(d)$ and $\lambda(a, d) > 1$.

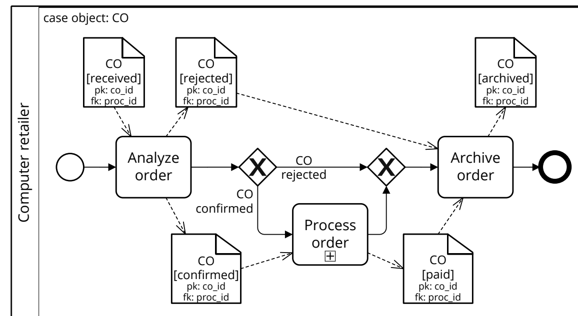


Figure 29: Process model.

Figure 29 shows the process model from Chapter 3 representing an extract from the build-to-order and delivery process from Section 2.4. This process model is annotated with the new data concepts (primary key, foreign key, data operation type) and will be used to show application of above functions exemplarily. $\lambda(\mathit{CO}, \mathit{Analyze\ order}) = 1$ since, for the input data nodes, *CO* is the only one. Considering the output data nodes only, each of them would be accessed with a probability of 50% – equal distribution since no further information is given. If κ would return 0.9 for the control flow edge connecting the XOE split and activity *Process order* and 0.1 for the control flow edge connecting both gateways, then, $\lambda(\mathit{CO}, \mathit{Analyze\ order})$ would return 90% probability for state *confirmed* and 10% for state *rejected*. Having input and output data nodes assigned to an activity, the maximum probability value for each data class is selected as result of $\lambda(d, a)$. In this example, the maximum of 1, 0.9, and 0.1 is 1 – the above stated result for $\lambda(\mathit{CO}, \mathit{Analyze\ order})$. Since $\lambda(a, \mathit{CO})$ returns for all data nodes and all activities a a value greater 0 and since $\beta(a)$ returns the *Computer Retailer* for each activity a, access of a data node during process execution is based on the as-

sociation of resources to data nodes. For all data nodes, the *Computer Retailer* is allowed to access a corresponding data object, the access will take place during execution; i. e., $\gamma(CO) =$ *Computer Retailer* is required. The remaining ones lead to deadlock situations because no one allowed to execute the activity is allowed to access the associated data node.

Following the schema introduced in Chapter 3, we use subscripts, e. g., A_{pm}, D_{pm}, and μ_{pm} to denote the relation of sets and functions to process model pm and omit subscripts where the context is clear. Data stores reference full information systems or databases, where the associated information is stored. Each activity contains exactly one activity label describing the work to be performed to execute this activity. According to the grammar introduced in Definition 3.3, three styles may be used for activity labeling: verb-object labeling (*Analyze order*), action-noun labeling (*Order analysis*), and descriptive labeling (*Computer retailer analyzes order*).

Integration of the traditional process and the data view requires to extend the structural correctness criteria formulated in Chapter 3 with respect to data. Therefore, we expect each process model pm to satisfy the structural correctness criteria SCC-1 to SCC-6 as well as:

(SCC-7) exactly one condition is assigned to each outgoing control flow edge of a split, whereas all other control flow edges must not get assigned a condition,

(SCC-8) the assigned data conditions must be non-blocking,i. e., at all times, at least one of the conditions must evaluate to true[1], and.

(SCC-9) each data node may only be read *or* written by some activity but a data node must not be read *and* written by the same activity as each write changes a data node (object)[2] and thus changes its state.

Scope

For the direct contributions of this thesis, a process model $pm = (N, D, DS, Q, R, bp, \mathfrak{C}, \mathfrak{F}, type_a, type_t, type_g, \mu, \beta, DCF)$ with $DCF_{pm} = (\mathfrak{P}, \xi, case, \varkappa)$ is sufficiently specified.

Node Life Cycle

Generalizing the concept of activity life cycles [370], each control flow node contains a node life cycle describing the states and state transitions applicable to that node during process execution. These can be separately defined for each node at any complexity level. In the course of this thesis, we assign to each activity the same node life cycle as given in Figure 30. The states applicable for an activity during process execution are *initialized*, *enabled*, *running*, and *terminated*. Further, each activity has an unlabeled initial state. The node state transitions connecting appropriate pairs of node states are *initialize*, *enable*, *start*, and *terminate* respectively. Each activity gets initialized upon process instantiation. An activity is enabled when it is ready for immediate execution. Running indicates current execution of the activity while terminated indicates execution completion. In the context of this thesis,

1 In case multiple conditions evaluate to true, the first one is taken and disables all others. Order is given by the process model in reading direction (left to right and top to bottom)

2 At design-time, it is the data node. At run-time, it is the data object.

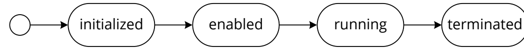


Figure 30: Activity life cycle (cf. [243, 370]).

we abstract from aborted and interrupted as well as misbehaving or skipped activities (see, for instance, [243, 370]) and we assume proper termination in all cases. Activities being initialized but not executed, e.g., not chosen paths in XOR blocks, remain initialized as final state. In practice, however, these activities are usually transitioned into a final state *skipped*. Analogously to activities, the remaining control flow nodes (gateways and event models) get assigned a similar node life cycle containing the node states *initialized*, *enabled*, and *terminated* and the corresponding node state transitions *initialize*, *enable*, and *terminate* respectively. Formally, we define a node life cycle as follows.

Definition 4.12 (Node Life Cycle)**.**
A *node life cycle* $nl = (NS, \mathfrak{T}_{NS}, n)$ contains a finite non-empty set NS of node states with $\mathfrak{T}_{NS} \subseteq NS \times NS$ denoting the node state transition relation. Further, node life cycle nl is assigned to a control flow node that is referenced by n. ◂

NL denotes the set of all node life cycles defined for control flow nodes N_{pm} of process model pm. Auxiliary function $\varpi : N \rightarrow NL$ assigns each node $n \in N$ of pm to one node life cycle $nl \in NL$.

4.3 PROCESS INSTANCE VIEW

Process state

Having an integrated view on control flow and data, a process instance still consists of a sequence of process instance states (cf. Definition 3.6), which in turn refer to markings of the process model. However, the marking of a process model needs to consider control flow and data aspects instead of control flow only as introduced in Definition 3.4. Therefore, we redefine the mapping from Definition 3.4 such that there exists a control flow mapping $m_{\mathfrak{C}} : \mathfrak{C}_{pm} \rightarrow \mathbb{N}$ as given in Definition 3.4 and a data flow mapping $m_{\mathfrak{F}} : \mathfrak{F}_{pm} \rightarrow \mathbb{N}$. The union of both markings $m_{\mathfrak{C}}$ and $m_{\mathfrak{F}}$ represents the marking m_{pm} of process model pm. In each such marking, all activities with sufficiently marked control flow and data flow edges are enabled. Details about activity enablement will follow in Section 4.7.

A single *process instance state* can now be defined using the states of data objects being processed during execution of the corresponding process instance.

Definition 4.13 (Process Instance State)**.**
Let i be a process instance of process model pm and let O_z denote the set of data objects being present in a *process instance state* z. Each data

object $o \in O_z$ belongs to one data class $c \in C$ and has a specific state $s \in S_c$ in process instance state z. H_z is the set of data states in process instance i such that the process instance state function $z_i : O_z \rightarrow H_z$ assigns to each data object its current state in i. ◀

Process instance state

The state z of a process instance is composed of a finite non-empty set of states of data objects, each belonging to a data class used in the corresponding process model. A sequence of process instance states describes a process instance of process model pm as defined in Definition 3.6 and each such state refers to a marking of pm. Under the assumption that no two identical activities exist in pm, i. e., two activities having identical input data nodes and identical output data nodes, the process instance state is sufficient to deduce the marking of process model pm. Indeed, the marking of one process model pm_2 may be dependent on the execution of another process model pm_1, if an activity of pm_1 writes a data object which is afterwards also manipulated by some activity in pm_2. In fact, the data flow component $m_{\mathfrak{F}}$ of m_{pm_2} is affected. Assuming, there also exists an activity in pm_1 that reads the object in the same data state as pm_2 does, but this activity does not change the state, then the order of execution is important. If the activity in pm_2 gets executed before the reading activity in pm_1, process model pm_1 may deadlock, if the required state is not restored by some other activity.

Figure 31 shows three process models extracted and slightly deviated from the running example manipulating data classes *PO*, *Request*, and *Booking*. The latter two are handled in exactly one of the process models while the *PO* is handled in all three models resulting in inter-model dependencies. First, activity *Create purchase order* sets data state *created* for an object of class *PO*. Second, activities *Approve purchase order* and *Create request* can be executed in any order. After termination of activity *Approve purchase order*, the activities in the process model in Figure 31c can be executed. Proper termination of this process model leads to a change of the process instance state of the process model in Figure 31a since the object of class *PO* was changed to state *purchased* that now enables activity *Book purchase internally*; the process instance state changes from {approved,initial} to {purchased,initial}. After execution of activity *Book purchase*, the process instance state is {booked,created}. For the process model in Figure 31a, the sequence {initial,initial}, {created,initial}, {approved,initial}, {purchased,initial}, and {booked,created} describes the discussed process instance. At any point during process execution, the current sequence of the process instance states leads to the marking of the process model.

Figure 32 visualizes the control flow and data flow components of the marking after termination of activity create purchase order. The corresponding sequence of process instance states is {initial,initial} and {created,initial}. Activity *Specify supplier* of the process model in Figure 31b depends on the execution of process model Figure 31c; executing activ-

(a)

(b) (c)

Figure 31: Three process models extracted and slightly deviated from the running example in Section 2.4 manipulating data classes *PO*, *Request*, and *Booking*. The process model in (a) requires termination of (c) while (b) deadlocks if activity *Order PO* of (c) is executed before activity *Specify supplier* of (b) since the object of data class *PO* is not available in state *approved* anymore.

Figure 32: The control flow and data flow components of the marking for the process model introduced in Figure 31a after execution of activity *Create purchase order*. Highlighted edges visualize the enablement. The object of class *PO* is in state *created* and allows the enablement in combination with the control flow marking the enablement of activity *Approve purchase order*. Assuming an ordering from left to right and top to bottom, the control flow and data flow components of the marking can be represented by arrays such that $m_{\mathfrak{C}} = [0,1,0,0]$ and $m_{\mathfrak{F}} = [0,1,0,0,0,0]$.

ity *Order PO* before it disables the execution of activity *Specify supplier* such that the corresponding process model in Figure 31b deadlocks at the second activity.

Further, Z denotes the set of process instance states of one process model. Given a business process consisting of multiple process models PM, the set $\mathcal{Z}$ denotes the set of current process instance states for one case such that each process instance state $z \in \mathcal{Z}$ belongs to a different process instance i_k, where each belongs to the corresponding process model $\rho_I(i_k) = pm_k \in PM$.

The correlation between a process instance and its corresponding data objects is realized via the data object's primary and foreign keys as discussed in Chapter 8. Briefly, the case object is correlated to the process instance and all further data objects used within a process model are directly or indirectly related to the case object.

Process instance distinction

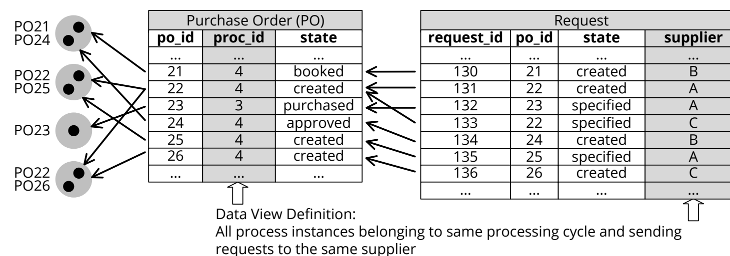
Multiple process instances can be differentiated with respect to various properties, e. g., running vs. not running, terminated in time vs. terminated over time, reference to a process model, actual process participants executing the instance, or based on their data utilization. Targeting the last property, we introduce the concept of *instance data views* for business process instances. The instance data view is a projection on the values of *relevant* data attributes in a specific state of one process instance. Process instances with identical values for all relevant data attributes are considered *similar*. Relevance is specified by the process designer through the *data view definition* – a set of fully qualified data attributes. Thus, the instance data view specification requires as input a set of data attributes being of interest for a business situation as well as the current state of a process instance that is executed in the corresponding business environment represented by a process model. Formally, we define the instance data view as follows.

Definition 4.14 (Instance Data View)**.**
Let $X = [x_1, x_2, \ldots, x_k]$ be a list of fully qualified data attributes of interest in process model pm referred to as *data view definition*, where for each x in list X holds that $x = c.j$ such that $c \in C_{pm} \wedge j \in \mathcal{J}_c$. Then, an *instance data view* for a process instance i in a specific state z is a list of values $V' = [v_1, v_2, \ldots, v_k]$. Given the relevant data attribute $x_l = c.j, 1 \leqslant l \leqslant k$, the corresponding value v_l is calculated by data view function $\psi(x, i) = \psi(c.j, i) = (z_i(o))(j)$, where object o is an instance of data class c used in process instance i; $\varphi(x, z) = v_l \Leftrightarrow z_i(o) = s_l \wedge s_l(j) = v_l$ as given in Definitions 4.13 and 4.4 respectively. ◀

The data view function $\psi(x, z)$ returns for a given relevant data attribute $x = c.j$ the value of a data object o of class c in the given process instance state z. To identify the corresponding object o from the given, fully qualified data attribute x and the process instance state z, auxiliary function $\iota : C \times \{z\} \rightarrow O$ is used. $\iota(c, z) = o$ crawls the set of data objects for which a data state is given in process instance state z and

returns the one object corresponding to data class c by utilizing function φ_O. Data class c being input to the function ι can be derived from the data attribute x by name matching – the part preceding the dot which separates the class name and the attribute name. Then, data view function ψ uses the process instance state function $z(o)$ to return the state of data object o and the data state function $s_o(j)$ to return the value for the given data attribute j. In fact, both functions are combined to $\psi(x, z) = (z_i(o))(j)$ which is executed once for each relevant data attribute $x = c.j$ to construct the instance data view V' for one process instance i.

After providing the formal discussions on the concept of data views, we now explain it using the simplification of the build-to-order and delivery scenario as shown in Figure 31b. In this example, we utilize data classes purchase order *PO* and *Request* with the attributes as given in the data model (cf. Figure 26 on page 64). The data view definition X comprises the data attributes x_1 = *PO.proc_id* and x_2 = *Request.supplier* to allow distinction of process instances based on their correlation to the processing cycle *ProC* and which *supplier* is chosen to send a request to (see gray-colored columns in Figure 33). In this figure, each table represents one data class and each row in some table represents a distinct data object with *Request* objects referring to *PO* objects (see correlation arrows between both tables). For each purchase order to be handled, one process instance of the model in Figure 31b is instantiated and executed. Considering the process instance handling the purchase order with *po_id* = 21, the values of the data attributes indicate that this process instance must be terminated already since the state of object *PO* is $z(PO)$ = *booked* that is reached after execution of activity *Book purchase internally* that follows – in case of correctly ordered process execution – activity *Specify supplier*. The data state function returns the following values for the given relevant attributes: $s_{PO}(proc_id) = 4$ and $s_{Request}(supplier)$ = B. Thus, the resulting data view for the process instance handling *PO* with id 21 is [4, B]. Referring to Figure 33, it is highlighted together with the instance data views for the other process instances by gray colored columns in the presented tables.



Purchase Order (PO)		
po_id	proc_id	state
...	...	...
21	4	booked
22	4	created
23	3	purchased
24	4	approved
25	4	created
26	4	created
...	...	...

Request			
request_id	po_id	state	supplier
...	...	...	...
130	21	created	B
131	22	created	A
132	23	specified	A
133	22	specified	C
134	24	created	B
135	25	specified	A
136	26	created	C
...	...	...	...

Figure 33: Example of a data view definition and the resulting data view clusters.

In a data view definition, attributes of multiple data classes can be used as shown in above example. Indeed, this increases the complexity

of creating and managing these views. However, handling complexity is a question of tooling and will not be addressed in this chapter.

Instances of one process model can be distinguished and grouped based on their instance data view by assigning each instance to corresponding *data view clusters*. Process instances with similar instance data views are collected in the same cluster. Thereby, a single process instance might be correlated to multiple data view clusters – depending on the data view specification. A data view cluster is defined as follows.

Definition 4.15 (Data View Cluster)**.**
Let I be a set of process instances of one process model pm. A *data view cluster* w is a set of process instances $I' \subseteq I$ of pm, where all process instances $i \in I'$ share the same instance data view V'. ◀

The set of all data view clusters for one process model is denoted as W. As illustrated on the left in Figure 33, grouping the given process instances based on data view definition $X = [$*PO.proc_id*, *Request.supplier*$]$, four different data view clusters exist. The process instances handling the purchase orders *PO* with ids 21 and 22 belong to the same cluster since both refer to the processing cycle with id 4 and for both, there exists a *Request* that is sent to supplier B. The process instance of purchase order *PO* with id 22 is assigned to two data view clusters since there exist two requests that are sent to different suppliers. Although referring to the same processing cycle (id = 4), the process instances of purchase orders *PO* with ids 21 and 22 belong to different data view clusters since both do not send a request to the same supplier – B compared to A and C.

4.4 BUSINESS PROCESSES

After discussing the single concepts, we put them together to *process scenarios* and *business processes*. These relations are shown in Figure 38 and will also be discussed in Section 4.6. A process scenario comprises a single process model and describes its structure and behavior.

Definition 4.16 (Process Scenario)**.**
A *process scenario* $ps = (pm, C, \mathcal{L})$ consists of a process model pm, a finite set C of data classes utilized in process model pm, and a synchronized object life cycle $\mathcal{L}$ comprising all data classes used in pm. ◀

Process scenarios comprise single steps within larger scope business processes or present multiple views of the same business process such that each business process comprises multiple related process scenarios and a data model determining the structure of and relationships between data objects utilized during business process execution (cf. informal definition in Definition 1.1). Formally, we define a business process as follows.

Definition 4.17 (Business Process)**.**
A *business process* $bp = (PS, dm)$ consists of a finite non-empty set PS of process scenarios and a corresponding data model dm. ◀

All classes of the data nodes (objects) used in one process model of the business process are contained in the business process' data model. For each process model contained in process scenario, there exists a tree within this data model with the process model's case object (see Definition 4.10) being the root of that tree. Thereby, classes not used in the process model but linking two utilized classes are also part of that tree. Figure 34 shows the tree of the data model in Figure 26 on page 64 for the prepare purchase order process model in Figure 8 on page 31. This process model utilizes nodes of data classes purchase order *PO*, *Quote*, components *CP*, quote information *QI*, and quote details *QD* with *PO* representing the case object. Since data class *Request* links classes *PO* and *Quote*, it is also contained in the tree.

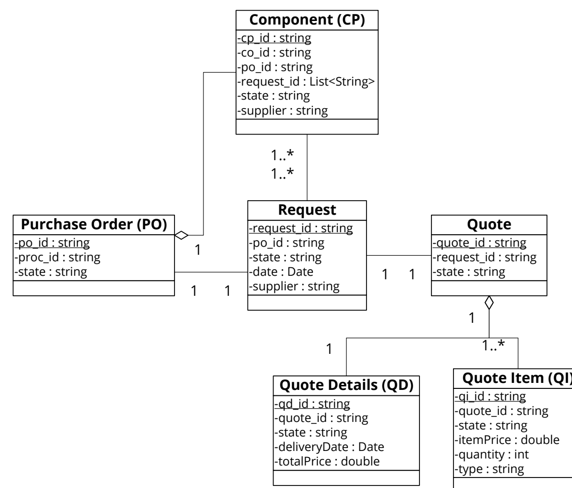


Figure 34: Tree of the data model in Figure 26 for the prepare purchase order process model (see Figure 8) that utilizes data classes purchase order *PO*, *Quote*, components *CP*, quote information *QI*, and quote details *QD*. Data class *Request* is also shown since it links the classes *PO* and *Quote*.

All process scenarios of one business process refer either to the same synchronized object life cycle or to *compatible* synchronized object life cycles. Compatible means that the OLCs are in subset or superset relation to each other such that they might be combined into a single synchronized object life cycle via state chart integration techniques [108, 235, 258, 328]. Several business processes may interact with each other, e. g., the build-to-order and delivery process presented in Section 2.4, corresponding human resource processes making sure the correct people are employed to execute the business process, internal finance and accounting processes, and internal logistics processes to, for instance, transport manufactured products to the warehouse, where it gets stocked.

The structure of interactions between business processes can be visualized with business process architectures [79, 88]. We assume that organization internal processes communicate and synchronize via implicit data dependencies only [88]. Communication to external partners is handled via message flow and process choreographies (see Section 4.5. Thereby, each partner may utilize different synchronized object life cycles in the corresponding process scenarios. However, these object life cycles need to be compatible to not contradict to a choreography's realizability [69, 264].

To allow explicit correlation of process models to process scenarios and these in turn to process models, we introduce two mapping functions: ρ_{PM} and ρ_{PS}. Thereby, BP denotes business processes within a specified scope – usually an organization or a division of an organization.

Definition 4.18 (Mapping Functions)**.**
Function $\rho_{PM} : PM \rightarrow PS$ maps each process model $pm \in PM$ to a process scenario $ps \in PS$ while function $\rho_{PS} : PS \rightarrow BP$ maps each process scenario $ps \in PS$ to a business process $bp \in BP$. ◂

Considering the running example in Section 2.4, the overall business process is the the build-to-order and delivery process from the computer retailer point of view. This business process consists of seven process scenarios that in turn each contain exactly one process model from the set of models represented through Figures 5 to 11. Assume, the purchase order preparation process in Figure 8 is referred to as pm. Then, $\rho_{PM}(pm)$ results in ps that in turn leads to the overall build-to-order and delivery process bp; i. e., $\rho_{PS}(ps) = bp$.

4.5 PROCESS CHOREOGRAPHIES

Interaction between different organizations is modeled via process choreographies. First, the organizations have to agree on a global data model which describes the information that might be exchanged. Thereby, messages transferred between different business processes, i. e., different participants, are explicitly modeled – called *message flow*. Each message flow contains a *message* that is transmitted and this message is referred to via a unique name that also represents the message type. A message flow may connect tasks of type *send* and tasks of type *receive* of different participants and has exactly one source and one target. Hence, it also shows the direction of message transmission. The process models of different participants may be of any abstraction level but need to contain at least all activities that are utilized for sending and receiving messages. This refers to the so-called public view as described in the public-to-private (P2P) approach. Thereby, local processing information are hidden from the other choreography participants. On an abstract level, these tasks may also be hidden such that the message flow is con-

nected to a subprocess or a process indicating that the message is sent or received somewhere within this subprocess or process. However, this representation is not sufficient for enactment as discussed in Chapter 8 and is therefore only used for illustration purposes in the scope of this thesis.

Furthermore, a process model representing (parts of) a business process may contain one additional modeling concept that is not included in Definition 4.10 on page 70. A gateway may also be of type *EVENT* referred to as *event-based gateway* such that $type_g : G \rightarrow \{XOR, AND, EVENT\}$ holds. On orchestration level, an event-based gateway acts as deferred exclusive decision – based on some received message, a path is chosen.

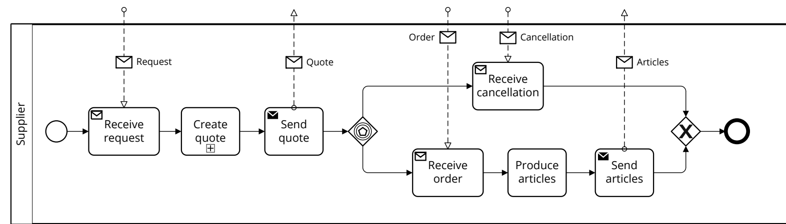


Figure 35: Order and delivery process from *supplier*'s point of view containing an *event-based gateway* to decide upon message retrieval whether the request is canceled (upper path if the cancellation message arrives) or whether it is changed into an order (lower path if the order message arrives).

For instance, considering Figure 35 that represents the build-to-order and delivery process from the supplier's point of view, a *Cancellation* message triggers the upper path while an *Order* message triggers the lower path. This extension to the gateway type function is not added to the generic process model definition because the event-based gateway is rather a specialty most often used in the context of process choreographies. Additionally, event-based gateways can be re-modeled using exclusive gateways as shown for the given example in Figure 16 on page 35.

The global data model only contains information required for interaction. Nevertheless, it must be compatible to each single local data model utilized in the participants' process models. Whether the local data models get adapted to fit the global one or whether the global one is specified to fit the local ones depends on the implementation and is out of scope for this chapter. Formally, we define a process choreography as follows.

Definition 4.19 (Process Choreography).
A *process choreography* $pc = (BP, \mathfrak{M}, dm)$ consists of a finite non-empty set BP of business processes which are connected via the message flow relation $\mathfrak{M} \subseteq (A_{bp_i} \cup pm_{bp_i}) \times (A_{bp_j} \cup pm_{bp_j})$, $bp_i \neq bp_j$ and a global data model dm which is compatible to all data models dm_k for all

business processes bp_k. Each message flow contains a message msg consisting of a unique *name*. ◂

PC denotes the set of process choreographies such that mapping function $\rho_{BP} : BP \rightarrow PC$ maps each business process bp to the process choreography pc utilizing it.

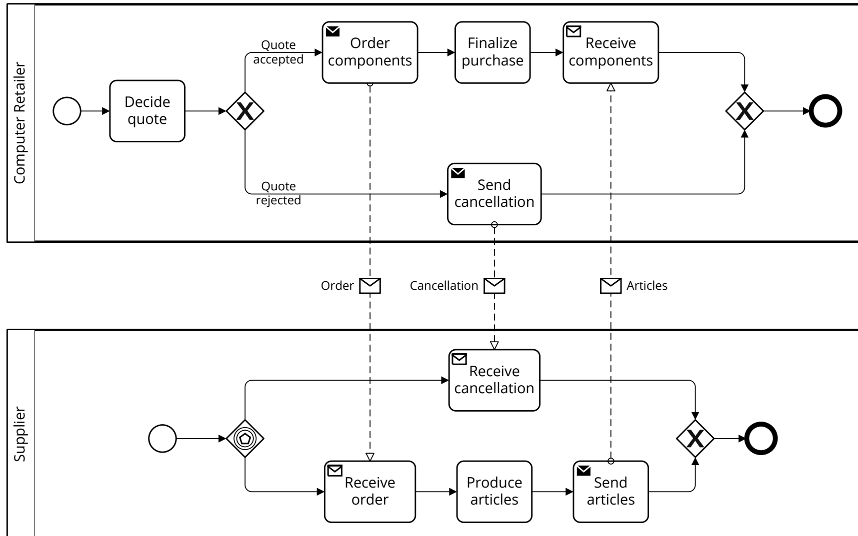


Figure 36: Process choreography showing the simplified interaction between the *Computer Retailer* and the Supplier after the retailer received a quote as response to the sent request for acquiring some articles that in turn refer to the components in the models of the running example in Section 2.4.

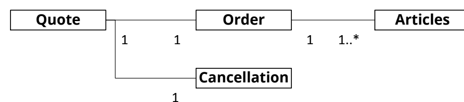


Figure 37: Global data model for process choreography in Figure 36.

Figure 37 visualizes a global data model for the process choreography shown in Figure 36. This choreography is extracted from the running example showing the simplified interaction between the *Computer Retailer* and the *Supplier* after the supplier answered the retailer's request with a quote. If the retailer accepts the quote, an *Order* is transmitted upon which the corresponding *Articles* are shipped to the retailer. In case of rejecting the quote, the retailer sends out a *Cancellation* message. Since the *Quote* links the different messages exchanged in this example, we added this to the data model as well. Though, for better readability, we omitted data nodes in the process models and data attributes in the classes of the data model. For compatibility considerations of the global data model to the local ones, a schema mapping to each local data model must exist. For instance, class *Order* maps to the local class *Purchase Order (PO)* of the computer retailer; the *Articles* map to the *Components (CP)* and the remaining two classes refer to content of

the local *Quote* class and are mapped accordingly. The messages flows connect *send* and *receive* activities; e. g., the *Cancellation* message is sent from activity *Send cancellation* on the retailer's side while it is received from activity *Receive cancellation* of the supplier. All message flows have exactly one source and one target activity.

Correctness of a process choreography, e. g., realizability, is important to ensure expected execution of the interactions and will be extensively discussed in Chapter 8 in the context of model-driven enactment. Interaction between partners is especially important for process execution to align each partner's processes, e. g., a logistics company may only pick-up the parcel if packaging was finished and the parcel got released. For organization internal processes, they are not utilized. In the scope of this thesis, process choreographies are only considered for the model-driven execution of business processes (see Chapter 8) because execution requires cross-organizational interactions as discussed above. The remaining concepts of this thesis are only discussed for process orchestrations as these focus on intra-organizational actions. For instance, data nodes are only reasonably extracted for single process models (Chapter 5). Additionally, task enablement is also not affected in general since messages are treated as data objects and the task gets enabled from the data point of view after retrieval of the message, i. e., upon existence of the data object representing the message. Details about formal semantics will follow in Section 4.7.

4.6 CONCEPTUAL MODEL

Figure 38 visualizes the formal concepts of process and data integration introduced above as model view in a UML class diagram. Thereby, the white colored classes represent traditional, activity-driven process modeling concepts while the gray colored classes represent data-related concepts not present in traditional process modeling. Although being data-related, the concepts of a *data node*, *data state*, *data store*, *data flow edge*, *persistence relation*, and *message flow edge* are correlated to the process modeling concepts, since they already exist in various process description languages including the industry standard Business Process Model and Notation (BPMN) for several years.

Choosing the class *process model*, the figure visualizes that it consists of any positive natural number of *control flow nodes* and any natural number including zero of *data nodes*, *data stores*, *control flow edges*, and *data flow edges* as defined in Definition 4.10. Further, each process model is associated to exactly one *synchronized object life cycle* and belongs to exactly one *process scenario* as stated in the context of Definition 4.8 and as defined in Definition 4.16 respectively. Analogously, each *data class* is part of one data model (cf. Definition 4.5), is associated to any number of *process scenarios* (cf. Definition 4.16) and to exactly one *OLC* (cf. Definition 3.8), refers to any number of *data nodes* (cf. Definition 4.2),

Figure 38: Conceptual model visualizing process and data integration, where the white elements represent traditional, activity-driven modeling concepts and the gray elements represent new integration-related modeling concepts.

Figure 39: Visualization of mapping functions between the concepts from Figure 38 as well mapping to the instance view. The conceptual relationships between the classes is indicated by gray connectors while the mappings are shown with solid black arrows.

and consists of at least two further *data attributes*, the unique name and the primary key, and exactly one *data state* (cf. Definition 4.1). The remaining concepts and associations as well as generalizations refer to various definitions mainly given in this chapter but also in Chapter 3 and will not be discussed in detail here.

The concepts of *data objects*, *active data states*, *process instances*, *process instance states*, and *data views* are not presented in this conceptual model since they are part of the instance view instead of the model view. A *data object* is the instance representation of a data class or data node respectively in n:1 relationships. A *process instance* is the instance representation of a process model in an 1:1 relationship. *Process instance states* are directly correlated to process instances in n:1 relationships but each process instance possesses exactly one specific state at all points in time. *Active data states* relate in n:1 relationships to the data states of a data class. *Data views* relate in 1:n relationships to values of attributes of data classes.

In addition to the conceptual model, Figure 39 presents a visualization of the mapping functions between various concepts as stated in the formalization including the instance view. The class diagram in Figure 38 got reduced to the concepts participating in mapping functions either as source or as target, e. g., a *data class* is the target of mapping functions η, φ_D, and φ_O to map an *object life cycle*, a *data node*, and a *data object* to the corresponding data class respectively. The mapping functions are presented via solid black arrows between the concept classes. The class diagram relationships as presented in Figure 38 are indicated with gray colored connectors.

4.7 FORMAL SEMANTICS

After introducing the abstract syntax for process and data modeling, we proceed with formal semantics. The generic process model (cf. Definition 4.10) builds the basis for various process description languages currently used in industry as, for instance, BPMN [243], event-driven process chains (EPCs) [157], and activity diagrams (ADs) [244]. These languages usually lack formal semantics and analysis techniques to describe the execution and to check, amongst others, behavioral consistency. Following the concept of *translational semantics* [225, 319], we utilize Petri nets [253], a well established formalism to verify various properties of process models and to describe their execution semantics [233, 331]. Most existing process description languages can be transformed into Petri nets dealing with the tradeoff of information loss for complex and language-specific structures as the non-interrupting intermediate events in BPMN, for instance; [190] gives an overview.

One such mapping was introduced by Dijkman et al. [80] for BPMN 1.0 [242] whose modeling concepts with respect to control flow are a superset of the ones presented in Definition 4.10. Thus, the rules can

be applied to other process description languages as well, especially a generic one as defined. The consideration of data is completely omitted.

Petri net Mapping for Process Orchestrations

Petri net mapping for orchestrations

We utilize this mapping as basis for the control flow mapping and extend it by a set of eleven rules to cover the data flow as well in process orchestrations. Generally, a control flow node is enabled if the control flow reaches this node and – in case of activities – the required execution information in terms of data nodes (objects) in respective states exists. An activity is properly terminated if the expected results are achieved, i. e., the data nodes (objects) written. Figure 40 summarizes the rules required for data flow coverage. In fact, the combination of rules given in [80] and the ones given in Figure 40 transforms a process model into its Petri net representation showing the execution semantics and allowing further behavioral correctness checks as the weak conformance check introduced in Chapter 6.

Application of the rule set from Figure 40 requires some assumptions (RSA) to hold:

(RSA-1) The process model is a subset with respect to the generic one specified in Definition 4.10 such that $pm = (N, D, Q, \mathfrak{C}, \mathfrak{F}, type_g, \mu, DCF)$ with $DCF_{pm} = (\xi)$.

(RSA-2) The data annotations specify the information required to execute an activity (read) and the information expected to exist after termination (write).

(RSA-3) Multiple data nodes with the same name read or written by one activity are disjunctive while data nodes with different names are conjunctive.

(RSA-4) XOR decisions are based on the data results of the activity directly preceding the XOR gateway.

(RSA-5) All data nodes with the corresponding data states required for some view on the process model are annotated to the activities, but the data annotation does not need to be complete in terms of comprising all probably occurring data state transitions during process execution.

(RSA-6) An activity is enabled, if and only if the control flow reaches that activity and all data objects read by the activity exist in the states specified by the data nodes in the process model.

(RSA-7) Concurrent read of a data node is allowed, whereas concurrent write or a mixture of concurrent read and write are forbidden.

The mapping rules from Figure 40 can be distinguished into three categories: control flow mapping (rules 1 and 8), data flow mapping (rules 2 to 7), and concurrency handling using semaphores (rules 9 to 11). For scenarios, where assumption RSA-7 does not hold, i. e., where concurrent writes shall be allowed, the rules of the last categories shall be ignored. In these cases, data integrity cannot be guaranteed anymore as, for instance, the risk of lost updates exists. Relaxing assump-

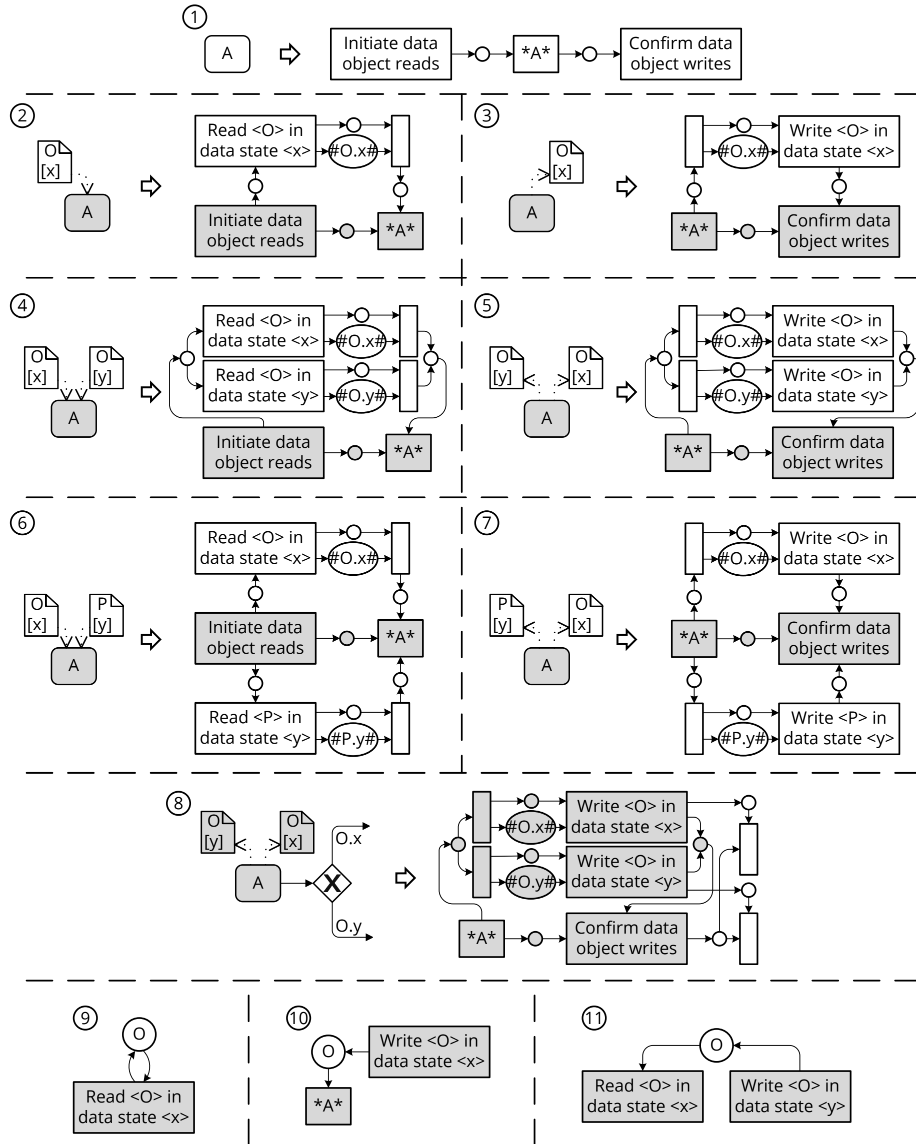


Figure 40: Rules to map data dependencies/constraints of a process orchestration model to a Petri net.

tion RSA-3 by allowing further relations between the read respectively written data nodes as, for instance, with the input and output sets of BPMN [243] would require a re-calibration of rules 4 to 7. Rules 4 and 5 would need to be applied to data nodes in disjunctive relation independent from their names while rules 6 and 7 would need to be applied to conjunctive data nodes, both with respect to the additionally given specifications. However, in this chapter, we base the mapping on explicit process model information only. In each rule from Figure 40, the gray modeling constructs are helpers setting the context for the white modeling constructs that are tackled or affected by a rule.

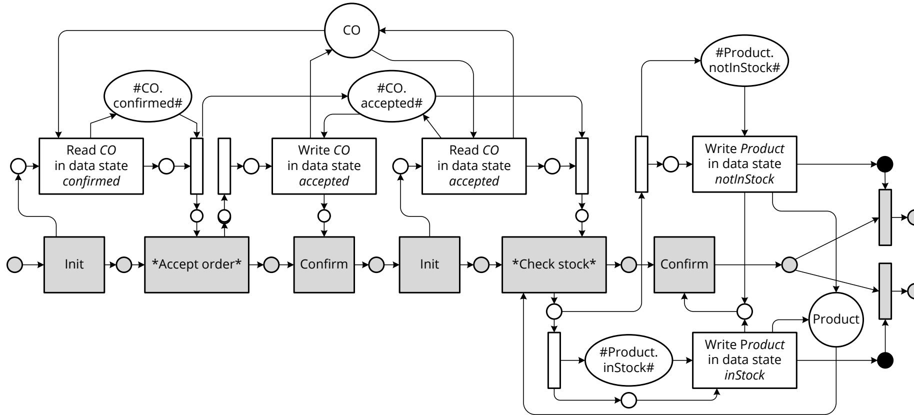


Figure 41: Petri net excerpt with data dependency (constraints) coverage for activities *Accept order* and *Check stock* of the process model in Figure 7.

Figure 41 shows an excerpt of a Petri net comprising control and data flow mapped from the process model in Figure 7. The excerpt comprises the activities *Accept order* and *Check stock* as well as the subsequent XOR gateway. The gray colored modeling constructs represent the control flow. Rule 1 extends the mapping of one activity to one transition from [80] to a set of three transitions to initiate data reads (first transition) and to confirm and synchronize data writes (third transition). In Figure 41, both transition labels are shortened to *Init* respectively *Confirm*. The second transition comprises the actual work performed during activity execution indicated by two asterisks enclosing the activity label. Further, choices at data-based XOR gateways must be handled. While these are non-deterministic from control flow point of view, they get deterministic if the specified data specifications are complete (cf. SCC-8). Therefore, rule 8 introduces XOR split determinism based on data conditions annotated to control flow edges. In Figure 41, the corresponding places are marked black. Based on the states of data nodes either path is taken. If *Product* is data state *not in stock*, a purchase order must be created which is later sent to the chosen supplier. If a *Product* is already in data state *in stock*, the stock can directly be reserved from the inventory.

The white modeling constructs represent the data access including the semaphore places being the places labeled with *CO* or *Product* respectively. Rule 2 describes the read of a single data node with a specific data state. The labeled activity prepares the actual read, i. e., it checks whether the mentioned data node (object)[3] is in the respective data state and sets tokens in both succeeding places. The small, unlabeled one ensures deterministic behavior of the net with respect to data reading in complex scenarios; e. g., if multiple activities write the same state of one data node (object), this place allows to differentiate which write is the actual one (cf. activities *Stock-up inventory* and *Check stock* in Figure 7 for data class *Product* and data state *in stock*). The large, labeled place represents the existence of the corresponding data node (object) in the required state. The label consists of the data class name and the required data state separated by a dot and surrounded by dashes #. Rule 4 describes the read of one data node in one out of x data states in disjunctive relation. For visualization, the case of one out of two is presented in Figure 40 but the remaining options are added in parallel to the presented two and get also synchronized by the white places being in the postset of transition *Initiate data object reads* respectively being in the preset of transition $*A*$. Rule 6 describes the read of independent data nodes in conjunctive relation. For visualization, the case of two independent data nodes is shown. For each further data node, a new place is added to the postset of transition *Initiate data object reads* and another place is added to the preset of transition $*A*$ with the corresponding nodes between these places. Analogously, rules 3, 5, and 7 represent the writing procedures for a single data node, one out of x data states of one data node, and multiple independent data nodes respectively; the visualization in Figure 40 shows the case for two data nodes for rules 5 and 7.

A semaphore place is involved differently based on the type of access to a data node by an activity. We distinguish read (rule 9), write (rule 10), and modifying (rule 11) access. For each data class, one of these places is created and reused for each access to the corresponding data node (object). In the Petri net, a semaphore is visualized by a place labeled with the corresponding data class' name.

During the mapping process, places and labeled transitions, except for the *initiate read* and *confirm write* transitions, with identical labels are identical and are therefore merged into single places or activities respectively. The presented rules guarantee that the resulting Petri net satisfies the soundness property [331] by construction under the assumptions that no concurrent data modifications take place and that the original process model is deadlock and lifelock free from the control flow perspective. Each of the fragments replacing an activity or a data node following rules 1 to 7 is a single entry single exit fragment [150, 151, 358] and sound such that their composition also remains

3 At design-time, it is the data node. At run-time, it is the data object.

sound. The soundness property also holds for the semaphore rules 9 to 11, if no concurrent data access takes place (cf. RSA-7), because there is always a transition consuming the token and either the same transition or one succeeding it shortly puts the token back to the semaphore place without influencing the control flow. With respect to our assumption that data conditions assigned to control flow edges are non-blocking, we can safely reason that rule 8 does not induce deadlocks into the net. Finally, the mapping from [80] produces sound Petri nets and therefore, the resulting Petri net after applying control flow and data flow rules is sound by construction.

The formal semantics of a process orchestration model follows Petri net semantics [233] based on the mapping introduced above. Next, we will summarize these formal semantics for process orchestrations after parts have been discussed before in terms of assumptions, e. g., RSA-3.

Formal Semantics for Process Orchestrations

Orchestration semantics

To describe the formal semantics for process orchestrations, we utilize Figure 41 which presents Petri net representations of activities *Accept order* and *Check stock* of the process model in Section 2.4 detailing the *process order subprocess* (Figure 7). The marking of the Petri net can be transformed into marked control flow and data flow edges in the process model. Such marking in the process model is visualized in Figure 42 for the case that activity *Check stock* with state *not in stock* of data class *Product* as result and the succeeding XOR gateway have been executed. We differentiate enablement of a control flow node from control flow and from data flow point of view based on the respecting markings as introduced in Section 4.3.

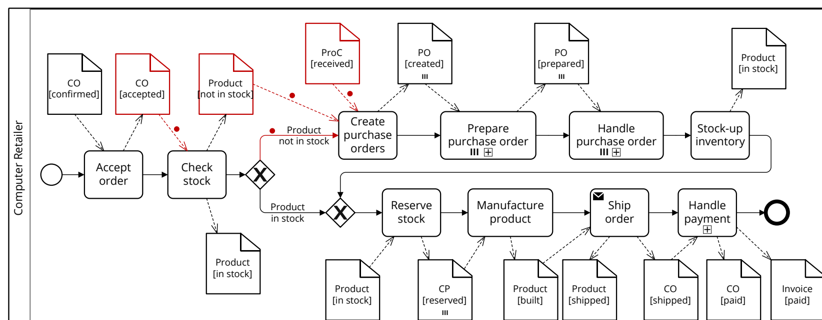


Figure 42: Marking of the process model detailing the process order subprocess in Section 2.4 case that activity *Check stock* with state *not in stock* of data class *Product* as result and the succeeding XOR gateway have been executed

From control flow point of view, a control flow node is enabled, if all appropriate control flow edges targeting that node are marked. Termination of execution of a control flow node changes the edge markings such that all incoming, marked control flow edges get unmarked and all appropriate outgoing edges get marked. In case of events, activities,

AND gateways, and XOR joins, all outgoing control flow edges are appropriate. In case of XOR splits, one out of probably multiple control flow edges is appropriate. The appropriate one is chosen via conditions annotated to the control flow edges and checked against current process data (cf. ξ in Definition 4.11).

From data flow point of view, a control flow node is enabled, if all appropriate data flow edges targeting that node are marked. Termination of execution of a control flow node changes the edge markings such that all incoming, marked data flow edges representing modifying access get unmarked and all appropriate outgoing edges get marked while markings remain for read only accesses. Appropriateness is defined via conjunctions of different data classes and disjunctions of equal data classes (cf. RSA-3). Considering send and receive tasks in process orchestrations, we handle them alike the other tasks and assume incoming messages to be input data nodes and outgoing messages to be output data nodes such that the content of incoming messages must exist in the orchestration context before task enablement and such that the sent message is one result upon task termination. Alternatively, messages can be handled explicitly as we discuss in detail in the context of a *Petri net mapping and formal semantics for process choreographies*.

Putting both views together, generally, a control flow node is enabled, if it is enabled from control flow and data flow point of view. With respect to Petri net semantics [233], activities concurrently enabled in the process model may be executed in parallel. But as discussed in Section 3.3, we assume trace semantics in the scope of this thesis meaning that concurrently enabled activities are started consecutively. However, in contrast to transitions in Petri nets, activities in process models may take some time until termination such that multiple activities may be in state running (cf. Figure 30) at the same time referred to as *interleaving semantics*. Therefore, data access requires explicit handling as two interleavingly executed activities may try to access the same data object. Write access to a data object from an activity is transactional, i. e., concurrent *write* or *modifying accesses* are not allowed, whereas concurrent *read only accesses* are allowed. This transactional property of process model execution semantics is ensured by the usage of semaphore places in the Petri net (places labeled with the data class' name; see above).

Petri net Mapping and Formal Semantics for Process Choreographies

Choreography semantics

After abstracting from sending and receiving messages in the context of process orchestrations, we now explicitly discuss the corresponding choreography semantics. While process choreographies utilize the orchestration semantics for the local process models of the utilized business processes, handling the message flow dependencies require two additions to the above introduced Petri net mapping due to the send respectively receive task. The corresponding mapping rules 12 and 13 are visualized in Figure 43. Again, the gray modeling constructs are

Choreography Petri net mapping

helpers setting the context for the white modeling constructs that are tackled or affected by a rule.

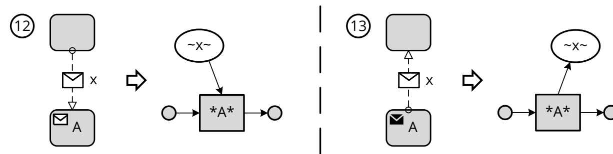
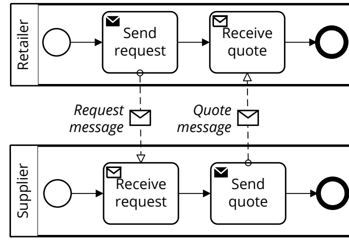


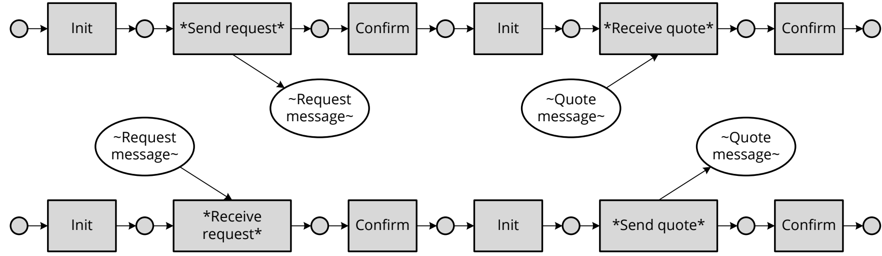
Figure 43: Rules to map the message flow dependencies of a process choreography to a Petri net as extension to the orchestration rules visualized in Figure 40.

Similarly to data nodes, *messages* are represented by means of places in a Petri net. An activity sending a message puts a token into the corresponding place (rule 12) while an activity receiving a message can only get enabled if the required message has been sent before by some other participant such that the message place added to the input set is marked (rule 13), i. e., that message can be received. The additionally involved event-driven gateway – the deferred choice – does not influence the Petri net mapping. Control flow constructs preceding an event-driven gateway are handled as specified in Figure 40 – activities follow rules 1 to 7 and an XOR split follows rule 80 – and tasks of type *send* or *receive* are handled as just discussed and visualized in Figure 43.

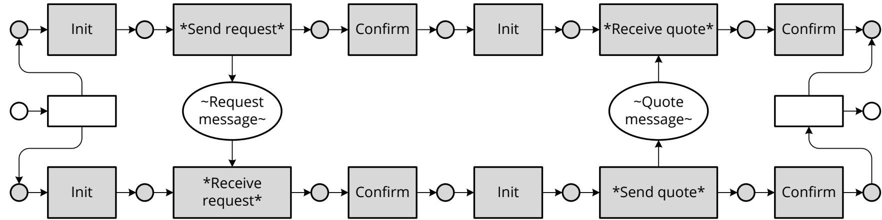
For each participant of the process choreography, such Petri net is derived with the messages denoting *communication interfaces*. The resulting Petri nets get combined via communication interface mapping. Further, a single start place is added. This place is connected to each previous start place of the single Petri nets via an additionally added transition. Analogously, the single end places get connected towards a single end place in the combined Petri net. Figure 44 shows an example derived from the running example in Section 2.4 comprising the request response part between the computer retailer (*Retailer* in Figure 44a) and the *Supplier*. Both process models of the process choreography are mapped to the corresponding Petri nets using the orchestration rules for the gray modeling constructs and the choreography rules for the white modeling constructs (see Figure 44b). In the last step, the communication interfaces (white labeled places) get combined via label matching and the start and end places get connected to single places each (see Figure 44c for the result). In the given example, each sent message is received and each received message was sent in the context of the choreography. Therefore, each message place' preset as well as postset is of size 1. In this case, the process choreography is structurally compatible [67] and the according Petri net describes the execution semantics. In other cases, i. e., if a sent message message is not received respectively a received message is not sent within the process choreography, the process choreography is not structurally compatible and thus, the execution semantics not entirely clear, since some information is hidden and not represented in the Petri net. Details on process



(a) Process choreography.



(b) Petri net mapping based on the orchestration rules 1 to 11 (grey modeling constructs) as well as the choreography rules 12 and 13 (white modeling constructs).



(c) Combination of mapped Petri nets where the white modeling constructs are utilized for the combination as well as single source and sink nodes.

Figure 44: Example for Petri net mapping of a process choreography that is derived from the running example in Section 2.4.

choreography correctness including structural compatibility will follow in Section 8.5.

4.8 RELATED WORK

The increasing interest in the development of process models for execution has shifted the focus from control flow to data flow perspective leading to integrated scenarios providing control as well as data flow views. The aim to integrate business information (data flow) and business processes (control flow) tries to overcome the limitation of traditional workflow management systems that lack a comprehensive data-oriented view. One step in this regard are object-centric processes [54, 173, 231, 344, 384] that connect data classes with the control flow of process models by specifying object life cycles initially described in [237] and refined in [54]. [175] introduces the essential requirements of this modeling paradigm. Object-centric process models (OCPs) connect data classes with the control flow of process models by specifying

object life cycles that represent data dependencies and based thereon, the order of task execution. [176, 292] present an approach which connects OLCs with process models by determining commonalities between both representations and transforming one into the other. Covering one direction of the integration, [186] derives OLCs from process models considering synchronization between actions (state transitions). [361] also stresses the importance of handling inter-dependencies between different data classes for process execution they refer to as coupling that corresponds to typed synchronization edges introduced in this chapter. While we specify these inter-dependencies explicitly in the object life cycles, the authors predict probable couplings between implementations of data objects based on workflow patterns [345]. In [232], a rule-based approach is described; it allows to connect control flow with data flow and, thus, to automatically create data-driven executable process models.

Similarly to the mentioned approaches, we concentrate on integrated scenarios incorporating process models and object life cycles. In contrast to object-centric approaches, we utilize an activity-centric process model as basis and incorporate control flow and data handling there instead of introducing a new modeling paradigm which requires process engineers and participants to learn new concepts instead of the ones they already know using current industry standards as, for instance, BPMN. Additionally, we only model the process and set the synchronized object life cycles as reference for the overall process scenario describing the data manipulations to be utilized by process models within the scenario.

Petri nets is a well established formalism to describe formal semantics [233] of informal process description languages as, for instance, BPMN, EPCs, and the business process execution language (BPEL). Following the concept of translational semantics [225, 319], various Petri net mappings have been specified; e. g., from BPMN 1.0 [80], EPCs [333], BPEL [137, 247], or workflow task structures [339]. [190] provides a survey about mappings of various process description languages to Petri nets. Thereby, the mappings (and source process description languages) focus on control flow constructs while we utilize one of them as basis and extend it with generic rules of data flow mapping. The mapping chosen for extension, [80], specifies a mapping for all control flow concepts defined in Definition 4.10 and thus, can be utilized for a generic mapping to a Petri net.

Besides Petri nets, further formalisms exist that have been used to specify formal semantics of informal process description languages. For instance, [376] describes formal semantics for BPMN by mapping BPMN diagrams to Communicating Sequential Processes (CSP) [139, 284] while [262, 263] describe formal semantics for generic process graphs using the pi-calculus [226, 227], a non-graph-based formalism. We decided on a graph representation of the formal semantics since

visual representations in terms of graphs are easier to understand than non-visual ones [179, 246] if they are sufficiently expressive as Petri nets, for instance, are for our use case. An CSP model consists of CSP processes (representing tasks) and events (representing the control flow). Compared to Petri nets, CSP models do not preserve the structure of the source process model and get comparably large and complex since each task is represented by one CSP process. Furthermore, formal semantics for BPEL have been specified with process algebra [44] and finite state processes (FSP) [107]. In year 2006 already, [330] lists more than 100 papers discussing formalizations for BPEL while since then still further formalizations have been published. However, only few papers describe full mappings. Most rather describe very specific aspects and develop a formalization for this part only.

There do also exist some approaches to represent data objects in Petri nets. [14] introduces a mapping from BPMN to Petri nets based on six rules. Compared to our mapping, the authors duplicate transitions in the Petri net to specify each pre- and postconditions of the corresponding activity separately and did not address challenges with respect to parallel data access. Further, they considered data access to be passive while we consider it to be active allowing to control the data access. [314] introduces a mapping of BPMN to Petri nets, where the control flow mapping bases on the Dijkman et al. mapping [80] and additional data mapping rules are introduced. Thereby, the authors abstract from data states and utilize data classes only. [304, 325] introduce WFD-nets, which are workflow nets extended with data capabilities. These WFD-nets could directly be used to represent business processes with the disadvantage that data flow cannot be visualized graphically and states of data objects are not regarded.

Mappings of an informal process description to a formalism is only one application case of mapping. Usually, one may also map one informal language into another one. For instance, many process models have been modeled using EPCs. Though, in recent years, the focus changed to BPMN. However, process models created in the “old” language shall not need to be wasted, but rather transformed into the new standard modeling language. Such transformation from EPCs to BPMN is described in [327]. Moreover, at some time BPMN did not contain a proper execution semantics but was already used for process descriptions since it is comparably easy adoptable by business users. In addition, BPEL was the standard for process execution. Thus, process design was done in BPMN, the resulting model was mapped to BPEL such that this result now could be executed with information system support. [248] describes one such mapping from BPMN to BPEL. However, since both process description languages face a conceptual mismatch [266], mapping (in both directions) is limited [367]. Likewise BPEL, Yet Another Workflow Language (YAWL) could be used for process execution. [71] specifies a corresponding mapping from BPMN.

While most formalization approaches focus on specific languages to map from and to map to, [202] abstracts from specific languages and provides strategies to generally map block-oriented (e. g., BPEL) and graph-oriented (e. g., BPMN, EPC) process description languages into each other.

4.9 CONCLUSION

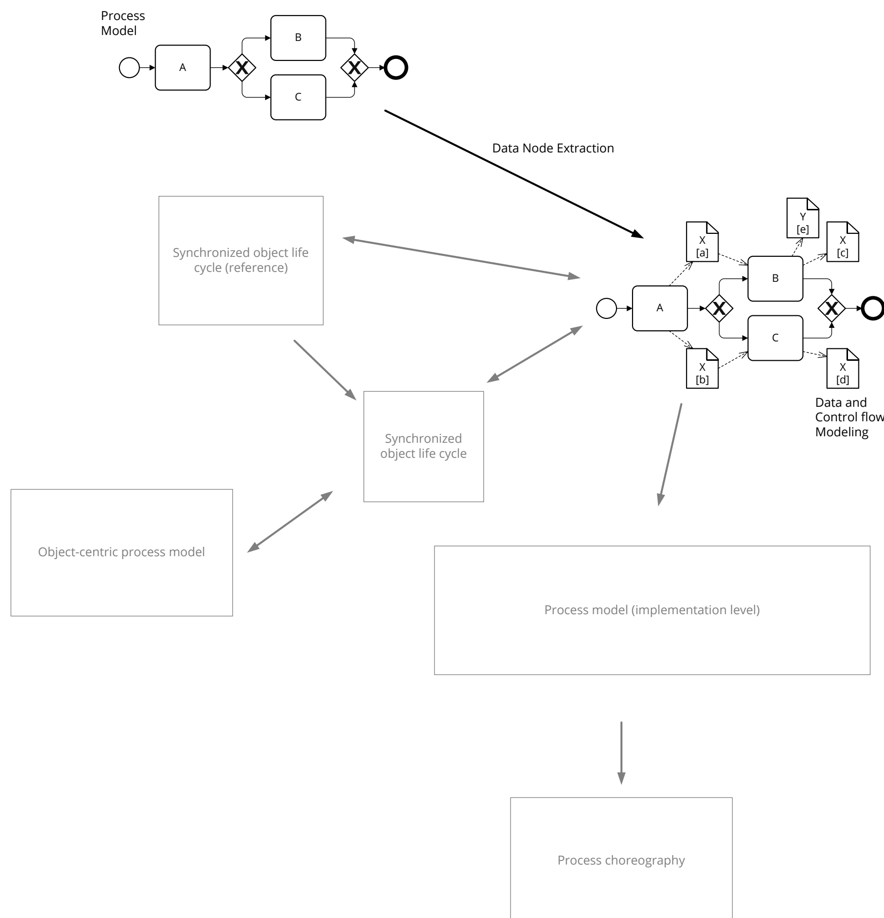
In this chapter, we introduced the formal framework to integrate the process in terms of control flow and the data perspectives as basis for utilization in multiple areas of BPM. In the scope of this thesis, the utilization is focused on process execution but the framework also contains details required in areas such as business process monitoring and business process model abstraction (out of scope of this thesis) as well as process correction and process modeling and design (comprised by this thesis). The framework contains a data side distinguishing the type and the instance level borrowed from object orientation.

The type level is represented by a data model consisting of data classes. For each data class, there exists an object life cycle, a state transition diagram, representing the allowed data manipulations. The concept of synchronization edges covers dependencies between multiple states or transitions of different object life cycles. The instance level is represented by data objects at run-time and data nodes at design-time as run-time representatives. These data nodes are integrated with additionally required information into an activity-centric process model. For distinguishing process instances at run-time, we utilize the concept of instance data views providing a projection on the data values of the data objects. For providing the integrated view on processes and data, we utilize process scenarios, a formalism combining a process model and the corresponding synchronized object life cycle. We summarize the formal framework through a conceptual model that links the utilized modeling concepts and the mapping functions.

Making the process models executable, a formal semantics is required. We introduce these through a Petri net mapping for process orchestrations and process choreographies considering control flow and data flow equally as first class modeling concepts. Activities might only be executed, if both control flow and data flow allow enablement. The introduced Petri net mapping extends the widely accepted one from Dijkman et al. [80] with data consideration.

5 EXTRACTION OF DATA NODES AND THEIR STATES

This chapter is based on results published in [212].



THE IMPORTANCE OF DATA for business processes is well known; for instance, their execution and automation requires detailed data specification such that the corresponding process engine executing a business process knows about potential next tasks towards the business process goal at all times. However, in practice, this information is rarely modeled in business processes such that the majority of existing publicly available as well as organization internal process model collections mostly contain control flow information only. This, for instance, holds true for the SAP reference model [56] and the BIT process library from IBM [142]. The Signavio academic initiative (BPMAI) [34] provides some process models with explicitly modeled data nodes, although the majority of the process models cover control flow only.

Though, various new techniques utilizing data objects and their states emerged recently. These techniques cover real world challenges as, for instance, data consistency checking [281], consistency checking between process models and object life cycles (OLCs) [176, 364] (also see Chapter 6) – OLCs specify actions allowed to be performed on objects of a certain data class – and the synthesis of OLCs from process models [97, 98, 186, 292] (also see Chapter 7) as well as data-aware process model abstraction [131, 209] and model-driven data-aware process model execution, the main topic of this thesis (see Chapter 8). Researchers developing these and similar techniques require model collections which contain process models with explicitly modeled data information for validation, verification, and refinement of the techniques before they can be applied in practice. From this, we deduce the need to adapt existing process model collections such that the contained process models contain information about data nodes and their states.

Indeed, there exist real-world process models with some explicitly modeled data annotations from organizations which automate their processes with the help of information systems. But these process models are usually not shared with the research community such that we cannot utilize them for empirical research. In contrast to the opportunities that will be presented in Chapter 8, process models are often used for documentation only or as basis for manual process automation, where process experts and developers program the data layer for the given specific use case after process model elicitation. Thus, the process models are required to show rather control flow than data flow information leading to a separation of the process and data side as they are handled one after another. This separation may also result in inconsistencies which need to be corrected after identification (see Chapter 6). Providing explicit data information within these process models helps to foster the integration of process and data and reduces inconsistencies, because implicitly existing information is made explicit. Thereof, we deduce the second need to extend given process models: application in practice.

Usually, information about data is hidden in activity labels (if they are not anonymized as in the IBM collection). [204] describes that each activity label can be decomposed in up to three components: an action, a data object an action is performed upon, and a fragment providing further details (e. g., locations, resources, or regulations). On process model level, the data objects are represented by data nodes (cf. Section 4.1). For instance, the activity labels *Ship order to customer* and *Send invoice via email* encode the information that data objects of classes *order* and *invoice* respectively are processed in the corresponding activity. The actions performed are *ship* and *send* while the additional fragments provide insights about the additional resource involved and the regulation demanding a specific communication channel. With respect to empirical research from Mendling et al. [204], the majority of all activ-

ity labels conform to this structure. In the set of process models used by the authors, the SAP reference model [56], 94% of all activity labels contain at least an action and the corresponding data object.

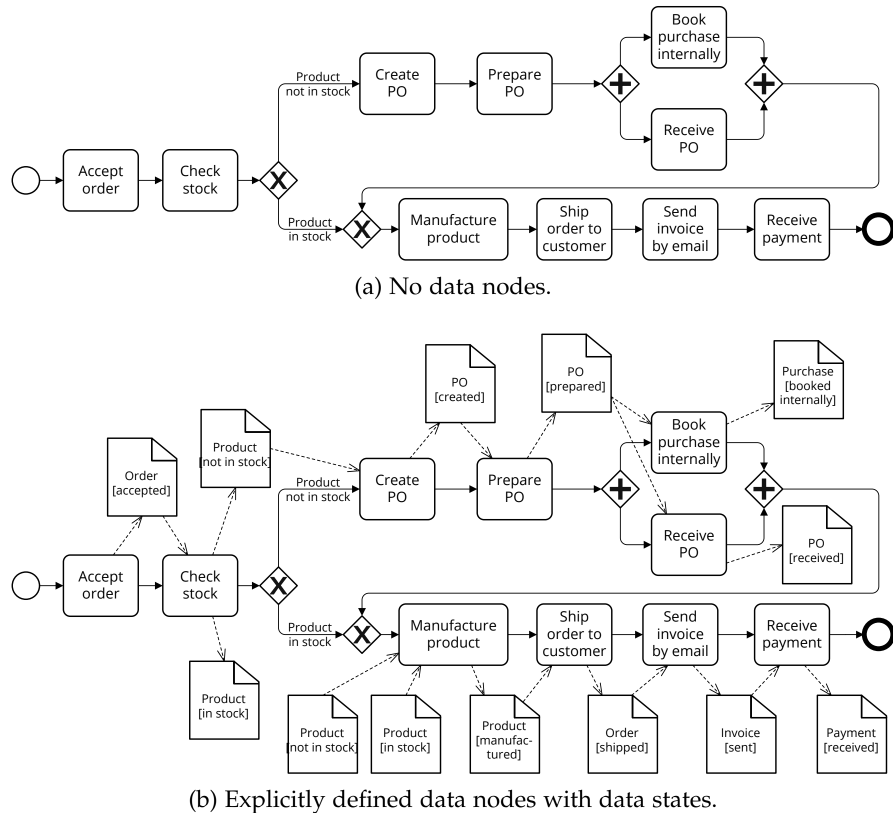
This chapter introduces a set of algorithms to analyze process models and make implicitly given data information explicit. These algorithms can be generally applied to most process models from various process description languages, because only activities, gateways, source and sink nodes, and control flow edges are used as input for data information extraction. The corresponding subset of the process model defined in Definition 4.10 is presented by tuple $pm = (N, D, Q, \mathfrak{C}, \mathfrak{F}, type_g, \mu, DCF)$ with $DCF = (\xi)$. Further modeling constructs like intermediate events and message flows as well as gateways other than XOR and AND are not supported in most process description languages (and would thus prevent generalization). Additionally, they and other non-generic constructs are rarely used in business process modeling [171, 389]. Basically, for each source process model fulfilling the mentioned minimum requirements, i. e., the one which shall be enriched with data nodes and data states, a generic process model consisting of the named modeling constructs can be created independently from the process description language. The created – also generic – process model provides the researcher or stakeholder with explicitly modeled data nodes and data states. Additionally, the algorithms can easily be adapted to be specifically tailored for a chosen process description language, e. g., Business Process Model and Notation (BPMN) or event-driven process chains (EPCs), utilizing all information provided by their modeling constructs to improve the extraction quality.

Section 5.1 introduces how to automatically transform a generic process model without data modeling into a process model with data nodes and their data states explicitly defined as shown in Figure 45. Afterwards, Section 5.2 discusses two extensions to the generic algorithms to incorporate the additional modeling constructs of BPMN and EPCs respectively. For both sections, the resulting process models are expected to be on the operational level [370], i. e., the process model describes the relationships between the activities and their input and output requirements in terms of data nodes with data states.

Figure 45a shows an extract of the order and delivery process from Section 2.4 focusing on the order processing. The process model aligns with Definition 4.10. The data nodes shown in Figure 45b have been extracted from the activity labels and the order of activities is provided by the process model in Figure 45a.

5.1 EXTRACTION ALGORITHMS FOR GENERIC PROCESS MODELS

Given a generic process model $pm = (N, D, Q, \mathfrak{C}, \mathfrak{F}, type_g, \mu, DCF)$ with $DCF = (\xi)$, we additionally require five labeling assumptions (LA) to hold to ensure convenient extraction results:



(a) No data nodes.

(b) Explicitly defined data nodes with data states.

Figure 45: Data extraction – before-after-comparison.

(LA-1) outgoing control flow edges of XOR splits are mandatorily labeled with a data condition ξ (see SCC-3 in page 39),

(LA-2) labeling is done homogeneously, i. e., the same node (object) is always referenced by the same phrase,

(LA-3) the object an action is performed upon and the action itself are present in each activity label although either of them may be missing in real world scenarios with respect to [182],

(LA-4) each activity performs exactly one action on possibly multiple objects, i. e., *create invoice and delivery notice* is an allowed label while *create invoice and print delivery notice* is not due to the second verb, and

(LA-5) all activity labels follow the verb-object-style labeling [182] (subset of the grammar introduced in Definition 3.3).

We choose the verb-object-labeling-style since it is widely used and also widely accepted as good modeling style. An activity shall rather be labeled *analyze order* than *order analysis* to explicitly and unambiguously show the work comprised by the corresponding activity. However, many activity labels violating assumption LA-5 can be transformed into the appropriate style using the techniques introduced from Leopold et al. [183]. The authors provide means to identify the used labeling style and to transform most of them into a verb-object-style for English language activity labels. Subsequently, these techniques could

be performed as preprocessing step to the extraction process, which comprises the following main steps:

1. Analyze the activity labels to determine the action and the data node (object),
2. determine and specify the activities' output data nodes, and
3. determine and specify the activity input data nodes.

General extraction information

The extraction algorithms – for output as well as input data nodes – covered by steps 2 and 3 follow four assumptions (EAA):

(EAA-1) an activity preceding a split or fork influences the activities within the block,

(EAA-2) an activity succeeding a join or merge is influenced from this block,

(EAA-3) each activity requires at least one output data node, and

(EAA-4) each activity that is not directly succeeding the start node requires at least one input data node.

The given order of algorithm application is essential for the extraction results, because the data input specification utilizes information determined during data output specification. Both algorithms analyze the process model towards *patterns* that target the structure of the model in terms of precedence and successorship relations between modeling constructs. Thereby, a pattern comprises two nodes – at least one being an activity – and the ordering relation (precedence, successorship) between them; e. g., an activity precedes an other activity, an activity succeeds an AND fork, or an activity precedes an end node. A complete set of patterns comprises all combinations of precedence and successorship of an activity with respect to an activity, an XOR (AND) gateway, or an event (precedence relation for an end event and successorship relation for a start event). Each pattern can unambiguously be grouped towards its influence on the input or output data nodes of the activity comprised by the pattern. Precedence relations influence the data output specification (step 2) while successorship relations influence the data input specification (step 3). Next, we discuss the label analysis and data extraction algorithms in detail.

1—Label analysis. The label analysis consists of three main steps (LAS) that are detailed in Algorithm 1:

(LAS-1) part-of-speech tagging of activity labels (see line 2),

(LAS-2) remove phrases not required for data node extraction (see lines 3 and 4), and

(LAS-3) correct tagging based on verb-object-style assumption (see lines 5 to 17).

These three steps are performed for each label of an activity. First, an activity label is parsed to tag each contained word with respect to its grammatical function as, for instance, verb, noun, adjective, or preposition (LAS-1; see line 2 in Algorithm 1); $\mu(a)$ returns the label of activity a. This is referred to as part-of-speech tagging. We utilize the *Stanford*

Algorithm 1 Analysis of activity labels for data node recognition.

```
1:  for all a ∈ A do
2:    a.label ← tag(μ(a));
3:    a.label ← removePrepPhrase(μ(a));
4:    a.label ← removeDeterminer(μ(a));
5:    if type(μ(a).getWord(0))! = VB then
6:      setType(μ(a).getWord(0), VB);
7:    end if
8:    noun ← new Boolean(false);
9:    for all word ∈ μ(a) do
10:     if type(word = NN || NNS) then
11:       noun ← true;
12:       break();
13:     end if
14:   end for
15:   if !noun then
16:     a.label ← tagNoun(μ(a));
17:   end if
18: end for
```

Log-linear Part-of-Speech Tagger [323, 324] to assign words their grammatical function that in turn uses the *Penn Treebank English POS tag list* [298] for distinguishing and visualizing the grammatical functions. A complete list of grammatical functions considered in this thesis – a subset of [298] – is given in Definition 3.3 about natural language grammar on page 40. Then, the additional information, which is represented as prepositional phrase if existing, gets removed from all activity labels because it has no influence on data node and data state retrieval (see line 3). A prepositional phrase starts with a subordinating conjunction, a preposition, or *to* indicated by tags *IN* and *TO* according to Definition 3.3. Similarly, determiner, e.g., *the*, *a*, and *these*, get removed from the activity label (see line 4). After reducing the activity labels towards the required phrases for data node extraction (LAS-2), corrections need to be applied to the part-of-speech tagging (LAS-3). Some words in English language may be used in different grammatical functions, e.g., *ship* as noun or verb, and therefore sometimes get tagged the wrong way. To handle this issue, we utilize the verb-object-style assumption LA-5 and verify the tagging whether there exists in the label a verb (see line 5) followed by a noun phrase (see lines 9 to 14). If the verification fails, the tagging needs to be adjusted either manually or automatically. As the failure rate is comparably low (fixed amount of words usable in different grammatical functions), manual correction would be feasible. However, usually, similar labels are affected by mis-tagging in different process models, e.g., *ship products* where ship gets tagged as noun although used as verb. The labeling style assumption LA-5 strongly limits the potential structures of an activity label. Therefore, we provide automatic means to correct the unverified tags. The first word of the activity label must be a verb with respect to the verb-object-style and thus, it needs to be tagged that way. If the first word is not tagged as verb, the tag is changed accordingly (see line 6). Additionally, the verification checks for existence of a noun tag after the verb. A fail

results in tagging all words except the first one – the verb – that might be classified as noun accordingly (see lines 15 to 17).

Considering activity *Ship order to customer* in Figure 45a, the activity label after tagging is *Ship_VB order_NN* stating that *ship* is a verb in third person singular and *order* is a noun in singular form. The part of the label containing the resource information where to ship the order got removed.

After analyzing the labels, tagging the words towards their grammatical functions, and ensuring existence of a verb and a noun tag in each label, the input and output data nodes and the corresponding data states need to be determined from this information. We start with the output data nodes.

2—Output data nodes. The output data node extraction consists of three main steps (ODE) that are applied on each activity of a process model. Algorithm 2 details these steps formally.

(ODE-1) Identify precedence pattern for given activity (see line 1 and line 5),

(ODE-2) accordingly create the data nodes with corresponding data states (see lines 2 to 4 and lines 6 to 10), and

(ODE-3) add them as output data nodes to the given activity (see lines 12 to 14).

The algorithm for output data node extraction gets an activity as input and returns this activity with annotated data nodes as output. Therefore, the precedence pattern for the given activity is evaluated (ODE-1). If the activity precedes an XOR split (see line 1), data nodes get extracted from the data conditions annotated to the outgoing control flow edges of the gateway (see lines 2 to 4). Since a condition consists of a single data class and a single data state (cf. Definition 4.10), one data node per such control flow edge is created. The required information is extracted from the conditions by two functions: *class* and *state* (ODE-2; see line 3). Considering condition *Product in stock* in Figure 45, the corresponding output data node of activity *Check stock* is of class *Product* and has *in stock* as state.

Algorithm 2 Data output specification.

1: **if** $type_g(g) = XOR$ && $|\bullet g| = 1$ && $a\bullet = g, (a \in A, g \in G)$ **then**
2: **for all** $(g, n), n \in g\bullet$ **do**
3: $dn \leftarrow new\ DataNode(class(\xi(g, n)), state(\xi(g, n)));$
4: **end for**
5: **else**
6: $dataState \leftarrow new\ String(state(\mu(a));$
7: $labelList \leftarrow new\ List(partitionLabel(\mu(a), 1, < and, or >));$
8: **for all** $label \in labelList$ **do**
9: $dn \leftarrow new\ DataNode(class(label), dataState);$
10: **end for**
11: **end if**
12: **for all** dn **do**
13: $a.addOutputDataNode(dn);$
14: **end for**

Otherwise, i. e., the given activity does not precede an XOR split, the required information for data modeling is extracted from the activity label based on the parts-of-speech tagging from step ODE-1 (see lines 6 to 10). Referring to assumption LA-4 and that the action implicates the data state, each object manipulated through a single activity result in the same data state. Thus, we first determine the data state by transforming the verb into past participle and extends the result with the adverb if existing (see line 6). Then, we partition the label starting from position 1, i. e., after the verb, into multiple sublabel at the delimiters *and* and *or* (see line 7). For each sublabel, a new data node is created. It gets the nouns in conjunction with existing adjectives as class and the prior determined *dataState* as state (see line 9). Finally, all data nodes retrieved on either way are associated to the given activity as output data nodes. Thereby, only not yet associated data nodes are considered utilizing label and data state matching to avoid duplicates (ODE-3; see line 13). Afterwards, the algorithm starts over again with the next activity until each activity of the process model got assigned the appropriate output data nodes.

Based on the tagged activity label *Ship_VB order_NN*, the output data node of class *Order* in data state *shipped* is extracted and annotated to the activity accordingly. The data state *shipped* results from the past participle form of the verb while the noun is directly taken as label of the data node.

3—Input data nodes. Similarly to the output data node extraction, the input data node extraction consists of three main steps (IDE) that are detailed in Algorithm 3:

(IDE-1) identify successorship pattern for given activity (see line 1, line 3, line 23, line 25, line 27, and line 29),

(IDE-2) accordingly create the data nodes with corresponding data states (see line 2, lines 4 to 22, line 24, line 26, line 28, and line 30), and

(IDE-3) add them as input data nodes to the given activity (see lines 32 to 34).

Comparably to the data output specification, successorship patterns are matched against the local process structure for the given activity. While for the data output specification most patterns are handled identically, data input specification requires to handle each pattern differently. Additionally, the data input specification also requires information about output data nodes of various activities preceding the given, currently handled activity. For space and clarity reasons, two patterns are detailed in Algorithms 4 and 5 as referenced in lines 26 and 30 respectively.

If the given activity a_1 succeeds a start node, no input data node gets assigned to that activity (see lines 1 and 2; cf. EAA-4). If the given activity a_1 succeeds an other activity a_2 (see line 3), the distinct classes of all output data nodes of a_1 are retrieved (see line 4). Next, we check each path from the start node to activity a_1 whether a given data class

Algorithm 3 Data input specification.

```
1: if •a1 = n&&| • n| = 0&&|n • | = 1, (a1 ∈ A, n ∈ N) then
2:   //no input data nodes for activity a1
3: else if •a1 = a2, (a2 ∈ A) then
4:   for all c ∈ a1.getOutputDataNodes().getClasses() do
5:     foundDn ← new Boolean(false);
6:     for all σ such that n --σ--> a1 do
7:       k ← |σ| − 1;
8:       while k ⩾ 0 do
9:         if c ∈ nk.getOutputDataNodes().getClasses() then
10:          dn ← new DataNode(c, state(nk.getDataNodeOfClass(c)));
11:          foundDn ← true;
12:          break();
13:        end if
14:        k ← k − 1;
15:      end while
16:    end for
17:  end for
18:  if !foundDn then
19:    for all dn2 ∈ a2.getOutputDataNodes() do
20:      dn ← new DataNode(class(dn2), state(dn2));
21:    end for
22:  end if
23: else if type_g(g) = XOR && | • g| = 1 && • a1 = g, (g ∈ G) then
24:   dn ← new DataNode(class(ξ(g, a1)), state(ξ(g, a1)));
25: else if type_g(g) = XOR && |g • | = 1 && • a1 = g, (g ∈ G) then
26:   activitySucceedsXorJoin(); //see Algorithm 4
27: else if type_g(g) = AND && | • g| = 1 && • a1 = g, (g ∈ G) then
28:   //depends on predecessor of fork and utilizes the corresponding computations shown in
      this algorithm
29: else if type_g(g) = AND && |g • | = 1 && • a1 = g, (g ∈ G) then
30:   activitySucceedsAndMerge(); //see Algorithm 5
31: end if
32: for all dn do
33:   a1.addInputDataNode(dn);
34: end for
```

is represented by some output data node to an activity that belongs to the path (see line 6 to 17). Success is visualized through the Boolean variable *foundDn* which gets initialized in line 5. The path analysis starts with the control flow nodes directly preceding a_1: activity a_2 (see line 7). As long as the start node has not been reached (see line 8) and no data node was found (see line 12 that aborts path analysis), the classes of all output data nodes of the control flow node referenced by position k in the execution sequence σ of control flow nodes are checked whether one of them shares the class with the given data class c (see line 9). For non-activities, the empty set indicating no output data nodes is returned. If so, a new data node is created from the information of the one sharing the class, the Boolean variable is set to *true*, and further searching is avoided (see lines 10 to 12). In case, no data node of the current activity shares the data class, the directly preceding one is analyzed in the next iteration (see line 14). This procedure ensures identification of the data node that is written last by some activity on a specific path. After checking all paths leading to activity a_1 and no data node shared the data class with c, the data nodes of activity a_2 are

considered as the data nodes to be added to the input of a_1. Thus, we create a new data node for each of them (see lines 18 to 22).

If the given activity a_1 succeeds an XOR split (see line 23), similarly to the XOR split handling in line 3 in Algorithm 2, the data condition on the outgoing control flow edge of the XOR split leading to a_1 is considered for data node creation (see line 24). The data class and data state information for the new data node is retrieved from the condition by the corresponding functions.

If the given activity a_1 succeeds an XOR join (see line 25), the data node creation is handled as detailed in Algorithm 4. For input data node creation, we assume that the XOR block influences the given activity so that the block provides the input data nodes of the given activity (cf. EAA-2). In general, this data node creation is similar to the handling of two succeeding activities (see lines 3 to 22 in Algorithm 3). First, all paths through the XOR block enclosed by the join gateway g preceding a_1 and the corresponding split gateway g' are identified (see line 1 in Algorithm 4). For each such path, we check whether there exists an activity that has data nodes as output that refer to the same class as one of the output data nodes of a_1 (see lines 5 to 12). Analogously to the handling of two succeeding activities, we start with the predecessor of activity a_1 (see line 4) and move backwards until reaching the XOR split (see line 11). After identification of such data node for a specific data class c, the data node is taken and checking for this class c is aborted (see line 9). If the split gateway is reached, i. e., no activity with some required date node output is found, the data condition being valid for this path is evaluated (see line 13). If the data

Algorithm 4 Activity a_1 succeeds XOR join g.

1: **for all** σ such that $g' \xrightarrow{\sigma} g, (g' \in G, g$ is join, g' is corresponding split) **do**
2: $\quad foundDn \leftarrow new\ Boolean(false);$
3: $\quad$ **for all** $c \in a_1.getOutputDataNodes().getClasses()$ **do**
4: $\quad\quad k \leftarrow |\sigma| - 1;$
5: $\quad\quad$ **while** $k \geqslant 0$ **do**
6: $\quad\quad\quad$ **if** $c \in n_k.getOutputDataNodes().getClasses()$ **then**
7: $\quad\quad\quad\quad dn \leftarrow new\ DataNode(c, state(n_k.getDataNodeOfClass(c)));$
8: $\quad\quad\quad\quad foundDn \leftarrow true;$
9: $\quad\quad\quad\quad break();$
10: $\quad\quad\quad$ **end if**
11: $\quad\quad\quad k \leftarrow k - 1;$
12: $\quad\quad$ **end while**
13: $\quad\quad$ **if** $k = 0\ \&\&\ class(\xi(g', n_0) == c), (n_0 \in g'\bullet, n_0 \in \sigma)$ **then**
14: $\quad\quad\quad dn \leftarrow new\ DataNode(c, state(\xi(g', n_0)));$
15: $\quad\quad\quad foundDn \leftarrow true;$
16: $\quad\quad$ **end if**
17: $\quad$ **end for**
18: $\quad$ **if** $!foundDn$ **then**
19: $\quad\quad$ **for all** $dn_2 \in a_3.getOutputDataNodes(), a_3\bullet = \hat{g} \wedge (\hat{g} = g \vee (\hat{g} \xrightarrow{\sigma_g} g \wedge \forall\ \bar{g} \in \sigma_g : \bar{g} \in G)), (a_3 \in A)$ **do**
20: $\quad\quad\quad dn \leftarrow new\ DataNode(class(dn_2), state(dn_2));$
21: $\quad\quad$ **end for**
22: $\quad$ **end if**
23: **end for**

condition references the given data class c, a corresponding data node of class c is created with the data state given in the condition (see line 14). If no data node is found on the given path (see line 18), the activity first preceding the join gateway g is identified (see line 19; a_3 represents this activity). From this activity a_3, all output data nodes are retrieved. For each of them, we create a new data node that is considered as input of a_1 (see lines 19 to 21). The steps represented in lines 2 to 22 in algorithm Algorithm 4 are repeated for each path through the XOR block.

If the given activity a_1 succeeds an AND fork (see line 27 in Algorithm 3), the handling depends on the direct predecessor of the fork (see line 28). Therefore, we identify all direct predecessors $\bullet g$ of $g = \bullet a_1$. For each control flow node n_i of $\bullet g$, we apply Algorithm 3 and assume $n_i = \bullet a_1$.

If the given activity a_1 succeeds an AND merge (see line 29), the data node creation is handled as detailed in Algorithm 5. Algorithm 5 is very similar to Algorithm 4 since blocks enclosed by gateways of any type influence input data nodes of the directly preceding activity a_1 (cf. EAA-2). While only one path gets taken in XOR blocks, in AND blocks all paths are taken during process execution. Therefore, we do not require a data node coming from each path as input to a_1 but we require an input data node coming from some path. The second main difference is that AND blocks do not contain data conditions that can be used for data node retrieval. Applying these changes to Algorithm 4, an AND merge preceding a_1 is handled as follows. First, all paths through the AND block reaching the merge g starting from the corresponding fork g' are identified. For each path, we check whether there exists an activity that has data nodes as output that refer to the same class as one

Algorithm 5 Activity a_1 succeeds AND merge g.

```
1: for all σ such that g' --σ--> g, (g' ∈ G, g is merge, g' is corresponding fork) do
2:   foundDn ← new Boolean(false);
3:   for all c ∈ a_1.getOutputDataNodes().getClasses() do
4:     k ← |σ| − 1;
5:     while k ⩾ 0 do
6:       if c ∈ n_k.getOutputDataNodes().getClasses() then
7:         dn ← new DataNode(c, state(n_k.getDataNodeOfClass(c)));
8:         foundDn ← true;
9:         break();
10:      end if
11:      k ← k − 1;
12:    end while
13:  end for
14: end for
15: if !foundDn then
16:   for all a_3 ∈ A such that a_3• = ĝ ∧ (ĝ = g ∨ (ĝ --σ_g--> g ∧ ∀ ḡ ∈ σ_g : ḡ ∈ G)) do
17:     for all dn_2 ∈ a_3.getOutputDataNodes() do
18:       dn ← new DataNode(class(dn_2), state(dn_2));
19:     end for
20:   end for
21: end if
```

of the output data nodes of a_1 (see lines 5 to 12). Again, we start with the predecessor of activity a_1 (see line 4) and move backwards until reaching the AND fork (see line 11). After identification of such data node for a specific data class c, the data node is taken and checking for this class c is aborted (see line 9). After checking all paths, the Boolean variable *foundDn*, indicating whether some path contained an activity with a corresponding data node as output, is evaluated. If no data node is found on some path (see line 15), all activities first preceding the merge gateway g on some path are identified (see line 16). For each such activity a_3, all output data nodes are retrieved. For each of them, we create a new data node that is considered as input of a_1 (see lines 17 to 19).

Finally, after determining the input data nodes of the given activity a_1 based on the corresponding successorship pattern (see lines 1 to 31 in Algorithm 3), these are associated to a_1 as input data nodes (IDE-3; see lines 32 to 34). Thereby, only not yet associated data nodes are considered utilizing label and data state matching to avoid duplicates. Afterwards, Algorithm 3 is applied to the next activity of the process model until all activities got assigned their appropriate input data nodes.

Considering activity *Ship order to customer*, the successorship pattern resolves to *activity succeeds activity* (see line 3). The output data node is of class *Order* and has state *shipped*. Thus, we require to find a data node of class *Order* as output to some activity preceding *Ship order to customer*. The first (and only) such activity is *Accept order* resulting in data node of class *Order* with state *accepted* as input data node to activity *Ship order to customer*. Considering activity *Manufacture product*, the input data nodes are determined as detailed in Algorithm 4 due to the XOR join directly preceding the activity. On both paths through the XOR block, no activity exists with a corresponding output data node of class *Product*. However, the conditions reveal a match such that activity *Manufacture product* gets data nodes of class *Product* in states *in stock* and *not in stock* respectively.

After successfully performing the three steps for data node extraction (label analysis, output data node specification, and input data node specification), duplicated data nodes may be combined to improve clarity and readability of the process model. Duplicates appear if the same data node is used as input or output to different activities. These can be consolidated resulting in a single data node in the process model. For instance, the same data node *PO* in state *created* may be written by activity *Create PO* and read by activity *Prepare PO* instead of using separate data nodes for both activities as being the result of algorithms application. Following, the total number of used modeling constructs gets reduced. Figure 45b on page 102 shows the completely annotated process model; the data node reduction has been partly applied; for instance, data node *Order* in state *accepted* and the above mentioned

one. In contrast, data node *Product* in state *not in stock* is left duplicated due to the large distance between both activities writing respectively reading this data node. The duplicate reduction shall be supervised by a stakeholder who decides on a case basis.

The process model may also contain data nodes prior applying the algorithms introduced above. In these cases, the data nodes are preserved and additionally extended with the newly extracted ones. Duplicate detection as mentioned in the context of line 13 in Algorithm 2 and of line 33 in Algorithm 3 ensures addition of only not yet existing data nodes.

5.2 APPLICATION TO PROCESS DESCRIPTION LANGUAGES

Various process description languages allow incorporation of more information into a process model than the information given in above used generic process model. Therefore, the introduced algorithms can either be extended to incorporate these additional modeling concepts (*extension*) or the modeling concepts of a process description language can be mapped to existing generic concepts and handled accordingly (*alignment*). Alternatively, both operations can be combined. Thereby, each modeling concept that shall be considered for data node extraction must be linked to some generic modeling concept. First, we discuss the application of above algorithms to BPMN by alignment before we discuss the application to EPCs by extension.

BPMN by Alignment

BPMN is an expressive process description language with a large number of modeling concepts. Thus, we restrict the alignment to those concepts frequently used [171, 389]. In detail, the extraction alignments for BPMN (BEA) are the following with BEA-1 to BEA-3 being trivial although required for completeness:

(BEA-1) BPMN activities map to activities,

(BEA-2) BPMN start and end events map to the corresponding start and end events,

(BEA-3) BPMN XOR respectively AND gateways map to XOR respectively AND gateways,

(BEA-4) existing BPMN data nodes map to data nodes including input and output associations to activities and existing data states,

(BEA-5) receiving respectively sending BPMN messages map to input respectively output data nodes of the corresponding activity (cf. discussion about process orchestration semantics in Section 4.7),

(BEA-6) preceding receive respectively succeeding send BPMN message events of an activity map to input respectively output data nodes, and

(BEA-7) BPMN pools and lanes are ignored as these do not influence data node specification utilizing above algorithms.

After applying this mapping to a BPMN process model, the algorithms discussed in Section 5.1 can be run as is. Thereby, only data nodes and associations are added that do not existing already to avoid duplicates. For data nodes without data state specification existing before, the missing data states can be retrieved from the activity label if it encodes the corresponding data node. Otherwise, the user has to decide about the data node after enrichment computation. She may decide to add the data state information manually, to keep the data node as is, or to remove it from the process model.

Considering all concepts of BPMN leads to issues especially with respect to the inclusive OR and the complex gateways. The internal behavior of the complex gateway can hardly be mapped to an XOR or an AND gateway because of its unpredictable behavior specification in different process models. Similarly, the inclusive OR gateway is difficult to map since it functions as m-out-of-n discriminator with n being fixed by the number of paths through the inclusive OR block while m varies even between process instances such that the behavior is also unpredictable in many cases. However, sometimes the inclusive OR block can be replaced by a number of *common* (see above) modeling concepts [105], i. e., a combination of XOR and AND gateways. Then, the refined process model can be used for data node extraction.

EPCs by Extension

Likewise BPMN, EPCs provide additional modeling concepts that increase the information input for data node extraction. This time, the additional concepts do not map intuitively to existing generic concepts. Therefore, it is valuable to extend the algorithms from Section 5.1. Explicitly, the according modeling construct is the *event* which specifies the input to and the output of an activity respectively depending on whether it precedes or succeeds the activity – called function in EPCs. The extension comprises event consideration for data node specification, a removal of start and end node consideration since these are covered by events, and a removal of some successorship patterns due to the syntax of EPCs.

Events preceding a given activity are of interest for input data nodes while events succeeding a given activity are of interest for output data nodes specification. Therefore, Algorithm 1 needs to be extended with means to analyze event labels as well. For them, the same assumptions as for activity labels hold except that the labeling style follows the widely used technique of an object followed by an auxiliary verb and the action, e. g., *order is shipped*. Subsequently, the set of analyzed and tagged modeling concepts needs to be enhanced to activity and event labels as first step for the overall approach. Then, the two algorithms for data output and data input specification need to be adapted as follows. The syntax of EPCs disallows the sequences activity – XOR join, activity – activity, and XOR split – activity. Thus, the corresponding parts can be

removed from the algorithms. Instead, EPCs allow sequences activity – event, event – activity, gateway of any type – event, and event – AND gateway. These sequences require according handling in the algorithms. Data nodes extracted from events preceding an activity get input data nodes; data nodes extracted from events succeeding an activity get output data nodes. This information is combined with the activity analysis and allows input respectively output data nodes of multiple classes for a single activity. Thereby, duplicates get ignored as in the generic algorithms.

The approaches of extension and alignment can also be combined apart from trivial mappings as shown in BEA-1 to BEA-3. These trivial mappings are required because the modeling constructs are named differently in various process description languages. For instance, extended event-driven process chains, developed by ARIS, allow modeling of – amongst others – directed and undirected data accesses and organizational units. Directed data accesses can be handled as described in BEA-4 for BPMN. For undirected accesses, a solution needs to be specified. Trivially, this may be to add read and write associations for each undirected data access and let the user handle them afterwards. Alternatively, again extending the algorithms, preceding and succeeding events may be used to determine via label matching whether the undirected associated data node is read, written, or modified. If the corresponding data class is contained in the label of a preceding event, the data node is read. If the corresponding data class is contained in the label of a succeeding event, the data node is written. If the corresponding data class is contained in labels of both preceding and succeeding events, the data node is modified. Organizational units are ignored during algorithm application since they do not influence data node specification (cf. BEA-7). OR gateway issues also apply to EPCs and can be handled as discussed for BPMN.

5.3 EVALUATION

We prototypically implemented the approach to extract data nodes and their states from generic process models (see Section 5.1) and used this implementation as basis for an empirical evaluation. First, we discuss important implementation details followed by the empirical study.

Implementation

The proof of concept implementation is available at http://bpt.hpi.uni-potsdam.de/Public/ExtDO/. We use two external libraries: jBPT[1] [254] for the generic process model representation and the Stanford Log-linear Part-of-Speech Tagger [323, 324] for label analysis. jBPT is a Java-based library containing techniques for managing and ana-

1 http://code.google.com/p/jbpt/

lyzing business processes; e.g., a common representation of process models, into which process models from different process description languages as BPMN and EPCs can be transformed. This common representation consists of a superset of modeling concepts, we require a process model to provide for data node extraction. Additionally, it allows the visualization of process models (see Figure 45b on page 102 for such visualized process model after algorithms application). The Stanford Part-of-Speech Tagger "is a piece of software that reads text in some language and assigns parts of speech to each word"[2], i.e., it tags words with respect to their grammatical function as described in Definition 3.3 following the Penn Treebank English POS tag list [298]. In addition, we implemented own classes for label postprocessing, data node determination, and data node annotation.

Summarized, the implementation comprises the following functionality applied in this order:

(DEI-1) retrieve activity labels of the given process model,

(DEI-2) part-of-speech tagging with Stanford tagger,

(DEI-3) postprocess tagged labels,

(DEI-4) determine output data nodes and store them in a map comprising the data nodes for each activity,

(DEI-5) annotate output data nodes to activities in the process model utilizing the information from the map,

(DEI-6) determine input data nodes and store them in a map comprising the data nodes for each activity,

(DEI-7) annotate input data nodes to activities in the process model utilizing the information from the map, and

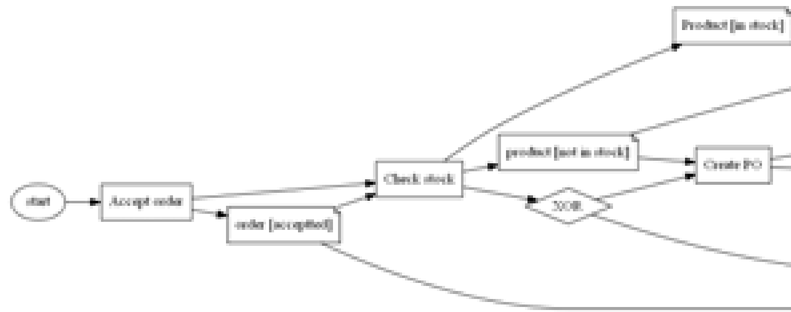
(DEI-8) remove data node duplicates.

Thereby, DEI-2 and DEI-3 refer to Algorithm 1, DEI-4 and DEI-5 refer to Algorithm 2, and DEI-6 and DEI-7 refer to Algorithm 3. Applying the implementation on the process model given in Figure 45a on page 102 leads to the process model shown in Figure 45b as BPMN diagram and shown in Figure 46 as screenshot distributed over five figures utilizing GraphViz [90] for visualization of the process graph. The provided implementation is not capable of transforming the generic process representation back to BPMN after data annotation. This is subject to tool improvements.

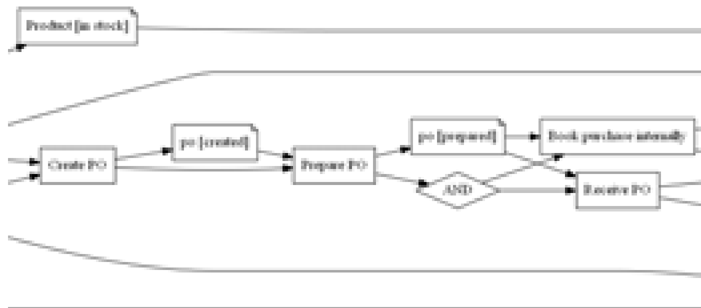
Empirical Evaluation

Utilizing this implementation, we conducted an experiment with 29 participants to evaluate the appropriateness and usefulness of the data node extraction algorithms. The participants of the user experiment are students and researchers in computer science. None of them has been involved in the development of the algorithms or their implementation. The level of experience ranges from beginner to expert; about

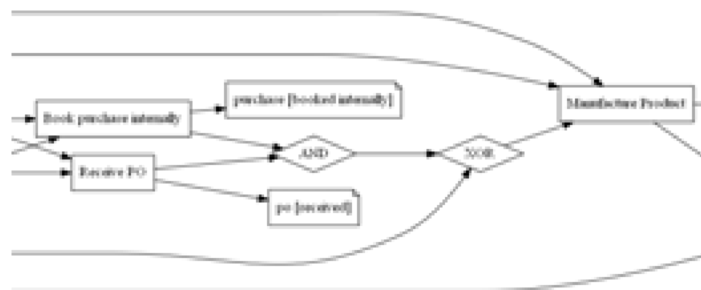
2 http://nlp.stanford.edu/software/tagger.shtml



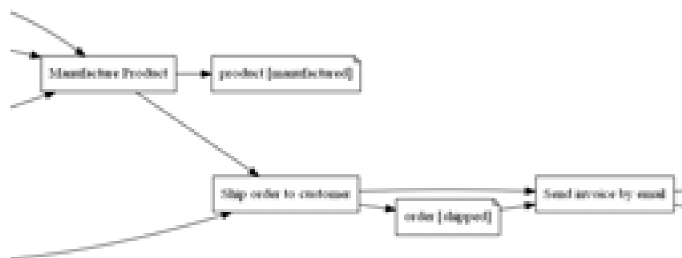
(a)



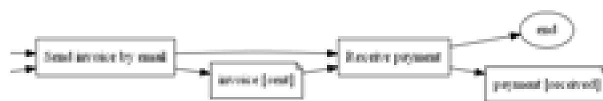
(b)



(c)



(d)



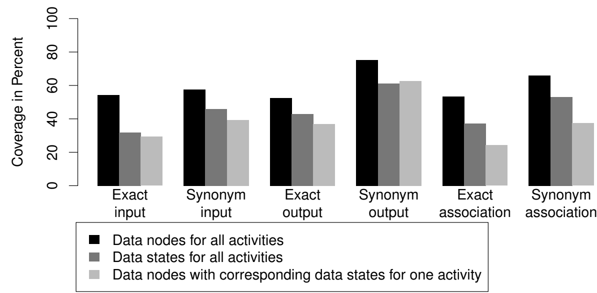
(e)

Figure 46: Screenshot of GraphViz-visualization [90] of computation result based on our implementation.

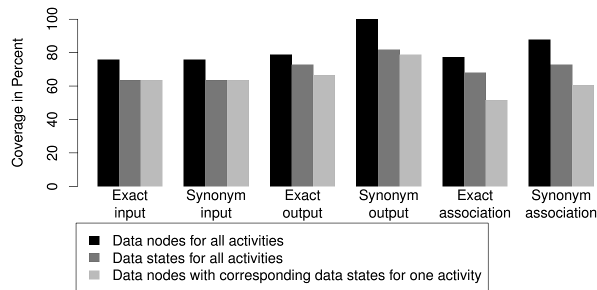
half the participants have a strong background in business process management (BPM). *Beginner* in this case refers to undergraduate students having successfully finished their first lecture on business process management focusing on BPM fundamentals and business process design. *Strong* in this case refers to at least two years continuous hands-on experience, e. g., conducting research or performing project works. The experiment was separated in three main parts (questionnaires) tackling the data node extraction. The first one dealt with annotating given process models with data nodes while the second one asked the participants to rate the data annotations for specific activities induced through application of our implementation. The third part asked for the appropriateness of input and output data node specifications as well as the understandability for entire process models. These three questionnaires allow validation whether the resulting, with data nodes annotated process models can be used for process model analysis or whether it can be refined towards an executable process model. The process models utilized in the questionnaires are taken from the BPMAI process model collection [34] to utilize real world process models from different users describing various scenarios, e. g., order processes as used in this thesis and known from online shopping, manufacturing of bikes, cars, etc., or treatment of patients in a hospital.

We used ten different process models for the experiment – reusing some in different questionnaires – with three to five process models in each questionnaire. Each process model was presented as BPMN diagram without further textual descriptions, i. e., the participants based their decisions on the presented graph only analogously to the extraction algorithms. Complexity-wise, the process models contained at average ten activities, four gateways, and the two start and end events summing up to 16 control flow nodes at average per process model with a standard deviation of 3.6 nodes and a median of 15.5 nodes, i. e., a common complexity often found in process model collections.

However, in practice, large process models with more than 1,000 control flow nodes exist [312]. We argue that the introduced approach works similarly for both the evaluated and such large process models, because the algorithms step through the process model and determine the part-of-speech tagging as well as the data node annotations one by one for the process model's control flow nodes. In addition, the data input specification requires to analyze one preceding AND or XOR block and might be required to check paths through a large fraction of the process model. Therefore, the computation complexity per control flow node is comparable for output data node specification while the input data node specification complexity may be increasing through the path analysis. Nevertheless, abstracting from loops by considering them once, path length is finite and path analysis is aborted upon first match. Thus, these large process models are manageable with reasonable effort. The overall computation time mainly increases with respect to the



(a) All assessed process models.



(b) Process models with good quality activity labels.

Figure 47: Coverage of manual compared to automatic data annotation utilizing our implementation based on Algorithms 1 to 3.

number of given control flow nodes. In the experiment, the participants have seen the corresponding process models for the first time such that they cannot utilize previous knowledge to allow comparability of the results. All participants received questionnaires that were identically structured and that utilized the same process models.

Below, we will present the results of the experiment which had the goal to tackle the following four aspects:

1. Comparison of automatic and manual data node annotations, where manual annotations provide the gold standard for automatic annotation,
2. quality and usability of data node annotations in different usage scenarios,
3. additional insights to and knowledge about the business process represented by the process model, and
4. understandability of a resulting process model with annotated data nodes.

The comparison results between automatic and manual data annotations are provided in Figure 47 (first questionnaire). The bars show the coverage in percent of the number of data nodes (black bars) or data states (dark gray bars) identically identified by the algorithms and by humans. The light gray bars indicate the coverage of activities completely specified with data nodes and their states. This conformity is shown separately for input and output data nodes of activities as well as combined for both types of data nodes. Thereby, we distinguish an

exact label match (group one, three, and five) and a synonym match, because humans may consider external knowledge and personal taste for annotation such that, for instance, a *bill* data node is extracted from a *send invoice* label or an *order* is changed to *customer order* or vice versa. Humans also tend to add extensions like *report* for assessment activities. With synonym match, we consider data nodes or states providing the same meaning and understanding although string matching might fail.

For all criteria, the data node extraction (black bars) reaches the highest conformity. The lowest values are naturally reached for the combination of data node and data state extraction (light gray bars), because it represents the intersection of the other two criteria.

Altogether, the data annotations done through our algorithms are generally quite close to the humans' annotations with the majority of data nodes and data states being compatible to each other (see Figure 47a). Thereby, output specifications provide better results than the input ones, because output specifications only base on information from one activity and do not – as the input specifications do – require analysis of preceding control flow nodes that reduces correctness since multiple uncertain sources are combined to determine the respective data nodes and their states. However, the main problems for both annotation procedures (input and output specification) are abstract or confusing activity labels, i. e., too abstract or detailed abstraction level as well as syntax and semantic errors. Additionally, the possibility in English language to use the same word in different grammatical functions caused confusions; e. g., the labels *select target audience* and *target audience wishes* in the same process model led to some confusion because, for both labels, some experiment participants considered *target audience* as one data class to be represented in a data node while *audience wishes* in state *targeted* is determined for the second label based on the introduced algorithms and some other human participants. For the first label, participants and algorithms determined compatible data nodes and states.

Evidence shows that the quality of activity labels plays a major role with respect to the results to be expected from our approach – and from the humans. A good label quality refers to labels that actually describe the work performed by the corresponding activity with clear and concrete statements and align to the structure of the process model. This also includes an appropriate level of abstraction which shall be comparable throughout the complete process model. For instance, activity label *send invoice by email* consists of a good quality, because it states clearly what happens without too many or too few details. In contrast, labels of poor quality either abstract completely from the work to be performed in the activity, contain ambiguous statements, or are very detailed by, for instance, making proposals about what should be done within activity execution; e. g., activity label *consider other causes of distress and pain* is very specific and also humans have difficulties to

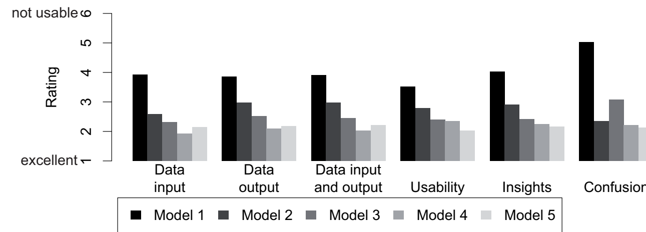


Figure 48: User rating of data annotation quality and usefulness.

extract data nodes which might be affected by such activity. Following, clear and good structured activity labels also increase the conformity between automatic and manual extraction of data nodes.

Referring to our user experiment, removing process models, which majority of activities is poorly labeled, the results increase by 25 percentage points at average (see Figure 47b). For example, the association of data nodes and states on a synonym basis increases from 37% to 61% (most-right bars in Figure 47). The remaining difference reflects the different thinking of humans as well as their usage of external knowledge.

Figure 48 visualizes the results from the third questionnaire: the assessment of the quality of the automatically added data annotations on a Likert scale from 1 (excellent) to 6 (not usable). The experiment participants rated the data input and data output specifications of complete process models separately as well as combined (bar groups one to three) and the usability (for further analysis or as starting point for refinement towards executable process models), the ability to derive new insights from the process model, and the level of confusion triggered by data nodes (lower rating indicates lower confusion). Especially for process models 2 to 5, the figure shows very convenient results and confirms the good quality of the automatic data annotations. Process model 1 contains almost only abstract activity labels from the healthcare domain such that the extracted information is, as indicated, of low use for process understanding or utilization.

Considering all process models, the arithmetic averages are 2.6 ± 0.8, 2.7 ± 0.7, and 2.7 ± 0.8 for the various data specifications and 2.6 ± 0.6, 2.8 ± 0.8, and 3.0 ± 1.2 for usability, insights, and the level confusion respectively. Ignoring process model 1, the averages improve to 2.2 ± 0.3, 2.4 ± 0.4, and 2.4 ± 0.4 for the data specifications and 2.4 ± 0.3 for usability and insights and 2.4 ± 0.4 for the level of confusion, which means stable and high-level results for different process models with a good label quality from different domains. Interestingly, the scores for input and output data node specification are very close in contrast to the first questionnaire with data input specification being slightly better than data output specification. This leads to the assumption that the participants similarly agree on the extraction results for data input and data output specification while the participants may choose – especially in case of input data nodes – different data nodes if they were asked to do

so without result representation. However, the strong agreement to the extracted data nodes further leads to the assumption that the difference only refers to the chosen label and not the data node's content. Additionally, we identified a direct correlation between all six evaluated aspects, which again correlate to the quality of the activity labels and the structure of the process model. For instance, long sequences of activities that read and write many different data nodes often result in repetition of the activity label in the data node and therefore, the complexity is increased without adding value. In contrast, process models with several building blocks benefit from the approach and provide insights about data usage and manipulation throughout the complete process model.

In the second questionnaire, the participants were asked to rate the appropriateness of the data node annotations for single activities in the context of the completely annotated process model, i. e., the participants were provided with the annotated process model. Most annotation results have been either rated very low (worse than score 4 at average) or very high (better than score 2 at average); only few scores are around three. The overall scores are 2.2 ± 1.1 for input data nodes and 2.3 ± 0.6 for output data nodes. The comparably high variance for input data nodes can again be traced back to the fact that multiple uncertainties are combined to determine an input data node. However, input data nodes may extremely benefit from a good label quality. Ignoring activities with a poor label quality decreases above numbers to 1.6 ± 0.3 and 2.1 ± 0.6 respectively leading to an appropriateness rating between good and excellent and shows the general applicability for well labeled activities. Again, as discussed for the third questionnaire and in contrast to the first one, the data input scores are better than the data output scores with both denoting a strong acceptance of the extraction results; mostly, one of the two highest scores (rating 1 or 2) were granted to the extraction result.

Summarizing the insights from the experiment, the results are very promising but highly depend on the label quality. Reflecting on the results for process models mainly consisting of activities with good quality labels, the comparison of automatic and manual data annotation has a conformity of about two thirds with respect to the number of data nodes and data states. The resulting automatically annotated activities and process models achieved high ratings with respect to annotation quality, usability in different scenarios, knowledge gaining, and understandability.

Good label quality is achieved by utilizing modeling guidelines in conjunction with a taxonomy or glossary specifying the correct term for each action and object [193, 203, 205, 302]. Generally, label quality is a requirement of the modeling process and can be enforced by the modeling tool. The experiment showed that the resulting process models can be directly used for empirical research or as basis for process au-

tomation, if the labels are clear, concrete, and aligned with the process structure. In contrast, ambiguous, very detailed, or very generic labels lead to process models, which may act as starting point to annotate the process model with data nodes and their states by providing insights about the manipulations performed by the activities.

5.4 RELATED WORK

The data extraction approach described in this chapter relies on findings from [204], where the authors determined a general label construction schema consisting of three building blocks. The insight of existence of an object and an action manipulating this object is the basis for our approach since we require both building blocks to specify the data objects being input or output to an activity. Additionally, further activity label analysis has been performed. Leopold and colleagues analyzed the labels with respect to its grammatical structure and determined seven different labeling styles [182, 183]. From these, they chose the verb object style labeling as the reference labeling style, because this is the one to be recognized as good practice for modeling processes. Therefore, the authors provide means to transform a given activity label of non-irregular-style into verb-object-style [183]. Labels of irregular-style have an anomalous structure that is represented, for instance, by "punctuation symbols [...] connecting the parts of the label in a specific manner"; e. g., "*Profit Center Assessment: Plan* or *LIFO: Valuation: Pool Level*" [183]. These transformations into the verb-object-style can be used as pre-step to our approach to increase the number of process models, which can be enriched with data nodes and data states.

To be able to gather information from activity labels, they need to be analyzed with natural language processing techniques. There exist several frameworks to do so. Two of the well-known ones are the Stanford Part-of-Speech Tagger [323, 324] and the KPML system [21]. The label analysis results in words tagged towards their grammatical function, e. g., verb in infinitive form. We use this information to reason about the data nodes and their states. In our implementation, we use the Stanford tagger due to easy integration as jar library.

Another stream of research deals with data in process models. For instance, there are approaches to derive OLCs from a process model [97, 98, 186, 292], which also need to determine data nodes (objects) and the corresponding data states in the process model. However, the existing approaches in this regard require process models being annotated with data. Thus, these approaches can be extended with the extraction approach introduced in this chapter to derive object life cycles from process models consisting of control flow only.

Data in business processes also leads to object-centric modeling approaches [54, 237], where a process is modeled by the involved data classes with each having a life cycle and multiple objects of these classes

synchronize via their state changes. In contrast, in activity-centric modeling, data nodes are used as second class modeling construct while activities describe the process behavior. Here, data nodes describe pre- and post-conditions to activities describing under which conditions an activity can be enabled and what is the output produced by an activity. The approach presented in this chapter focuses on enriching an activity-centric process model with data nodes rather than creating an object-centric process model (OCP). Modeling methodologies for OCPs are presented in literature [231, 237]. Additionally, an activity-centric process model with annotated data nodes can be transformed into an object-centric process model [186] and vice versa (see Chapter 7 for both directions).

5.5 CONCLUSION

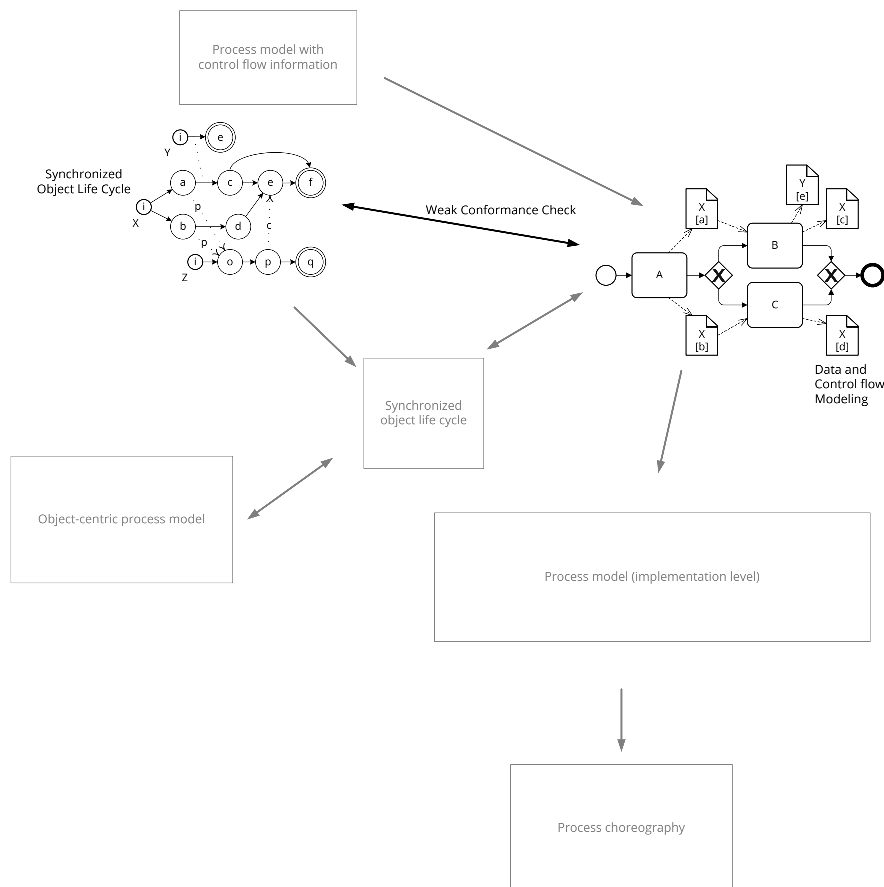
Targeting the support of data modeling, we introduced an approach to extract data nodes and their states from control flow information. The corresponding algorithms follow a three step methodology where we first analyze the activity label to determine actions and manipulated objects before we extract output data nodes directly from the activity label and input data nodes from the labels and the structure of preceding activities. We first introduced the generic procedure applicable for all process description languages comprising the stated modeling constructs: Control flow nodes, data nodes, activity labels, control flow, and data flow. Afterwards, we show how the algorithms can be extended by alignment or by extension to utilize additional information given by some process description language to increase result quality.

Overall, the resulting process models can be used, amongst others, as starting point for process execution, where they require some more refinement, or directly for empirical research in business process management, if the activity labels are of good quality. Otherwise, the resulting process models provides first insights on data utilization and can be used as basis for creating process models to be executed. Tool support – which was out of scope in this chapter – would highly support approach application and as such decrease the effort for data modeling.

A tool implementing the algorithms described would provide benefits to researchers working in the field of data-aware business process management with need to evaluate their algorithms and approaches as well as to organizations either moving from documentation to execution with their processes or directly intend to execute them. The latter receive comprehensive insights about the usage of data objects throughout their processes. Indeed, for single process models, the enrichment may be done manually. But considering a large process collection with hundreds of models, where each process model may also consist of more than thousand control flow nodes, manual processing is not feasible, since it would require much time.

6 WEAK CONFORMANCE OF PROCESS SCENARIOS

This chapter is based on results published in [213, 214, 217].



Ensuring correctness of process scenarios belonging to a business process is important to allow proper process execution and achievement of the desired goals, e. g., processing the order of a customer as discussed in the build-to-order and delivery process in Section 2.4. Incorrect models lead to ambiguities, may foster misinterpretations, and prevent process execution [382]. Correctness can be distinguished in different types: syntactical correctness targeting the design-time, e. g., unconnected nodes or improper use of nodes, (that we assume to be correct in this chapter) and behavioral correctness targeting the runtime, i. e., execution of processes. Usually, behavioral correctness of control flow is checked through, for instance, soundness [74, 331] or model [18, 51] checking. Model checking is often used to ensure the adherence to given rules referred to as compliance checking. Soundness

checking is a proven technique to reason about behavioral correctness of process models including executability and terminability. For process execution, checking control flow behavior is not sufficient since execution highly depends on data and its usage. Therefore, also data flow correctness must be ensured. Checking the correctness of data flow is two-fold. On the one hand, data manipulations specified in the process model must comply to organizational restrictions. Such restrictions are, for instance, represented by "global" object life cycles that specify all manipulations to data that are allowed in the organization. Determining consistency between a process model and an OLC is commonly done by data conformance checking [176, 292]. On the other hand, the data dependencies specified in a process model must not block process execution and must lead to a satisfactory process termination. Thereby, all data objects utilized in the process model must be considered collectively. Figure 51b on page 127 shows a typical example where data access blocks process execution. Activities *Create purchase order* and *Create request* are in exclusiveness relation and an object of data class *PO* is written in state *created* from the first activity while the second one reads this data node resulting in a deadlock once the process execution decides for the lower path in this process model.

The combination of both data flow correctness types represents behavioral correctness of the data flow. Targeting the first type, this chapter introduces the *notion of weak conformance* and its computation as extension to the notion of object life cycle conformance [176, 292] to allow the support of underspecified process models and object life cycle synchronization. In process orchestrations, a fully specified process model contains all reads and writes of data nodes by all activities. Additionally, each activity reads and writes at least one data node except the first and last activities of the process model, which may lack reading respectively writing a data node in case they only create respectively consume a data node. In contrast, underspecified process models may lack some reads or writes of data nodes such that they are implicit, performed by some other process, or they are hidden in aggregated activities changing the state multiple times with respect to the corresponding OLC. Though, full support of underspecified process models requires that the process model may omit state changes of data nodes completely although they are specified in the OLC.

Underspecification example

Figure 49 shows a condensed and slightly adapted version of the *process order* process model introduced in Section 2.4 (see Figure 7). We reduced the number of activities and utilized data classes to reduce the process model's complexity for a clearer presentation of the concepts of this chapter. The given process model utilizes four data classes: *CO*, *Product*, *ProC*, and *Invoice* and therefore also requires four object life cycles enriched with synchronization edges resulting in the synchronized object life cycle shown in Figure 50. The given process model is underspecified for several reasons. For instance, activity *Create purchase order*

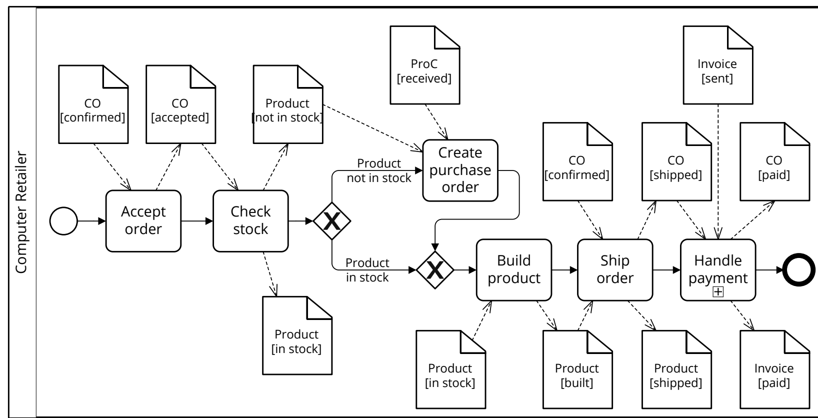


Figure 49: Condensed and slightly adapted version of the *process order* process model introduced in Section 2.4 (added data dependencies for the last three activities).

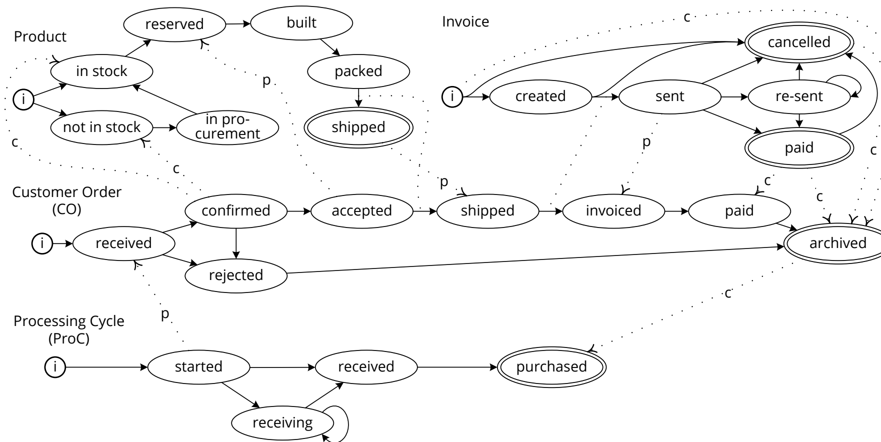


Figure 50: Synchronized object life cycle for data classes *ProC*, *CO*, *Product*, and *Invoice* utilized in the process order process model in Figure 49.

reads a data node *Product* in state *not in stock* while the very next activity, *Build product*, requires a *Product* in state *in stock* as input. The data state transitions from state *not in stock* to state *in procurement* and further to state *in stock* are not modeled in the process model. These transitions may either happen within this process model but in the current view, the information is hidden. Or these transitions may be covered by some other process model dealing with the procurement activities. Considering activity *Handle payment*, it comprises multiple data state transitions. Considering Figure 50, this activity changes the state of a *CO* object first from *shipped* to *invoiced* and subsequently from *invoiced* to *paid*. Similar holds true for activity *Build product* that comprises multiple state transitions with respect to an object of data class *Product*.

OLC conformance comparison

Next, we compare different approaches to check for conformance between a process model and corresponding OLCs (first data flow correctness type). Table 2 lists the applicability and specifies the time complexity of the computation algorithms for approaches described in

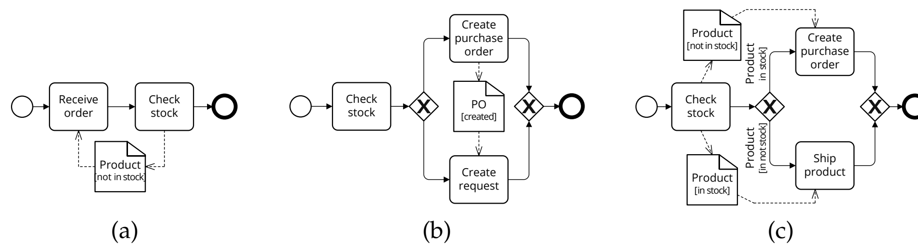
Table 2: Applicability and time complexity of data conformance computation algorithms.

Attribute	[176, 292]	[364]	this
Full specification	+	+	+
Underspecification	-	o	+
Synchronization	-	-	+
Complexity	poly.	exp.	exp.

[176, 292], [364], and this chapter: weak conformance via soundness checking (see Section 6.2). These approaches differ in complexity and the process models, that can be handled, such that they can be applied in different scenarios. The variety of conformance checking techniques, all independent to each other, also allows to double check and therefore strengthening the results. The notion from [176, 292] requires fully specified process models and abstracts from inter-dependencies between OLCs by not considering them for conformance checking in case they are modeled. Conformance computation is done in polynomial time. In [364], underspecification of process models is partly supported, because a single activity may change multiple data states at once (aggregated activity). Though, full support of underspecified process models would require that the process model may also omit data state changes completely although they are specified in the OLC. Fully specified process models are supported while synchronization between OLCs is not considered in that approach. Complexity-wise, the approach requires exponential time. The soundness checking approach supports fully and underspecified process models and provides solutions for object life cycle synchronization by including specified synchronization edges into the weak conformance notion. This approach requires exponential time through the Petri net mapping and subsequent soundness checking as described in Section 6.2. However, state space reduction techniques may help to reduce the computation time for soundness checking [101].

Since soundness checking is a proven standard technique to check for behavioral correctness of control flow in terms of proper executability and terminability and due to the existence of a Petri net mapping covering control flow and data aspects of a process model (see Section 4.7), we apply soundness checking to verify weak conformance as data flow correctness check. Besides following proven techniques, choosing soundness checking has multiple advantages. It allows to check for control flow and data flow correctness in one analysis step and still allows to distinguish occurring violations caused by control flow or data flow based on the affected places and transitions (see Section 6.2 for details on identification and see Section 6.3 for details on correction of errors.). While the notion of weak conformance only

covers the conformance between OLCs and process models (first data flow correctness type), application of soundness checking allows to identify structural errors with respect to data node utilization in the process model (second data flow correctness type). In case conformance to OLCs is not required, the second data flow correctness type, deadlock-free data execution, can be checked by utilizing the Petri net mapping and applying soundness checking without following the steps described in the next section.



(a) (b) (c)

Figure 51: Examples for identifiable modeling errors.

Figure 51 visualizes three errors that can – amongst others – be identified. Figure 51a shows the read of a data node *Product* in state *in stock* which is only written afterwards such that the corresponding data object cannot be in the requested data state at that point in time resulting in a deadlock on activity *Receive order*. Figure 51b visualizes the fact that activities *Create purchase order* and *Create request* are in exclusiveness relation and activity *Create request* needs to read a data node that is only written by activity *Create purchase order* such that the process deadlocks because of data flow if the lower path of this XOR block is executed. In Figure 51c, the conditions assigned to the control flow edges originating from the XOR gateway contradict to the specified data node reads of activities *Create purchase order* and *Ship order* within the XOR block resulting in a deadlocking process for either path although the assigned data conditions are non-blocking.

[14] introduces three classes of errors related to data the authors refer to as *data anomalies*:

(DAC-1) too restrictive preconditions,
(DAC-2) implicit routing, and
(DAC-3) implicit constraints on execution order.

DAC-1 denotes the fact that data nodes (objects) are not in the data state that is expected by an activity that is enabled from the control flow point of view. Figures 51a and 51b are examples for this class. DAC-2 is a special case of DAC-1 and denotes the fact that multiple branches require data nodes (objects) to be in specific states that are not imposed correctly by the branching conditions – the data conditions on the control flow edges originating from a split gateway. This can be due to missing branching conditions or to incorrect ones as given in Figure 51c. DAC-3 denotes the fact that control flow may allow con-

currency between two activities while the data flow requires a specific sequence between those activities since one may have a data output that is required by the other as input (cf. Figure 51b). While the authors claim that such sequentialization must be reflected in the control flow, we abstract from this requirement as we consider both data and control flow as first class modeling concepts and the execution semantics discussed in Chapter 4 ensure proper execution if the data flow is non blocking with respect to DAC-1. Therefore, we target identification of data errors of anomaly classes DAC-1 and DAC-2.

6.1 THE NOTION OF WEAK CONFORMANCE

Weak conformance is checked for a process model with respect to the OLCs referring to data classes used within the process model, i. e., a process scenario $ps = (pm, \mathcal{L}, C)$. Thereby, a process model requires information about the contained control flow nodes and data nodes as well as the corresponding control flow and data flow relations, activity labels, the type of gateways, and data conditions to control flow nodes originating from XOR gateways; i. e., in this chapter, we refer to a process model as tuple $pm = (N, D, \mathfrak{C}, \mathfrak{F}, Q, type_g, \mu, DCF)$ with $DCF = (\xi)$. This is the minimal information required to apply the notion of weak conformance to a process scenario. Additional information as specified in Definition 4.10 may be present and does not influence weak conformance computation and application.

Next, we define several notions for convenience considerations before we introduce the notion of weak conformance. Let $df \in \mathfrak{F}_{pm}$ be a data flow edge of process model pm. With df_A and df_D, we denote the activity and data node component of df respectively. For instance, if df is equal to (a, d) or to (d, a), then (in both cases) $df_A = a$ and $df_D = d$. With $s_{d,df}$, we denote the data state involved in a read ($df = (d, a) \in \mathfrak{F}$) or write ($df = (a, d) \in \mathfrak{F}$) operation of data node d. We denote the set of synchronization edges having data state $s_{d,df}$ as target data state with $SE_{s_{d,df}}$. Recapitulating execution sequences introduced in Chapter 3, $a \Rightarrow_{pm} a'$ denotes that there exists an execution sequence in process model pm that starts in the initial marking and executes activity $a \in A_{pm}$ before activity $a' \in A_{pm}$. Analogously, $s \Rightarrow_{l_c} s'$ denotes that there exists an execution sequence in the object life cycle l_c of data class c that starts in the initial state and reaches state $s \in S_c$ before state $s' \in S_c$.

Definition 6.1 (Weak Data Object Conformance).
Given process scenario $ps = (pm, \mathcal{L}, C)$ where process model $pm = (N, D, \mathfrak{C}, \mathfrak{F}, Q, type_g, \mu, DCF)$ with $DCF = (\xi)$ and where the synchronized object life cycle $\mathcal{L} = (L, SE)$ for data classes C, process model pm satisfies *weak conformance* with respect to data class $c \in C$ if for all $df, df' \in \mathfrak{F}$ such that $df_D = d = df'_D$, $d \in D$, with $\varphi_D(d) = c$

holds (i) $df_A \Rightarrow_{pm} df'_A$ implies $s_{d,df} \Rightarrow_{l_c} s_{d,df}$, (ii) $\forall\, se \in SE_{s_{d,df}}$ originating from the same object life cycle $l \in L : \exists \Pi(se) = true$, and (iii) $df_A = df'_A$ implies df represents a read and df' represents a write operation of the same activity. ◂

Given a process scenario, we say that it satisfies weak conformance, if the process model satisfies weak conformance with respect to each of its data classes. Weak data object conformance is satisfied, if for each two succeeding data states of a data class in a process model there exists an execution sequence from the first to the second data state in the corresponding OLC and if the dependencies specified by synchronization edges with a target state matching the second data state of the two succeeding ones hold such that all dependency conjunctions and disjunctions are fulfilled. Due to the transitivity property, we can reduce the checking from each pair of succeeding data states to pairs of directly succeeding data states. Consider activities accessing data states *received*, *confirmed*, and *accepted* from the OLC for data class *CO* in Figure 50. The fact that *accepted* is reachable from *received* is implied by the facts that *confirmed* is reachable from *received* and that *accepted* is reachable from *confirmed*. Thus, the check for reachability of *accepted* from *received* can be omitted to reduce computation complexity. Two data states are *directly succeeding* in the process model, if either (i) they are accessed by the same activity with one being part of an input and one being part of an output data flow edge or (ii) there exists an execution sequence in the process model, in which two different activities access the same data class in two data states with no further access to this data class in-between.

Considering the build-to-order and delivery process introduced in Section 2.4, weak conformance must be checked for each of the presented process models. For instance, if the process model in Figure 5, the big picture on the process, conforms to the utilized data classes, this does not imply that all contained subprocesses also comply. Therefore, each subprocess, for instance the *collect order* one in Figure 6, must be checked against its utilized data classes. If the big picture and all subprocesses satisfy the notion of weak conformance, we say that the process scenario and thus the business process completely conforms to the given synchronized object life cycle. Combining this statement with the soundness check utilized for weak conformance computation, the satisfaction implies a correct business process execution with respect to control flow and both types of data flow correctness.

Consider the process model in Figure 49 on page 125 and the corresponding synchronized OLC in Figure 50 as a reduced subset of the full scenario from Section 2.4. This process model satisfies the notion of weak conformance with respect to data classes processing cycle *ProC* and *Invoice* and does not satisfy weak conformance with respect to data classes customer order *CO* and *Product*. For instance, data class *Invoice* satisfies the notion of weak conformance since each pair of directly

succeeding data states is reachable in the OLC; e. g., the transition of state *sent* to state *paid* as specified by activity *Handle payment* is covered by the object life cycle – data state *paid* is reachable from state *sent*. For data class *CO*, the notion of weak conformance is violated, since there exists an execution sequence in the process model that executes activity *Check stock* before activity *Ship order* such that both are directly succeeding with respect to accessing data class *CO* in data states *accepted* and *confirmed* respectively. However, there does not exist an execution sequence from data state *accepted* to data state *confirmed* in the corresponding OLC of data class customer order *CO*. The weak conformance check regarding data class *Product* fails for synchronization issues. With respect to the synchronized object life cycle in Figure 50, data states *in stock* and *not in stock* can only be reached, if the corresponding object of data class *CO* is in data state *confirmed* once the transition to either of the mentioned data states of data class *Product* shall occur. In the given process model, activity *Check stock* writes an object of data class *Product* either in data state *not in stock* or in data state *in stock* while it reads an object of class *CO* in state *accepted*. This contradicts the dependency requirement of data state *confirmed*. In fact, in the given process scenario, an object of class *CO* is never in data state *confirmed* when an object of class *Product* shall transition to data state *in stock* or data state *not in stock* as specified by activity *Check stock*.

There are several opportunities to solve the discussed issues. Satisfaction of the weak conformance property for data class *CO* can be achieved, for instance, by changing the data state of the input data node of activity *Ship order* to *accepted* or to remove the data node from the process model. With respect to data class *Product*, the sources of the according synchronization edges between data classes *CO* and *Product* may be moved from data state *confirmed* to data state *accepted* to satisfy the notion of weak conformance. Alternatively, the dependency type may be changed to *previously*. Application of one of the changes for each discussed violation leads to a process scenario that satisfies the notion of weak conformance for all utilized data classes. A detailed discussion about correction of process scenarios violating the notion of weak conformance follows in Section 6.3.

The notion of weak conformance cannot only be used to check process scenarios for data conformance. It may also support process experts during the refinement of process models from organizational to implementation level. Further, given OLCs can be reduced in size and complexity with respect to the process model at hand. We will discuss both application options in detail in Chapter 7 as *intra-view transformations*.

6.2 computation via soundness checking

A given process scenario $ps = (pm, C, \mathcal{L})$ with $\mathcal{L} = (L, SE)$ can be checked for weak conformance by applying the following four steps in sequence:

1. Map the process model pm and the synchronized object life cycle $\mathcal{L}$ to Petri nets,
2. integrate both Petri nets,
3. post-process the integrated Petri net and map it to a workflow net system, and
4. apply soundness checking to identify violations within the process scenario ps.

Before proceeding with discussing these four steps, we recall the notions of preset and postset. A preset of a transition t respectively a place p denotes the set of all places respectively transitions directly preceding t respectively p. A postset of a transition t respectively a place p denotes the set of all places respectively transitions directly succeeding t respectively p.

1—Petri net mapping. The process model is mapped to a Petri net following the rules described in [80] for the control flow and in Section 4.7 for the data flow. Figure 52 shows an extract for activities *Accept order* and *Check stock* of the process model given in Figure 49 on page 125. The mapping of the synchronized object life cycle is twofold. First, each single OLC $l \in L$ is mapped to a Petri net, which than secondly are integrated utilizing the set of synchronization edges. The mapping of single OLCs utilizes the fact that Petri nets are state machines, if and only if each transition has exactly one preceding and one succeeding place [334]. Thus, each state of an OLC is mapped to a Petri net place and each data state transition connecting two states is mapped to a Petri net transition connecting the corresponding places.

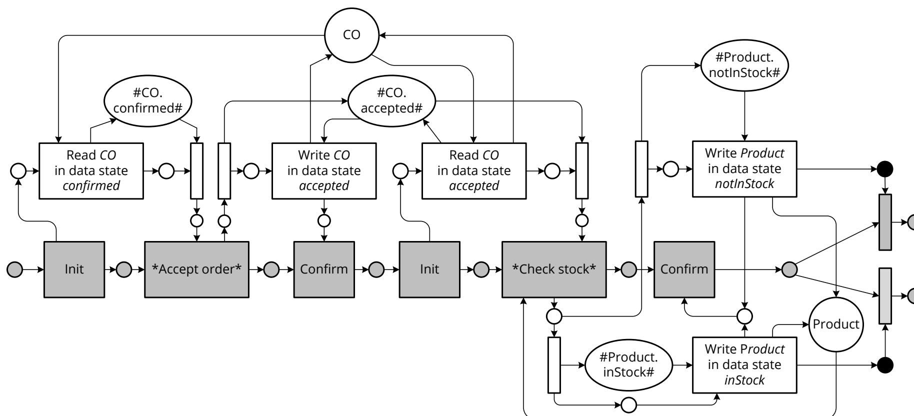


Figure 52: Extract of Petri net representing the process model of the process scenario given in Figures 49 and 50 on page 125.

For each typed synchronization edge, one place is added to the Petri net. If two typed synchronization edges have the same source, the same dependency type, target the same OLC, and if the corresponding target states each have exactly one incoming synchronization edge, both places are merged to one. Similarly, two places are merged, if two typed synchronization edges have the same target, the same dependency type, and origin from the same OLC. Figures 53a and 53b show extracts of synchronized OLCs for both situations.

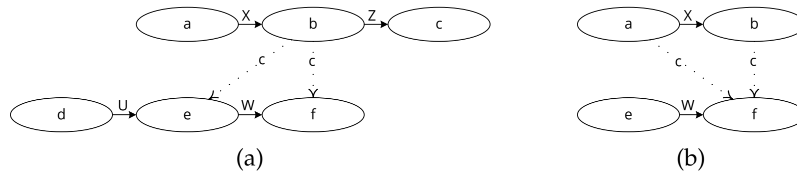


Figure 53: Example fragments of a synchronized object life cycle where places get merged in the resulting Petri net.

The preset of an added place comprises all transitions directly preceding the places representing the source and the target data states of the corresponding synchronization edge. The postset of an added place comprises all transitions directly preceding the place representing the target state of the synchronization edge. For currently typed edges, the postset additionally comprises the set of all transitions directly succeeding the place representing the source state. Figure 54 shows the resulting Petri net extract after creating the additional place (gray colored) for both synchronization edges as given in Figure 53a.

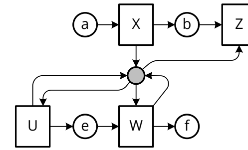


Figure 54: Mapping of currently-typed synchronization edges for the example given in Figure 53a.

For each untyped synchronization edge, one transition is added to the Petri net. If $\bigcap_{se_{\mathfrak{T}_S}}\{src \cup tgt\} \neq \emptyset$ for two untyped synchronization edges, i. e., they share one data state, then both transitions are merged. The preset and postset of each transition comprise newly added places; one for each (transitively) involved synchronization edge for the preset and the postset respectively. Such preset place directly succeeds the transitions that in turn are part of the preset of the place representing the data state from which the data state transition origins. Such postset place directly precedes the transition representing the corresponding source or target transition of the typed

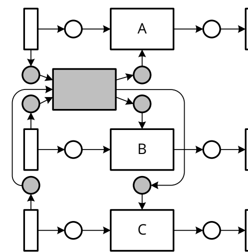


Figure 55: Mapping of untyped synchronization edges.

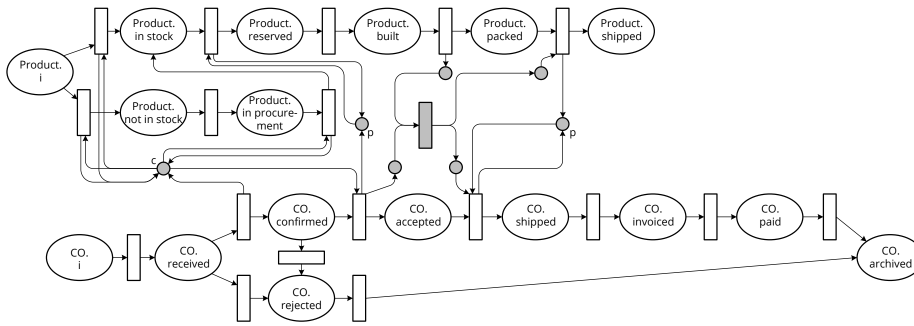


Figure 56: Petri net representation of a subset of the synchronized object life cycle of the process scenario given in Figures 49 and 50 on page 125. The Petri net extract comprises two data classes: *Product* (top) and customer order *CO* (bottom). Gray colored places and transitions represent the synchronization edges.

synchronization edge. Figure 55 visualizes this for synchronization edges $se_{\mathfrak{T}_S,1} = (A, B)$ and $se_{\mathfrak{T}_S,2} = (B, C)$. Thereby, transitions *A*, *B*, and *C* belong to different object life cycles. The gray colored places and transition have been added to the Petri net.

Figure 56 shows an extract of the Petri net for the synchronized object life cycle given in Figure 50 on page 125 comprising data classes *Customer Order (CO)* and *Product*. The gray colored places and transitions represent the synchronization constructs.

In the remainder of this chapter, we refer to the Petri net derived from the process model as *process model Petri net*, we refer to the Petri net derived from the synchronized OLC as *object life cycle Petri net*, and we refer to the Petri net containing the composition of the process model Petri net and the OLC Petri net as *integrated Petri net*.

2—Petri net integration. First, data states occurring in the OLCs but not in the process model need to be handled, i. e., avoidance of enablement, to ensure deadlock free integration of both Petri nets, since otherwise, they could get activated in the OLC part but not utilized in the process part and block its execution. We add one place p to the Petri net, which handles all not occurring states, i. e., avoids execution of these paths. Let each q_i be a place representing such not occurring data state. Then, the preset of each transition t_j being part of the preset of q_i is extended with place p, if the preset of t_j contains a data state which postset comprises more than one transition in the original Petri net mapped from the synchronized object life cycle.

Figure 57 visualizes this procedure for the data state transitions between the initial state and states *received*, *confirmed*, and *rejected* of data class *CO*. The added place *p* is highlighted in black. Data state *rejected* is not utilized in the process modes and therefore, the corresponding state needs to be handled in the Petri net. The transitions representing the data state transitions $\mathtt{received} \longrightarrow \mathtt{rejected}$ and $\mathtt{confirmed} \longrightarrow \mathtt{rejected}$ are part of the preset of the place representing data state *re-*

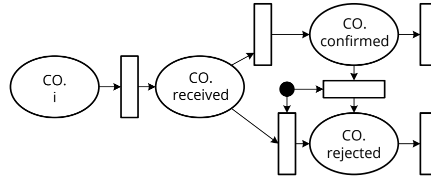


Figure 57: Avoiding enablement of data states in the Petri net derived from the OLCs that are not utilized in the process model. In the given example, state *rejected* is not utilized and since all requirements apply, an additional place (marked black) is inserted into the Petri net. This place becomes part of the preset of both data state transitions targeting state *rejected*.

jected. The preset of the first transition is data state *received* while the preset of the second transition is state *confirmed*. The places of both states contain postsets of size greater 1. Thus, we add the black place to the Petri net and extending its postset with both transitions.

Let $S' = \bigcup_{l \in L} S_l$ and let $S^* \subseteq S'$ be the set of all data states used in the process model of the process scenario. Further, let each data state $s \in S'$ being represented by one place $q \in P$, where P denotes the set of all places of the Petri net. Then, we formally define the handling of not utilized data states as $\forall s \in S' \backslash S^* : \forall t \in \bullet s : \exists q \in P : q \in \bullet t \wedge q \in S' \wedge |q \bullet| > 1 : p \in \bullet t, p \in P$. Place p is added to the Petri net if the condition holds true at least once and p is the same place for each data state where the condition is fulfilled.

Each data state represented as place in the Petri net mapped from the object life cycle consists of a control flow and a data flow component as visualized in Figure 58 with *C* and *D*. Within the integrated Petri net, the control flow component is responsible for the flow of the OLC and the data flow component is responsible for the data flow in the process model. The integration of both Petri nets follows four integration rules (IR), distinguishable with respect to read (IR-1, IR-2, and IR-3) and write (IR-4) operations. These integration rules use the data flow component of data state places.

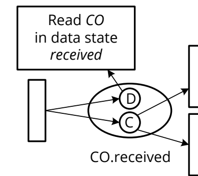


Figure 58: Internal structure of data state places.

(IR-1) A place p from the object life cycle Petri net representing a data state of a data class to be read by some activity in the process model is added to the preset of the transition stating that this data node (object) is read in this specific data state, e. g., the preset of transition *Read CO in data state confirmed* is extended with the place representing data state *confirmed* of data class *CO* (cf. Figure 59a), and

(IR-2) a new place q is added to the integrated Petri net, which extends the postset of the transition stating that the data node (object) is read in the specific data state and which also extends the preset of each

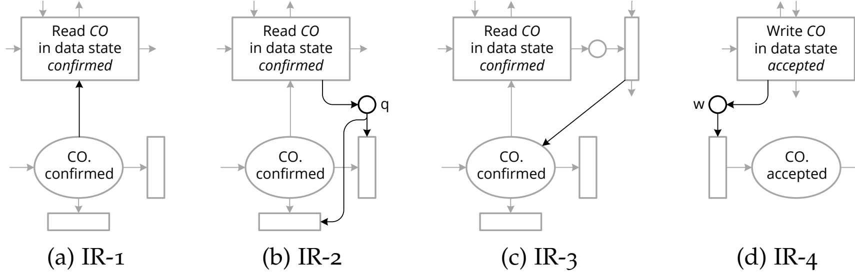


Figure 59: Integration rules to integrate the process model Petri net and the object life cycle Petri net. The gray colored constructs are part of the original Petri nets while the black colored constructs are added for Petri net integration. IR-1 and IR-2 are applied for each read, IR-4 is applied for each write, and IR-3 is additionally applied for each read of nodes (objects) of one data class with no succeeding write of another state for the same class by the same activity.

transition being part of the postset of place p, e. g., the place connecting transition *Read CO in data state confirmed* and the two nop transitions succeeding the place labeled *CO.confirmed* (cf. Figure 59b).

(IR-3) If a node (object) of some data class is read in some data state by an activity but this activity does not write a node (object) of the same data class, then the read data state is added to the postset of the no operation transition that is the postset of the place that in turn is in the postset of transition *Read <class> in data state <state>* that was added to the postset of the read state by IR-1, e. g., *CO.confirmed* is added to the postset of the corresponding nop transition succeeding transition *Read CO in data state confirmed*.

(IR-4) Let v be a place from the object life cycle Petri net representing a data state of a data class to be written by some activity in the process model. Then a new place w is added to the integrated Petri net, which extends the preset of each transition being part of the preset of v and which also extends the postset of the transition stating that the data node (object) is written in the specific data state, e. g., the place connecting the nop transition preceding the place labeled *CO.accepted* and the transition *Write CO in data state accepted*.

Let each data state $s \in S_l$ of all object life cycles $l \in L$ of the synchronized object life cycle $\mathcal{L}$ be represented by one place $p \in P$ in the integrated Petri net. Then, we require rules IR-1 to IR-4 to hold for the integrated Petri net as follows.

(IR-1) $s' \in \bullet t$ with $s' \in P$ representing $s \in S$ and t stating to read data state s of the corresponding data node (object),

(IR-2) $\exists p \in P : p \in t \bullet \wedge \forall u \in s' \bullet : p \in \bullet u$ with s', t being the place respectively transition from (IR-1),

(IR-3) $s' \in t' \bullet$ such that $t' \in q \bullet \wedge q \in t \bullet$ if and only if $\bullet h = t \wedge h \bullet = t$ with $h \in P$ being the corresponding semaphore place, $t' \in \mathcal{T}, q \in P$, and s', t being the state respectively transition from (IR-1), and

(IR-4) $\exists p \in P : p \in \bullet t \wedge \forall u \in \bullet s' : p \in \bullet u$ with s' being the place in the object life cycle Petri net representing the data state s to be written and t representing the transition in the process model Petri net stating this write of s.

3—Workflow net system. We aim at checking correctness by soundness checking. Since soundness was introduced for workflow net systems [195, 331], we will map the integrated Petri net to a workflow net system. As discussed in Section 3.3, workflows nets are Petri nets with a single source and a single sink place and they are strongly connected after adding a transition connecting the sink place with the source place (cf. page 52). Workflow net systems are workflow nets with an initial marking. Next, we describe this mapping. Targeting the requirements for workflow nets, we add two concepts to the Petri net: an *enabler* and a *collector* fragment. The enabler fragment distributes the token from the single source place to each source place of the integrated Petri net. The collector fragment collects the tokens from each sink place of the integrated Petri net routing them into a single sink place. Thereby, source and sink places of the process model as well as the object life cycles, some typed synchronization places, the semaphore places and the place handling not occurring data states must be considered. Initially, the enabler fragment consists of the single source place directly succeeded by a transition y and the collector fragment first consists of a transition t preceding the single sink place.

The postset of transition y of the enabler comprises all places representing an initial data state of some OLC and the source place of the process model Petri net. The preset of these places is adapted accordingly[1]. For each distinct data class of the process scenario, one place p_i and one place q_i are added to the collector. Each place p_i has transition t as postset. Then, for each final data state of some OLC, a transition u_i is added to the collector. Each transition u_i has as preset the place representing the corresponding final data state and some place q_i referring to the same data class as the final state does. The postset of a transition u_i is the corresponding place p_i also referring to the same data class. Additionally, a transition z succeeded by one place is added to the collector. The place's postset is transition t. The preset of z is the sink place of the process model Petri net. The postset of z is extended with all places q_i. Figure 60 shows the current state of adding enabler and collector fragments for an abstract example with a process model containing one activity a, a start node s, and an end node e and an OLC containing an initial state i that is followed by one out of two final states $o1$ or $o2$ via transition $f1$ and $f2$ respectively. Thereby, state $o1$ is not utilized in the process model. Representation of data utilization in the process model Petri net is omitted for readability reasons.

1 Generally, we assume that addition of one element a to the preset (postset) of another element b implies the addition of b to the postset (preset) of a.

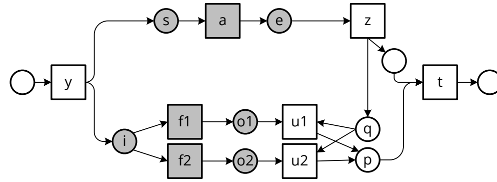


Figure 60: Abstract example for enabler and collector fragment addition (white colored) after handling of source and sink places of the process model and the object life cycle (gray colored). In this example, the semaphore place and data utilization in the process model Petri net are omitted for readability reasons.

Second, the synchronization places need to be considered. If a typed synchronization edge involves the initial state of some OLC as source, then the corresponding synchronization place is added to the post-set of transition y of the enabler fragment. For all synchronization edges typed previously, the postset of the corresponding synchronization place is extended with transition t of the collector. If a currently typed synchronization edge involves a final state of some OLC as source, then the corresponding synchronization place is added to the preset of the transition u_i of the collector fragment that corresponds to that final state.

Next, the semaphore places need to be integrated. Therefore, for each semaphore place, the preset is extended with transition y from the enabler and the postset is extended with transition t from the collector fragments. Finally, the place, handling data states of the object life cycles that do not occur in the process model, is connected to the collector. The preset of that place is extended with transition z The postset of that place is extended with transition t. Figure 61 shows the current state of adding enabler and collector fragments for the mentioned example after the final step; the black place represents the place handling not occurring data states. We omitted the semaphore place for the used data class since we also omitted the process model's data utilization for readability reasons (see above).

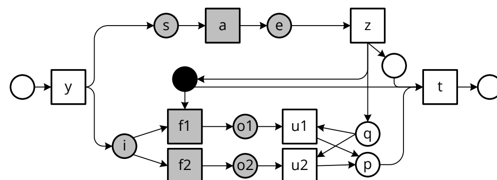


Figure 61: Abstract example for final enabler and collector fragment addition (white colored) as extension to Figure 60. The black colored place represents the place that handles data states not utilized in the process model. In this example, the semaphore place and data utilization in the process model Petri net are omitted for readability reasons.

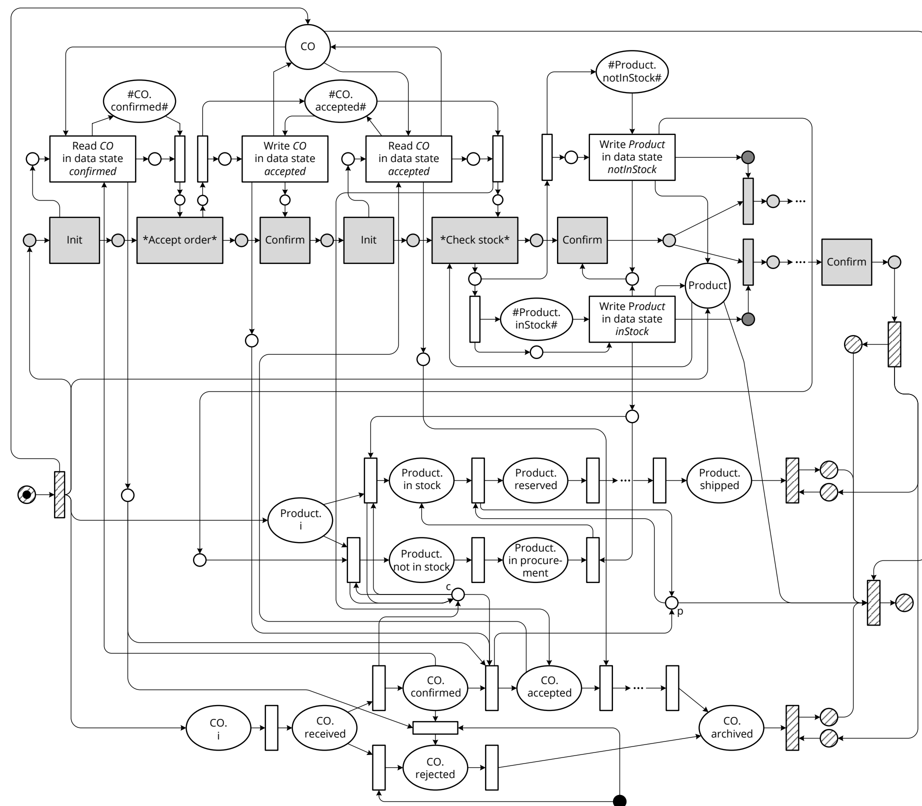


Figure 62: Extract of the workflow net system representing the process scenario given in Figures 49 and 50. The extract comprises activities *Accept order* and *Check stock* of the process model (gray colored), the utilized states of data classes customer order *CO* and *Product*, and the corresponding enabler and collector fragments (shaded).

This concludes the addition of the enabler and collector fragments resulting in a workflow net integrating the process model and the synchronized object life cycle. Now, connecting sink and source node, the workflow net is strongly connected. A workflow net system consists of a workflow net and some initial marking. The workflow net is given above and the initial marking puts a token into the single source place and nowhere else. Figure 62 shows an extract of the integrated workflow net system representing the process scenario given in Figures 49 and 50. It contains activities *Accept order* and *Check stock* of the process model, some states of data classes *CO* and *Product* and their synchronization, and the corresponding enabler and collector fragments.

4—Soundness checking. We utilize soundness checking of the integrated workflow net system to identify control flow and data flow errors. Assuming control flow correctness, deadlocks and livelocks in the net system indicate data flow errors. Then, if the net system satisfies the soundness property [331], no contradictions between the process model and the synchronized object life cycle exist and all data states presented in all OLCs are implicitly or explicitly utilized in the process

model, i. e., all paths in the OLCs may be taken. If the net system satisfies the weak soundness property [195], no contradictions between the process model and the synchronized object life cycle exist but some of the data states are never reached during execution of the process model; for instance, data state *rejected* of class *Customer Order (CO)* in the given process scenario.

Figure 63 presents the complete workflow net system for the process scenario given in Figures 49 and 50. As indicated above, the process scenario does not satisfy the notion of weak conformance, since this net system neither fulfills the soundness nor the weak soundness property. It deadlocks for two reasons. First, transition *Read CO in data state confirmed* (see blue highlight) as predecessor to transition **Ship order** will never be enabled, because it requires a token in place *CO.confirmed* (see green highlights) but this token already advanced to place *CO.accepted* and state *confirmed* is not reachable from state *accepted* in the OLC of data class *CO*. Second, the workflow net system deadlocks when trying to write either data state *inStock* or state *notInStock* of class *Product* (see red highlights). With respect to Figure 50, both states are only allowed to be written, if the customer order *CO* is in state *confirmed* (see purple highlight), which cannot be the case, since the input data node of class *CO* is in state *accepted* and – as already indicated – there does not exist a path from state *accepted* to state *confirmed* in the corresponding OLC. In case, control flow inconsistencies would appear, places and transitions representing the control flow would cause the violation allowing to distinguish between control flow and data flow issues.

TOOL SUPPORT. The soundness checking can be applied with tool support. Tools like *LoLA*[2] [301, 375] take a workflow net as input and provide the result whether the net is sound or not. In the negative case, the tool can provide a counterexample showing where, e. g., the deadlock, occurs. For implementation of the weak conformance computation, the four step algorithm resulting in the integrated workflow net system would need to be implemented in addition to a mapper, that matches the interface of the chosen soundness analyzer, e. g., LoLA.

VALIDATION. The described approach reliably decides about weak conformance of a process scenario. It takes process model and object life cycle Petri nets as input, combines them with respect to specified data dependencies, and transforms the resulting integrated Petri net into a workflow net system. Assuming the input Petri nets have sound workflow net representations, the output workflow net as integration result will also be sound, if our four-step approach does not interfere with process model and object life cycle correctness.

Since soundness checking is a proper technique to analyze the behavior of models being transformed into workflow net systems, we need to

2 Low Level Analyzer

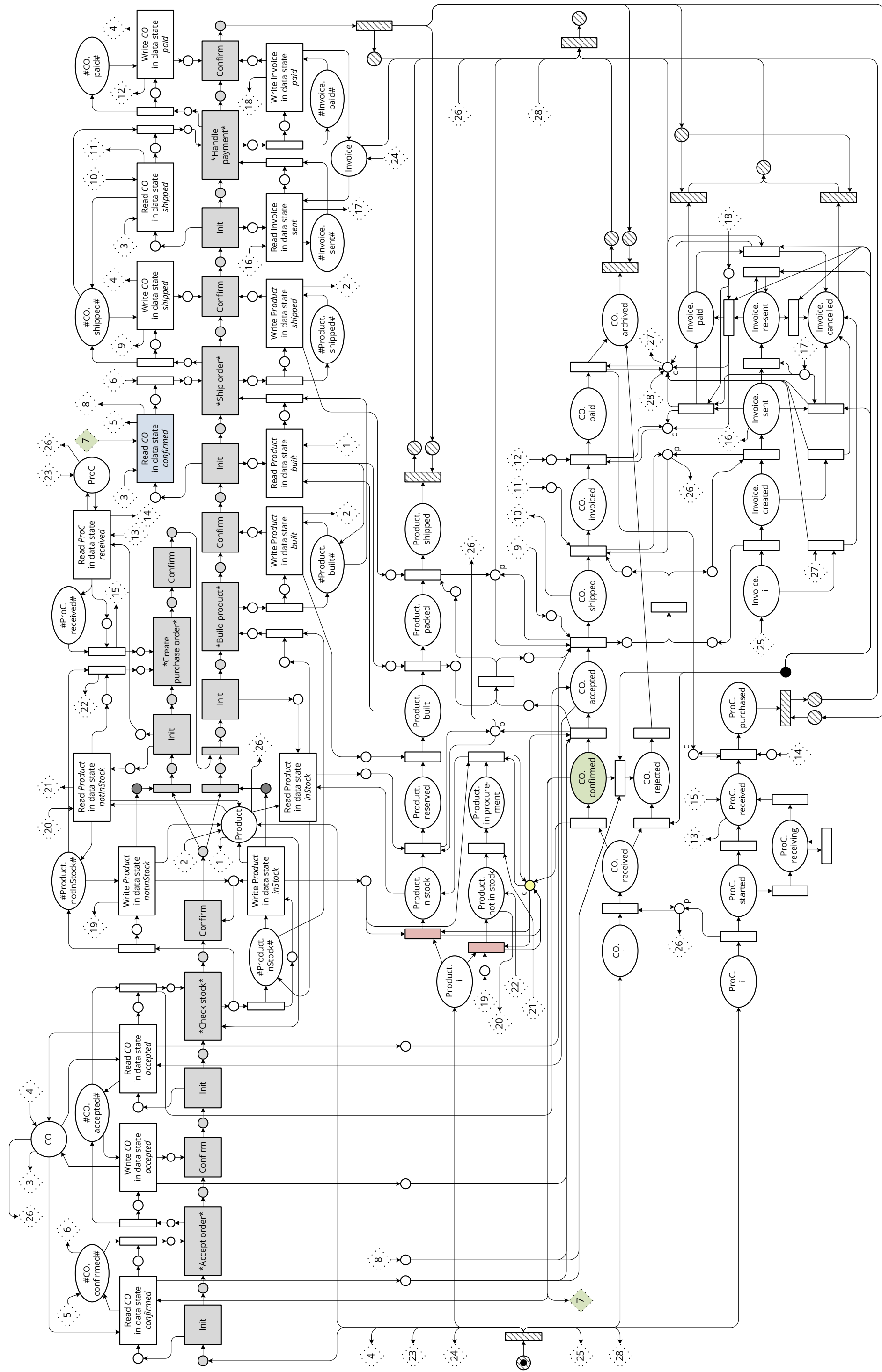


Figure 63: Workflow net system representing the process scenario given in Figures 49 and 50.

argue about the consistency of each of the single steps of our approach. The Petri net mapping of the control flow [80] is consistent as proven by the authors. The mapping of data dependencies as described in Section 4.7 adds places and transitions allowing exactly the data operations specified in the process model without introducing deadlocks or livelocks that are not induced by modeling errors. Each of the rules adds a sound,connected fragment of the process model to the corresponding Petri net. The mapping of single OLCs to a Petri net follows a definition from van der Aalst [334] and produces sound Petri nets by design as long as the originating OLCs are correct.

Indeed, adding the synchronization edges does change the behavior of the object life cycles, but only by adding restrictions required and explicitly modeled by the stakeholder through synchronization edge specification such that the order of states for each single object life cycle is not changed; i. e., the induced behavioral changes are intended and inconsistencies shall not be avoided at this stage but be identified during soundness checking. The integration of the object life cycle Petri net (consisting of multiple OLCs) and the process model Petri net adds *no operation* transitions and places that represent the dependencies between control flow and data flow. Assuming compatible control flow and data flow behavior, the integration increases the number of tokens and synchronizes both nets and proper execution is not affected by these additions.

Finally, the integrated Petri net is transformed into a workflow net system by adding further no operation transitions and places – the enabler and collector fragments – that are connected to the source and sink places of the process model and the object life cycle Petri nets without changing the behavior of any of them. Each single OLC as well as the process model get directly enabled (token in the initial place). After finishing process execution and traversing through all OLCs until some final data state, the tokens existing in the net are collected and put into the single sink place of the workflow net system. Thus, they do not change the behavior of the process model and the OLCs, i. e., they do not influence the result. Assuming control flow correctnes, the workflow net system deadlocks only if the process model requires state changes being not compatible with some object life cycle – and this is expected for notifying about weak conformance violation.

6.3 Correction of Process Scenarios

Identifying violations in process scenarios is only the first step. Secondly, equally important, these violations need to be corrected. A violation can be caused by multiple issues. In this section, we concentrate on correcting violations induced by data flow and refer the reader to related work, e. g., [114], for the correction of control-flow-based violations. For the process scenario discussed in this chapter (see Figures 49

and 50 on page 125 for the visual representations), we already showed two types of data flow violations: synchronization issues and incompatible data states. In total, we identified four types of data flow violations (DFV) that will be discussed in the remainder of this section:

(DFV-1) concurrency issues,
(DFV-2) incompatible data states,
(DFV-3) synchronization issues, and
(DFV-4) modeling errors (see Figure 51 on page 127 for examples).

In literature, multiple data flow errors are reported, e. g., [14, 293, 316]. Some of them target process model underspecification and thus are not considered faulty with respect to our approach, if a consistent process state can be reached from the current one; if it cannot, one of the four mentioned types is sufficient to describe why not. The remaining ones refer to one of the mentioned categories DFV-1 to DFV-4. Most of these existing works only introduce means to identify these errors; [14] also allows automatic resolution of the identified issues. Many existing works in the domain of business process management – for correcting control or data flow – as well as in related domains as software correctness and bug fixing [5, 6] rely on automatic correctness to reduce complexity for the user. Thereby, all these approaches focus on a single level, e. g., software bug, control flow, data flow, and provide some correct result that is not necessarily the optimal one but resolves the issue. In contrast, we consider three different levels for model correction: control flow, data flow within the process model, and data flow imposed by outside models – the object life cycles. This leads to an highly increased complexity in terms of possible corrections. Similarly to the existing approaches, we can provide some solution that resolves an identified issue. However, most violations can be resolved on two or even all three of these levels such that the probability is reduced to chose a correction aligning to the users' intentions. Therefore, we will introduce a semi-automatic approach where we provide the user with ranked options to correct a specific violation and she has to finally choose which is the appropriate one.

Before discussing the four data flow violation types, we will discuss generic violation representation and generic means to correct data flow violations that then may be applied to the violation types. A violation representation contains information about the affected activity of the process model, the affected data class with the affected data states, and context information as, for instance, involved synchronization edges and the data access type. Thereby, we distinguish control flow and data flow violation representations. Generally, we utilize key-value pairs where the key represents the affected entity in the workflow net system and where the value comprises a list of violations with each list item containing information relevant for the specific violation. For control flow violations, the key is the *affected activity* represented by a gray colored transition in Figure 63. One list item of the violations contains the

following entries from which some may be undefined: directly preceding control flow node, input data nodes, and output data nodes. For data flow violations, the key is the *affected data class* (white colored in Figure 63). One list item of the violations contains the following entries from which some may be undefined: data state, access type, activity (that collectively reference an involved data node in the process model), directly preceding data state, synchronization edge, and concurrency issue.

The entries input and output data nodes are lists of tuples each specifying the data class and the state of the corresponding data node. The entry synchronization edge is a list of tuples each specifying the source state, the target state (both fully qualified), and the synchronization type. The entry access type is an enumeration *{read, write}*. The entry concurrency issue is of type Boolean specifying whether the deadlock arises from a semaphore place. The remaining entries and the keys are of data type string.

For the process scenario discussed in this chapter, two data violations are identified. These are be represented as

CO – {dataState: {confirmed}, accessType: {read}, activity: {Ship order}, precDataState: {}, syncEdge: {}, concIssue: false} and

Product – {dataState: {inStock}, accessType: {write}, activity: {Check stock}, precDataState: {i}, syncEdge: {(CO.confirmed, Product.inStock, currently)} , concIssue: false}, {dataState: {notInStock}, accessType: {write}, activity: {Check stock}, precDataState: {i}, syncEdge: {(CO.confirmed, Product.notInStock, currently)} , concIssue: false}.

These violations are summarized in a key-value-store based on which we may highlight critical parts of the corresponding process model and object life cycles (see Figure 63). The key-value-store can be build from the results of a utilized soundness analyzer that provides counterexamples in terms of blocking paths or affected states. The deadlock location is directly given while the remaining entries are to be calculated from the context. The actual calculations are out of scope for this thesis.

This directly leads to possible correction mechanisms (CM) which can be applied to a process model (CM-2 to CM-4), a synchronized object life cycle (CM-5 to CM-8), or both (CM-1). These transformations are:

(CM-1) change of control flow, i. e., rearranging activities in the process model or data states in an OLC,

(CM-2) removal of the affected activity and all data nodes accessed by this activity from the process model,

(CM-3) removal of the data node causing the violation,

(CM-4) change of the state of the data node causing the violation,

(CM-5) creation of additional state transitions in an OLC,

(CM-6) removal of synchronization edges,

(CM-7) creation of new synchronization edges, and

(CM-8) change of synchronization edges.

The decision about the transformations actually applied to the process scenario have to consider several aspects. The transformations must not introduce new violations to the process scenario except the stakeholder explicitly allows this. Such interference possibility is required for guideline changes etc. while unintentional violations shall be avoided. Correcting one violation may lead to new inconsistencies that existed before but have not been detected in the process scenario or may also impact other violations and their correction. Further, in many cases, several options resolve a violation while some of them may also remove intentional constraints. The removal transformations (CM-2), (CM-3), and (CM-6) often help solving conflicts but shall be avoided if other transformations also lead to success. Following these aspects, one may automatically propose a set of transformations to correct a process scenario. But due to the complexity of choosing the correct ones, e. g., impact on other process scenarios, company guidelines, or level to be corrected (see above), these transformations shall not be applied automatically but proposed to a stakeholder or process expert, who eventually decides which transformations shall be applied. Hence, the correction process is semi-automatic such that we provide a ranked list of proposals to resolve each violation to support the selection process. Next, we discuss the data flow violation types and the application of correction mechanisms to them.

Concurrency issues

We refer to concurrency issues (DFV-1), if two activities on different branches in one *AND* block access the same data class and are enabled from control flow and data flow but the execution of one activity prevents the other activity to be executed afterwards. These violations require a change of the control flow in the process model by sequentializing the affected activities, if possible (cf. CM-1). Figure 64 sketches two exemplary situations for data class customer order *CO* having the object life cycle presented in Figure 50 on page 125. In Figure 64a, activity *A* changes data state *shipped* to state *paid* while activity *B* requires to read the customer order in state *shipped* to create an object of class *Invoice* in state *created*. In Figure 64b, activity *A* again changes data state *shipped* to state *paid* while activity *C* changes state *shipped* to state *invoiced*. While the first situation can be harmonized by putting activity *B* and *A* in this sequence (see Figure 64c), the second concurrency issue cannot be resolved by activity reordering, because the violation is also caused by some state incompatibility that needs to be handled. Alternatively to reordering activities in the scope of CM-1, the removal of affected activities or data nodes are valid correction mechanisms.

Concurrency violations are indicated by deadlocks arising from a semaphore place. Thus, an identified violation is of type concurrency issue, if the corresponding entry of a list item referring to some key is *true*. To handle all concurrency issues, a list of violations where the specific entry is set to *true* is extracted. In case there are multiple concurrency issues for one data class, they could be distinguished either

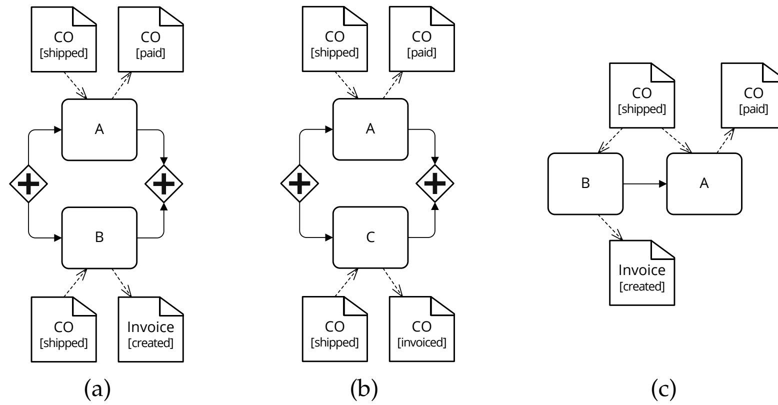


Figure 64: Two examples (a) and (b) for concurrent data access of class *CO* from which (a) can be resolved by activity sequentialization (see (c)) while (b) cannot be resolved due to the overlapping paths in the object life cycle.

by the affected data states or by the behavioral relations between the involved activities. Due to the option of omitting data states in the process model (cf. underspecified process model), we utilize the behavioral relations. Hence, the concurrency relations between the activities of one concurrency violation are calculated (cf. [166, 167]) before they get clustered accordingly. For each cluster, an appropriate order of activities is computed based on their data dependencies utilizing the notion of weak conformance, e. g., activity *B* followed by activity *A* in Figure 64c. If no appropriate order is found, the concurrency violation is accompanied by some other type of violation as, for instance, data state incompatibility (cf. Figure 64b). Additionally, also CM-2 or CM-3 might be applied to remove the violating modeling constructs.

Incompatible data states

We refer to incompatible data states, if a read of an object cannot happen, because the object is not in the appropriate state nor may reach it, or if a write of an object cannot happen, because the object cannot transition into the appropriate state from its current one, i. e., there does not exist a path from the current state to the expected state. Referring to the integrated workflow net system, data state incompatibility occurs, if a *read <object> in data state <x>* transition cannot be executed due to a missing token in the corresponding place *Object.state* or if the unlabeled transition preceding the place *Object.state*, where state refers to the state to be written, cannot be executed due to a missing token in a data state directly preceding the state to be written.

Theoretically, incompatible data states can be corrected with transformations CM-1 to CM-5, but they are not applicable under all circumstances and may require additional changes. Therefore, applicability of each transformation needs to be checked. A change of the control flow in the OLC (cf. CM-1) may require an adaptation of synchronization edges utilizing transformations CM-6 to CM-8. Considering the given

process scenario discussed in this chapter, the violation with respect to data state *confirmed* can be resolved by exchanging this state with state *accepted* resulting in the following order of states: *received*, *accepted*, *confirmed*. Checking the synchronization edges, they need to be adapted as well. This can either be identified by another iteration of weak conformance checking or directly considered in the correction process. The synchronization edges indicating that *currently confirmed* is a precondition for reaching data states *in stock* or *not in stock* respectively must be changed to *currently accepted* synchronization edges. Additionally, the edge indicating that *accepted previously* is the precondition for state *reserved* of class *Product* can be changed to *confirmed previously* to cover the state exchange. However, the violation correction would be correct without this last adaptation.

Note, these changes on how objects of class customer order *CO* are processed is applied generally. This may also influence other process scenarios utilizing the same synchronized object life cycle (or a subset/-superset of this one). Hence, such changes need to be thought through well and a different correction strategy may be preferred; e. g., changing the state of the corresponding input data node to *accepted* as discussed above. Nevertheless, changing the order of data states in the object life cycle is an effective method of resolving data state incompatibilities.

Removal of violating modeling constructs (cf. CM-2 and CM-3) basically always works with above mentioned drawbacks. Changing the state of the violating data node (cf. CM-4) requires the existence of a data state s_n on the path from the last correctly used date state s to the next correctly used one s' such that $\exists\, s_n \in S_l : s \overset{\sigma_1}{\longmapsto} s_n \wedge s_n \overset{\sigma_2}{\longmapsto} s'$. Each data state of object life cycle l fulfilling this statement may be chosen to exchange the given data state. Often, adding the missing state transition to the OLC (cf. CM-5) solves the existing issue but also changes the behavior of the corresponding data class which then may not conform to general guidelines or restrictions. Therefore, this type of transformation is to be used – similarly as CM-1 with respect to data states – with care. In the given process scenario, adding a state transition from *accepted* to *confirmed* allows satisfaction of weak conformance with respect to data class *CO*. However, this allows to reject an already accepted customer order which is not intended in the initial version of the OLC.

Synchronization issues

We refer to synchronization issues, if state incompatibility is induced by missing tokens in OLCs that refer to data classes other than the class of the node (object) to be read or written. These issues may be resolved by applying any of the transformations introduced above. While transformations CM-1 to CM-5 are analogous to incompatible state resolution, we now discuss transformations CM-6 to CM-8. Removing the synchronization edge (cf. CM-6) is always an option under the expense of relaxing made restrictions as discussed for the other removals. As long as guidelines do not change or an existing restriction is not mod-

eled wrongly, the removal is not a valid option for violation resolution. Changing the type of a synchronization edge (cf. CM-8) also relaxes made restrictions but retains the basic temporal dependency. However, sometimes, e. g., with guideline and regulation changes, also the temporal dependency might be subject to change as available with CM-6 or a change of a synchronization edge's source or target in the scope of CM-8. Considering the given process scenario, relaxing the *currently* constraint to *previously* for the synchronization edges from state *confirmed* of class *CO* to states *in stock* and *not in stock* respectively of class *Product* resolves the synchronization issue explained above. It allows to reach both states making the process model conforming to the OLC of class *Product*. Creating new synchronization edges (cf. CM-7), i. e., introducing new constraints, is usually no transformation resolving issues alone but it may be used in cooperation with further transformations.

Modeling errors

Modeling errors often result in incompatible data states (cf. Figure 51 on page 127). Therefore, the violation resolution is done analogously. For all violation types holds that the actual model transformations need are calculated based on the given process and object life cycle models as sketched above. In cases where none of the discussed error types can be matched, the violation correction proposals have to be calculated on a case basis. Hence, we shift the responsibility towards the stakeholder or process expert by only stating which modeling constructs induce the violation and require their input for correction.

Violation type identification

Type identification for some violation is done by analyzing the given key-value-pairs. If the concurrency issues entry is set to true, violation DFV-1 (concurrency issues) applies. If the concurrency issue entry is set to false and some synchronization edges are specified, violation DFV-3 (synchronization) applies. Otherwise, i. e., if the concurrency issue entry is set to false and no synchronization edge is specified, violation DFV-2 (incompatible data states) applies. Following this identification procedure, the violations identified in the process scenario utilized in this chapter are of types *incompatible data states* (DFV-2) and *synchronization issues* (DFV-3) respectively.

Ranking

After identifying transformations that allow resolution of found violations of the introduced types, they need to be combined such that all violations can be corrected. The choice on which transformations are actually applied to each violation depends on the stakeholder or process expert. Supporting this decision process, we provide a generic ranking of the correction mechanisms for each type. This allows reduction of applicable transformations based on correction mechanism choice. However, usually, there exist multiple transformations referring to one correction mechanism that are applicable to correct a given violation. Here, the stakeholder or process experts get presented these options in before and after comparisons; the actual decision is manually.

Table 3 provides a ranking of the top-five of the correction mechanisms CM-1 to CM-8 for the discussed violation types *concurrency is-*

Table 3: Top-five ranking of correction mechanisms (CM) to resolve a violation induced by a specific type.

Violation type / rank	1	2	3	4	5
Concurrency	CM-1	CM-3	CM-2	—	—
Incompatible states	CM-4	CM-1	CM-5	CM-3	CM-2
Synchronization	CM-8	CM-4	CM-1	CM-6/7	CM-5

sues, *incompatible data states*, and *synchronization issues*. The ranking is based on above discussions to the various correction mechanisms for each violation type. For concurrency issues, CM-1 is the only mechanism without removing modeling constructs. Hence, it is listed on rank 1. CM-3 follows on rank 2 and CM-2 on rank 3, because removing a single data node has less impact on a process model than removing an activity.

Resolving incompatible data states best utilizes CM-4 as this has the least impact in terms of change to the process scenario while providing good solutions; changing the state of a data node to an appropriate state does not affect other modeling constructs. The second rank goes to CM-1 which also provides good results for violation resolution but impacts other modeling constructs and may require additional changes in the process scenario. CM-5 is the easiest method to correct data state incompatibilities probably imposing new means of data processing due to new data state transitions. These data state transition also impact other process scenarios utilizing the same OLC and thus, CM-5 is ranked third. CM-3 and CM-2 follow on ranks 4 and 5 respectively with the same reasons as above.

Synchronization issues can be resolved with all introduced correction mechanisms; Table 3 presents the five best-suited mechanisms: CM-8, CM-4, CM-1, CM-6 and CM-7, and CM-5. These are followed by CM-6, CM-3, and CM-2 in this order. Ranks 7 and 8 are reasoned analogously to above. Rank 6 goes to CM-6, because the removal of a synchronization edge does not change the data states reached during process execution as the removal of a data node does (cf. CM-3). One may argue that removing a synchronization edge removes regulations which one may have to follow for legal purposes. Following this argumentation, one may switch ranks 5 and 6 putting CM-3 on rank 5. However, we follow the argumentation referring to the impact on the process scenario and thus decided for the mentioned ranking.

The best mechanism to correct synchronization issues is CM-8. The change of synchronization edges and especially an edge's type does not affect other modeling constructs and often relaxes enablement requirements sufficiently to avoid deadlocks during process execution. On ranks 2 and 3, the correction mechanisms CM-4 and CM-1 follow with the same argumentation as for incompatible data states. CM-4 is ranked

second, because changing the data state to align with synchronization requirements may impose further changes on the process scenario and thus, has a higher impact on other modeling constructs compared to CM-8. CM-7 does not resolve issues alone, since it introduces further constraints. However, the combination of CM-6 and CM-7 allows to restructure the synchronization behavior in a synchronized object life cycle. We rank this combination fourth. Finally, rank 5 goes to CM-5. The impact on other process scenarios is higher than for the combination of CM-6 and CM-7 that only introduces new synchronization behavior instead of completely new data behavior in terms of data state transitions.

The correction mechanisms, their ranking, and derived transformations are only proposals provided to stakeholders and process experts who need to review the proposals and decide for specific corrections to be applied to the process scenario. As aforementioned, multiple transformation proposals may exist for one violation type and correction mechanism. These transformation proposals are presented in random order with all having the same rank.

6.4 RELATED WORK

In this chapter, we concentrate on integrated scenarios, which incorporate process models and object life cycles. Similar works exist as discussed in Section 4.8. Thereby, especially the works [176, 292, 364] are to be mentioned since they also deal with process scenario correctness in terms of data utilization. In contrast to these works, we remove the assumption that both representations – process model and object life cycle – must completely correspond to each other. Instead, we set object life cycles of data objects as references that describe what can be utilized by process models. Additionally, we allow data synchronization by including synchronization edges into correctness computation. Table 2 on page 126 summarizes the main differences that will get detailed below.

[153] provides an overview on correctness issues in workflow management. Thereby, the issues and probable solutions are discussed. [7] formalizes these correctness issues based on set and graph theory to allow further formal analysis or process model correctness. Correctness, or compliance, in process models mostly refers to checks of the process model with respect to a defined rule set containing, for instance, business policies. Thereby, compliance checking can be directed forward or backward to ensure compliant process models or to identify violations after process execution using, for instance, process mining respectively. Forward compliance comprises compliance by design, model checking at design-time, or enforcement of compliant process execution [89, 295]. The approach presented in this chapter allows to check for compliance at design-time by using completed process models to check for con-

formance with data objects before they are executed manually or by a process engine.

The field of compliance is well researched, especially with respect to control flow compliance [3, 10, 12, 116, 117, 286]. [13] introduces means to check for compliance with respect to data dependencies, e.g., an object is required to be in a certain state for activity execution. These dependencies need to be specified with explicit rules in BPMN-Q [8], which are transformed to temporal logic for the actual compliance check. Compared to the approach described in this chapter, the authors require the process engineer to explicitly state data dependency rules instead of checking against object life cycles. Indeed, the object life cycles may be represented by a set of data dependency rules which number rapidly grows with the size of the synchronized object life cycle. Furthermore, [187, 188] applies compliance checking to object-centric processes by creating process models following the this paradigm from a set of rules These rules most often specify control flow requirements.

However, some works extended process model verification with data capabilities. [176] introduces compliance between a process model and an object life cycle of one data object used in the process model as the combination of object life cycle conformance (all data state transitions induced in the process model must occur in the object life cycle) and coverage (opposite containment relation). [364] introduces conformance checking between process models and product life cycles, which in fact are object life cycles because a product life cycle determines the data states and state transitions allowed to be performed upon a product, i.e., a data object. Compared to the notion of weak conformance, both notions do not support synchronization between multiple data objects and both set restrictions with respect to data object manipulations: All data object state changes specified in the object life cycle need be covered in the process model with explicit corresponding state changes by reading the source states and writing the target states of these state transitions.

[325] extends workflow nets with data capabilities – so-called WFD-nets – by annotating transitions with data objects to be read, written, and deleted and by annotating edges with guards for data-based decision taking. To allow conformance checking of the utilized data, the authors discover nine data flow anti patterns describing violations to data object utilization. These patterns are translated to CTL and LTL temporal logic, which allows to utilize existent model checker. In [304], extends this work by defining a soundness notion for WFD-nets which is proven to work. WFD-nets support the same data operations as we describe in this chapter plus adding an explicit deletion of data objects but the authors abstract from data states and the correlating object life cycles which we also consider for conformance checking. Eshuis [94] uses a symbolic model checker to verify conformance of UML activity diagrams [244] considering control and data flow perspectives while

data states are not considered in his approach. The data flow was also verified in further process description languages as, for instance, WS-BPEL [228]. [104] introduces the concepts of dual workflow nets separating the control flow and the data flow into separate models. Control flow is modeled as common workflow net and the data flow is modeled as directed graph indicating the order in which transitions are allowed to be executed. Then, the authors check whether the order induced by the workflow net transitions is covered by the directed graph of the data flow.

Weber et al. [366] annotate the activities of process models with semantic information (logical preconditions and effects from domain ontologies) and define a formal execution semantics for these annotated process models. The conformance checking addresses the overall process behavior and checks whether the control flow behavior and the annotated semantic behavior do not contradict. As shown by the authors, pre- and postconditions of activities in terms of data objects can be annotated to the activities, although annotation complexity rapidly grows with the size and number of corresponding object life cycles because all allowed and forbidden states of the data objects need to be mentioned. Further, the object life cycles need to be transformed into an ontology.

[354] follows the assumption that a single model integrating all aspects of lower level aspect models (use cases and object life cycles) is a proper way of verifying consistency and correctness in the field of engineering complex systems. Therefore, they synchronize the independently created aspect models and check afterwards whether each aspect model is derivable from the integrated model via projection. Similarly, state chart composition techniques [108, 235, 258, 328] can be used to integrate the Petri nets derived from the process model and synchronized object life cycle (soundness checking technique, see Section 6.2) followed by a soundness check for conformance computation. Following these approaches would require to redesign the mapping to the Petri nets such that the Petri net representation of the process model and the object life cycles do contain composable places or transitions meaning that they contain places or transitions being equal.

One aspect of process mining is to check for conformance after process execution (backward compliance) by comparing observed event traces of an event log with the corresponding process model (generated or modeled) to identify deviations, which represent violations. Therefore, the event log and the process model must be aligned by relating events to process model constructs and vice versa. [352] replay the process model identifying all paths and check whether these allow the observed event traces. This approach is about making moves in two models, here the process model and the log, which may be transferred to making moves in a process model and the corresponding object life cycle(s) leading to another technique checking for weak conformance

besides the ones presented in this chapter. Similarly, [64] utilize an adapted version of the A* algorithm [65] to identify the similarities and deviations between the event log and the process model. Thereby, the authors include information about resources and data utilization into the event logs allowing to check for data conformance. In comparison to our techniques residing in the area of forward compliance, this approach may complement our techniques to validate data conformance after process execution.

Bridging the gap between the activity-centric and object-centric process paradigm, Fahland et al. [102] use proclets [344] to model business artifacts [54]. They provide a technique to check for conformance of object-centric processes containing multiple data objects probably existing in m:n relationships to each other. This is mapped to an interaction conformance problem, which can be solved by decomposition into smaller sub-problems, which in turn are solved by using classical conformance checking techniques. This approach is based on execution logs as well.

In the area of object-centric processes as, for instance, business artifacts, several verification techniques have been introduced to ensure correct execution of such artifact system of procedural as well as declarative nature [25, 29, 75, 115]. Thereby, most techniques utilize temporal logic and model checking as in our third technique (see autorefsubsec:computationTemporalLogic). But their approaches cannot be applied to activity-centric process models, because the corresponding formalizations are specifically catered for artifact systems.

Identifying and dealing with data anomalies is closely related to process correctness. In [14], the authors describe data anomalies utilizing data states and provide a similar view on the topic as the authors in [176, 292] do with their compliance notions. Data anomalies refer to situations where a data object is not provided in the correct data state to a task or interleaving behavior cannot be achieved due to implicit data state dependencies. The authors implicitly use data flow information to identify the inconsistencies instead of deriving an object life cycle to check the process model against. Finally, the authors focused on deadlock identification as well as deadlock resolution. [293] also introduces data anomalies, which partially overlap with the ones mentioned above. Here, data anomalies refer to, for instance, lost data objects, missing data objects, and redundant data objects. Basically, these data anomalies appear in underspecified process models. [316] refines these data anomalies and provides an algorithm for identification. In [281], the authors add the concepts of optional read and optional write of data objects to data anomalies and therefore, introduce an extended set of them, which they correlate to the behavioral relationships in the process model. Finally, they implemented the identification of data anomalies in the context of an overall consistency checker for process models. The identification of data anomalies relates to conformance checking, but

mostly abstracts from data states and never uses the concept of reachability. The deadlock resolution adds to process model correction, but tackles only a subset of possible violations. Speaking in soundness terminology, deadlock avoidance ensures termination of the net but not proper termination, i. e., there might be places with a marking after the final node has been reached. [314] introduces a model-checking approach to determine data anomalies in a process model. Therefore, the authors map the process model to a Petri net, generate anti-patterns formula that are then used for model checking using computational tree logic (CTL) [92]. [197] introduces a graph traversal algorithm to identify data flow errors (e. g., wrong branching behavior, concurrency issues, data losses in loops, or non-executability of activities) avoiding proper execution of a business process. All these approaches have in common to identify data flow errors arising within the process model and to not consider outside models as, for instance, object life cycles. Furthermore, they do not provide correction mechanisms as we do.

In this chapter, we introduced a similar type of compliance as in [176] or [364]. In contrast to [176], [364] and we set object life cycles to be the reference, i. e., we assume they are correct. Hence, we can restrict ourselves to check for conformance only. There, we rely on data state reachability, instead of working with direct data state transitions as both other notions do. Additionally, in contrast to [364], we allow transitions to be specified in an object life cycle not to be used in the process model. We also abstract from data anomalies such that we allow their existence – and therefore underspecified process models – and check whether the implicit data state transitions can be covered by the object life cycle. Discussing the distinctions between the existing data conformance notions and the newly introduced one on weak conformance shows that it is not useful and not applicable to characterize a process model as non-conforming to an object life cycle because of missing or abstracted data information. Consequently, we provide important new insights on process scenarios and allow statements whether a process scenario, containing underspecified or partially specified process models, satisfies the given rules in terms of object life cycles such that the scenario can be enacted with some refinement work.

Abstracting from internal process behavior, business process architectures (BPAs) are used to visualize the dependencies between different processes [79, 86, 287]. Dependencies between multiple processes mainly arise from message flow and from data flow. While most approaches deal with the message flow level, [87] introduces means to build a BPA based on data dependencies. Having constructed a BPA, it can be formally analyzed [88] – including correctness of process interactions and process model ordering. In contrast to these works, we do not check for correct inter-dependencies between multiple process models but for correct utilization of data within a process scenario that leads to correctness checks for single process models.

Summarizing, compared to all other approaches in the domain of process correctness, we provide an integrated approach to check for behavioral correctness of control flow and data flow as well as for consistency between process models and object life cycles of data classes utilized in the process model.

6.5 CONCLUSION

In this chapter, we presented an approach for the integrated verification of control flow correctness and data correctness using soundness checking considering dependencies between multiple data classes, e. g., an order is only allowed to be shipped after the payment was received but needs to be shipped with an confirmed invoice in one package. We introduced the notion of weak conformance which checks whether data object accesses (read and write of data nodes) specified in a process model contradict to object life cycles of the utilized data classes. Weak refers to the capability of also checking abstracted or underspecified process models, where some data state transitions, i. e., data object manipulations, are given implicitly only.

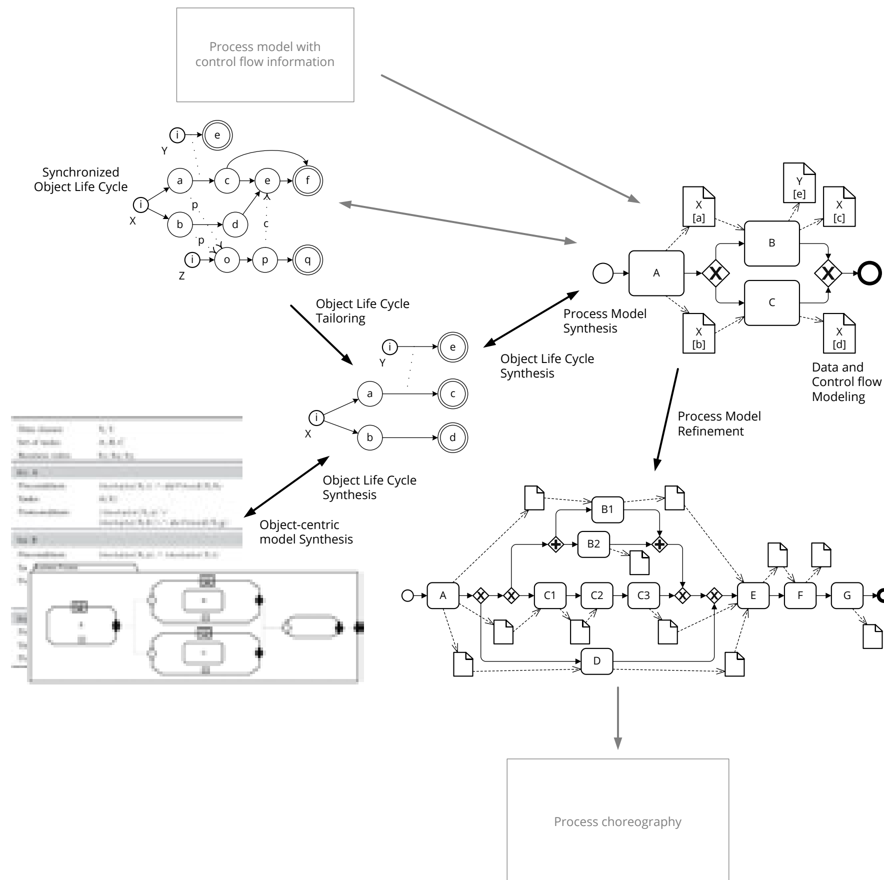
Whether a process scenario, a formalism combining a process model and the corresponding synchronized object life cycle, satisfies the notion of weak conformance is computed by soundness checking. Both, the process model and the synchronized object life cycle are mapped to Petri nets, the resulting nets are combined by matching places representing data states, and this integrated Petri net is extended by enabler and collector fragments that provide a workflow net. This workflow net is finally checked for soundness. Since the workflow net comprises control flow and data information, control flow correctness and data correctness can be checked by a single, integrated correctness check.

With respect to the places or transitions causing soundness violations, errors can be related to control flow or data issues. Revealed violations can be highlighted in the process model and the synchronized object life cycle to support correction. Since violation detection is only the first step towards a correct process scenario, we also introduced means to correct the process scenario. Thereby, we concentrated on data corrections since control flow correctness is already well researched. A process scenario may suffer from four types of data violations: concurrency, incompatible states, synchronization, and modeling errors with modeling errors usually referring to state incompatibilities. Based on a catalog, we can provide ranked proposals for adaptation of either model resulting in a correct process scenario from data point of view. Finally, the stakeholder must decide for the corresponding correction.

Soundness checking and correction on Petri net level allows application of our approach to a multitude of process description languages due to existing mappings to Petri nets [190].

7 MODEL TRANSFORMATIONS

This chapter is based on results published in [210, 211].



Transformations of models representing the behavior of and the actions taken in business processes allow (i) switching the view of the corresponding business process or (ii) adapting the model with respect to the level of details presented within one view towards the needs of the stakeholders. Therefore, we distinguish model transformations between different views of a business process, e. g., the object-centric and the activity-centric view, referred to as *inter-view transformations* and transformations to change the abstraction level of one view (*intra-view transformations*).

In practice, two process modeling paradigms – and therewith two major views on the business process – are of major importance: activity-centric process models (ACPs) and object-centric process models (OCPs)

as introduced in Chapters 3 and 4. ACPs use activities and control structures (gateways) as first class modeling concepts and regard data nodes in specific data states as pre- and postconditions for activity enablement or as main decision indicator at exclusive gateways. The usage of data nodes of one data class in different data states in combination with multiple activities allows derivation of an object life cycle (OLC), which describes the manipulations performed on that data class [97, 292].

Typical representatives are the industry standard Business Process Model and Notation (BPMN) and event-driven process chains (EPCs). OCPs [54, 173, 237] regard data classes and their OLCs as first class modeling concepts and multiple data objects of several classes synchronize on their data state changes, i. e., data state changes in different OLCs need to be performed together (cf. object life cycle synchronization). These dependencies are then represented in visual models as, for instance, using the Guard-Stage-Milestone (GSM) approach [141], business rules [384], or the Case Management Model and Notation (CMMN) [245]. Thereby, the order of activities is usually not modeled explicitly but can be extracted by analyzing the OCP. In the remainder of this chapter, we utilize the business rule representation as introduced in Definition 3.7. The different visualizations utilize the same concepts such that slightly adapted algorithms can be applied to different visualizations as well.

Currently, both process modeling paradigms compete for application – object-centrism and activity-centrism – although both of them may represent the same business process with different focus points – data vs. control flow. In fact, both paradigms are useful in different scenarios. OCPs are proved to be useful if the process flow follows from data objects as, for instance, in manufacturing processes [231]. In contrast, in many domains, e. g., accounting, insurance handling, and municipal procedures, the process flow follows from activities, which need to be executed in a predefined order. Here, ACPs are in favor. However, all scenarios may gain from representing the business process in both views allowing easy understanding of activity and data flow. Additionally, a transformation between both views allows to start modeling in one view, changing the views, and continue modeling in the other view depending on which information shall be added to the model representing the business process.

In this chapter, in the context of inter-view transformations, we utilize a synchronized OLC as mediator between both views to introduce a transformation between them because both views are tightly coupled to OLCs as shown in literature [98, 170, 176, 186]. Thereby, existing research utilizes one paradigm, OCP or ACP, and transforms this one into an OLC or back basing on specific and different process model description languages as well as different assumptions with respect to model concept utilization. This prevents an unified and consistent transformation cycle, which we will introduce in Sections 7.1 to 7.4

for generic activity-centric process models as defined in this thesis and object-centric process models in business rule representation.

While inter-view transformations derive one view from another one, intra-view transformations may require information from other views with probably different levels of detail: Based on the information given in an OLC, a corresponding process model can be changed accordingly. Vice versa, an OLC can be changed based on information given in a process model. In the context of this thesis, we focus on activity-centric process models and therefore, we limit the introduction of intra-view transformations to ACPs. In Sections 7.5 and 7.6, we introduce two intra-view transformations (i) allowing to enrich a process model representing a process scenario with information given in the synchronized object life cycle of the same process scenario (see Definition 4.16) and (ii) allowing to tailor an OLC towards the information required in a process model, both referring to the same process scenario. Thereby, tailoring refers to model reduction. Additionally, business process model abstraction (BPMA) and object life cycle refinement are additional intra-view transformations which are out of scope for this chapter. BPMA is discussed in [131, 209] independently from OLC information while we consider the synchronized object life cycle as the single source of truth (cf. Chapter 6) such that no process model shall contain more information about data manipulation than this one for a given process scenario. Thus, adapting a synchronized OLC after manual refinement of the corresponding process model remains as only use case for OLC refinement which can be handled by tailoring the initially given synchronized object life cycle with respect to the refined process model.

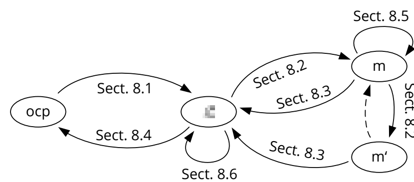


Figure 65: Inter- and intra-view transformations of models referring to one process scenario. Transformations represented by a solid line are discussed in detail in the respecting section while transformations represented by a dashed edge are trivial and thus are not discussed.

Altogether, inter-view as well as intra-view transformations lead to aligned views on the corresponding process scenario, e. g., between the OCP and ACP or between the synchronized OLC and the ACP. Figure 65 provides an overview about inter-view and intra-view transformations described in this chapter and in which section each specific step is discussed. Our transformation algorithms also support underspecified and abstracted process models (see also discussion in Chapter 6), i. e., gaps in data manipulation specification are assumed to be handled externally. We introduce seven transformations applicable to generic process models as defined in Definition 3.7 for OCPs and

$\mathfrak{pm} = (N, D, Q, bp, \mathfrak{C}, \mathfrak{F}, type_g, DCF)$ with $DCF = (\xi)$ as subset from the ACP definition given in Definition 4.10; i. e., no specific process process description must be used but the required information must be present in the description language of choice. Section 7.1 describes the transformation of an object-centric process model ocp into a synchronized object life cycle $\mathcal{L}$, which in turn can be transformed into an activity-centric process model pm. This transformation as well as the enrichment of pm with data class attribute information towards pm′ is described in Section 7.2. For this enrichment, information about the attributes in terms of results for functions instate and defined are required from the business rules specified in the object-centric process model. For the opposite transformation from pm′ to pm, we do not discuss an algorithm in detail. Summarized, the attribute information needs to be removed by utilizing an Extensible Markup Language (XML) parsing strategy as, for instance, defined in the BPMN specification [243]. Section 7.3 discusses the transformation of an activity-centric process model pm or pm′ with attribute definition into a synchronized object life cycle $\mathcal{L}$. Section 7.4 introduces the final inter-view transformation to transform a synchronized object life cycle $\mathcal{L}$ into an object-centric process model ocp by considering information about attributes taken from the corresponding activity-centric process model pm′ with attribute definition. Afterwards, Section 7.5 discusses the refinement of an activity-centric process model based on OLC information while Section 7.6 finally describes the tailoring of an object life cycle based on a given ACP. In the subsequent sections, we discuss all transformations in text form with corresponding examples while we present the detailed algorithms in Appendix A.

7.1 OBJECT-CENTRIC PROCESS MODEL TO OBJECT LIFE CYCLE

Given an object-centric process model in business rule representation, a synchronized object life cycle can be derived following nine intertwined steps. The detailed algorithm of this transformation is presented in Algorithm 6 on page 321.

(OSD-1) Initialize the synchronized object life cycle,
(OSD-2) initialize single object life cycles,
(OSD-3) get sets of pre- and postcondition states of business rules,
(OSD-4) add data states to the single OLCs,
(OSD-5) derive data state transition labels by concatenation,
(OSD-6) add data state transitions to the single OLCs,
(OSD-7) specify their dependencies by adding the synchronization edges to the synchronized OLC,
(OSD-8) set initial and final data states in the single OLCs, and
(OSD-9) add these object life cycles to the synchronized one.

First, the synchronized object life cycle gets initialized (OSD-1). Then, for each data class utilized in the given OCP, one single OLC gets initial-

ized as well (OSD-2). Next, the business rules of the OCP get analyzed separately to extract information about data states and the transitions between them (OSD-3 to OSD-7). Given a business rule, the following analysis steps are performed for each utilized data class iteratively. For a given data class, the sets of precondition states and postcondition states are determined (OSD-3) before these data states are added to the corresponding single object life cycle if they are not yet present in the respecting set of data states (OSD-4). The correct OLC is determined via data class matching. After adding the data states to the OLCs, we handle the transitions between these states. For each task involved in the currently analyzed business rule, the labels are extracted and concatenated resulting in a single label (OSD-5). This is required because of the semantics of activity-centric process models, where all actions changing the input data nodes into the output data nodes are comprised by one single activity instead of multiple tasks affecting different state transitions as allowed for object-centric process models in business rules representation. Then, we add a transition from each state used in the precondition to each state used in the postcondition where both states belong to the same data class. This transition is labeled with the created single, concatenated label indicating which collection of tasks imposes the data state transition (OSD-6) – in post-processing, the label can optionally be adapted by the stakeholder.

After processing all data classes involved in the current business rule, synchronization edge specifications are added to the synchronized object life cycle. Therefore, we iterate over all pairs of data state transitions added to some single OLC in the context of the current business rule. If both transitions within such pair refer to different data classes and thus object life cycles, a synchronization edge is added between these transitions indicating that they are executed together. If there exists a data class for which a precondition state is given in a business rule but no postcondition state, a synchronization edge of type *currently* is added to the synchronized object life cycle from this data state to each data state given in the postcondition of the business rule (OSD-7). Synchronization edges of type *previously* are not considered for the view transformation because they do not influence the association of data nodes to activities. While *currently* requires a corresponding data node with the required data state as input to the derived activity, *previously* does not require this. Assuming that the given OCP is correct, *previously* dependencies can be ignored as we do for these transformations. Otherwise, if they are required to be stated, they can be expressed via disjunction of all data states that are reachable from the mentioned state including the mentioned state.

After analyzing all business rules, all required data states and data state transitions are captured in the single object life cycles. In each OLC, the data state without an incoming data state transition, i.e., $\bullet s = \emptyset$, becomes the *initial data state* of that OLC while all data states

Table 4: Rule-based object-centric process model.

Data classes:	CO, Product, Invoice
Set of tasks:	collect, analyze, checkStock, initiateProcurement, stockUp, manufacture, ship, receivePayment, setPayed, archive
Business rules:	b1, b2, b3, b4, b5, b6, b7, b8, b9
b1: Computer retailer receives order from customer	
Precondition:	$\mathtt{instate}(\text{CO}, \text{init})$
Tasks:	collect(CO)
Postcondition:	$\mathtt{instate}(\text{CO}, \text{received}) \land \mathtt{defined}(\text{CO}, \text{CustomerNumber}) \land \mathtt{defined}(\text{CO}, \text{ReceiveDate}) \land \mathtt{defined}(\text{CO}, \text{Products})$
b2: Computer retailer analyzes customer order with respect to completeness and validity	
Precondition:	$\mathtt{instate}(\text{CO}, \text{received}) \land \mathtt{defined}(\text{CO}, \text{CustomerNumber})$
Tasks:	analyze(CO)
Postcondition:	$(\mathtt{instate}(\text{CO}, \text{confirmed}) \lor \mathtt{instate}(\text{CO}, \text{rejected})) \land \mathtt{defined}(\text{CO}, \text{AnalysisDescription})$
b3: Computer retailer checks warehouse stock for product availability	
Precondition:	$\mathtt{instate}(\text{CO}, \text{confirmed}) \land \mathtt{instate}(\text{Product}, \text{init})$
Tasks:	checkStock(CO, Product)
Postcondition:	$(\mathtt{instate}(\text{Product}, \text{inStock}) \lor \mathtt{instate}(\text{Product}, \text{notInStock})) \land \mathtt{defined}(\text{Product}, \text{Customer})$
b4: Computer retailer initiates the procurement of the missing product	
Precondition:	$\mathtt{instate}(\text{Product}, \text{notInStock}) \land \mathtt{defined}(\text{Product}, \text{Customer})$
Tasks:	initiateProcurement(Product)
Postcondition:	$\mathtt{instate}(\text{Product}, \text{inProcurement}) \land \mathtt{defined}(\text{Product}, \text{DeliveryDate})$
b5: Computer retailer stocks-up the warehouse with the received product	
Precondition:	$\mathtt{instate}(\text{Product}, \text{inProcurement})$
Tasks:	stockUp(Product)
Postcondition:	$\mathtt{instate}(\text{Product}, \text{inStock})$
b6: Computer retailer organizes manufacturing of the ordered product	
Precondition:	$\mathtt{instate}(\text{Product}, \text{inStock}) \land \mathtt{defined}(\text{Product}, \text{Customer})$
Tasks:	manufacture(Product)
Postcondition:	$\mathtt{instate}(\text{Product}, \text{built}) \land \mathtt{defined}(\text{Product}, \text{ManufacturingDate})$
b7: Computer retailer ships product to the customer	
Precondition:	$\mathtt{instate}(\text{CO}, \text{confirmed}) \land \mathtt{instate}(\text{Product}, \text{inStock}) \land \mathtt{defined}(\text{Product}, \text{Customer})$
Tasks:	ship(CO, Product)
Postcondition:	$\mathtt{instate}(\text{CO}, \text{shipped}) \land \mathtt{instate}(\text{Product}, \text{shipped}) \land \mathtt{defined}(\text{CO}, \text{ShippingDate})$
b8: Computer retailer invoices the order	
Precondition:	$\mathtt{instate}(\text{CO}, \text{shipped}) \land \mathtt{instate}(\text{Invoice}, \text{init})$
Tasks:	receivePayment(Invoice) setPayed(CO)
Postcondition:	$\mathtt{instate}(\text{CO}, \text{paid}) \land \mathtt{instate}(\text{Invoice}, \text{paid}) \land \mathtt{defined}(\text{CO}, \text{InvoicingDate}) \land \mathtt{defined}(\text{CO}, \text{PaymentDate}) \land \mathtt{defined}(\text{Invoice}, \text{Amount})$
b9: Computer retailer archives order in information system	
Precondition:	$(\mathtt{instate}(\text{CO}, \text{paid}) \land \mathtt{defined}(\text{CO}, \text{PaymentDate})) \lor \mathtt{instate}(\text{CO}, \text{rejected})$
Tasks:	archive(CO)
Postcondition:	$\mathtt{instate}(\text{CO}, \text{archived}) \land \mathtt{defined}(\text{CO}, \text{ArchivalDate})$

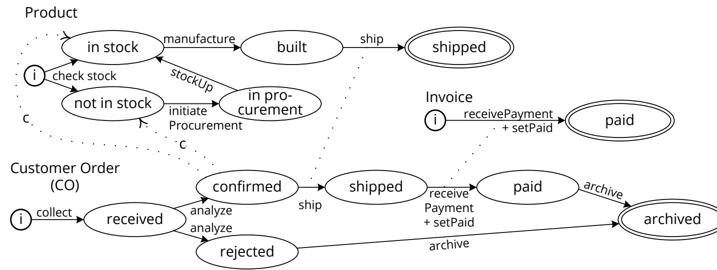


Figure 66: Synchronized object life cycle derived from the object-centric process model given in Table 4.

without an outgoing data state transition, i. e., $s\bullet = \emptyset$, become *final data states* (OSD-8). Finally, each single object life cycle is added to the synchronized object life cycle (OSD-9). The presented algorithm requires full specification of the object-centric process model; i. e., each data class used in the postcondition part of a business rule must also be used in the precondition part of the same business rule.

The object-centric process model in Table 4 consists of three data classes, ten tasks, and nine business rules. Accordingly, in correspondence to the three data classes, the resulting synchronized object life cycle consists of three single OLCs connected by synchronization edges. For instance, based on business rule *b3*, transitions from data state *init* to data states *in stock* or *not in stock* respectively labeled with the task name *checkStock* are added to the OLC of data class *Product*. Additionally, since an object of class *CO* in state *confirmed* is also part of the precondition while no object of this class is part of the postcondition, both transitions of the OLC referring to class *Product* are synchronized with data state *confirmed* in the object life cycle of class *customer order CO* through a *currently* typed synchronization edge. Figure 66 presents the synchronized object life cycle derived from the given object-centric process model and post-processed in terms of transition labels.

7.2 OBJECT LIFE CYCLE TO ACTIVITY-CENTRIC PROCESS MODEL

Given a synchronized object life cycle, an activity-centric process model can be derived following ten intertwined steps with steps SAD-3 to SAD-7 being repeated in a loop. The detailed algorithm of this transformation is presented in Algorithm 7 on page 322.

(SAD-1) Group transitions executed together into combined transitions,

(SAD-2) initialize activity-centric process model with a single start event,

(SAD-3) step through not yet checked control flow nodes of currently existing ACP and identify combined transitions that are enabled in some state of that ACP,

(SAD-4) add placeholder activities for execution paths without further activities,

(SAD-5) add one activity with input and output data nodes for each identified, enabled combined transition,

(SAD-6) add placeholder activities for transitions not considered by SAD-4 and SAD-5,

(SAD-7) add control flow edges and (if required) corresponding gateways based on the combined transitions,

(SAD-8) add a single end event,

(SAD-9) route all paths of the process model towards the single end event, and

(SAD-10) remove the placeholder activities and edges targeting them.

First, the transitions of the given synchronized OLC are grouped with respect to their execution dependencies – equally labeled transitions within a single object life cycle having the same source and different target states and transitions connected via an untyped synchronization edge (more specifically: the transitive closure over all transitions being connected by an untyped synchronization edge), i. e., transitions that are executed together, are grouped into a *combined transition* (SAD-1). In synchronized object life cycles that are derived from object-centric process models, all transitions contained within a combined transition have the same label. However, in practice or considering a modeled synchronized OLC, the labels might be different. In Figure 66, the transitions from *confirmed* to *shipped* and from *built* to *shipped* in OLCs of classes *CO* and *Product* respectively refer both to the same combined transition. Second, the activity-centric process model is initialized with a start event as only modeling construct (SAD-2). Afterwards, the ACP is created iteratively by adding control flow nodes, data nodes, and corresponding edges to the process model until all information from the synchronized OLC is covered by the ACP (SAD-3 to SAD-7).

Analyzing the current status of the ACP, we distinguish the control flow nodes whether they have already been checked for succeeding nodes or not. In the course of the transformation, each control flow node needs to be checked exactly once. Therefore, in each iteration, we start with the control flow nodes that have not been checked yet, i. e., the control flow nodes added to the ACP in the previous iteration. In the first iteration only the start event exists and has not yet been checked. For each such not yet checked control flow node n, we determine the set of combined transitions of the synchronized object life cycle, which are enabled after termination of that node. A combined transition is enabled if and only if all transitions comprised by this combined transition are enabled. A transition is enabled if and only if its source data state s is currently reached (after termination of n) and if there do not exist synchronization edges preventing the enablement, i. e., if all data states referring to other data classes but being the source of a synchronization edge of type *currently* targeting s are also currently reached or

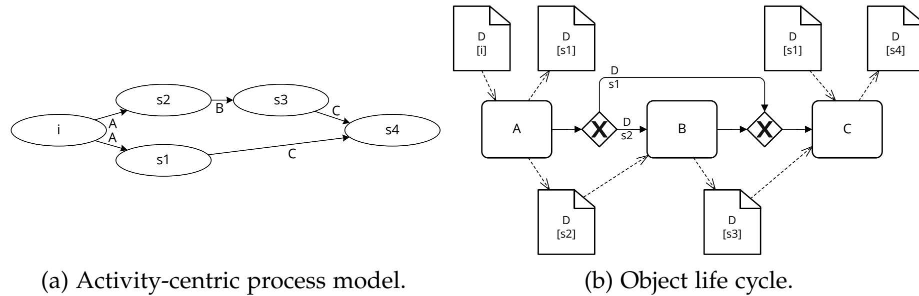
being the source of a synchronization edge of type *previously* targeting s are on a path from the initial state to the one currently reached for the respective data class. Alternatively, a transition is enabled if and only if above statement holds after termination of all not yet checked control flow nodes instead of only n. This is required in cases where multiple parallel branches are supposed to be merged. Each combined transition identified in step SAD-1 is checked whether all contained transitions are enabled (SAD-3).

Assume that n is the single start event of the newly created ACP, then all transitions that change the state of a data object from data state *initial* to some *data state* are potentially enabled. Given the synchronized OLC in Figure 66, corresponding combined transitions are *collect*, *check stock*, and *handle payment*. From them, the latter two combined transitions are not enabled because of the synchronization edges that prevent at least one contained single transition of being enabled. States *confirmed* or *shipped* for objects of class *customer order CO* are not yet reached, since only the initial state is reached upon instantiation of the start node. Thus, both contained transitions of the combined transition *check stock* in the object life cycle of class *Product* and the one contained transition of *handle payment* occurring in the OLC of class *Invoice* are not enabled.

Next, the output data nodes of control flow node n are analyzed. For each output data node representing a data node (object) in some final state of the corresponding OLC, we create a placeholder activity labeled $nop : state$, where *state* refers to the actual final data state (SAD-4). This activity does not contain any data association and gets added to the process model. At the same time, this placeholder activity is marked as *checked for combined transitions* because no more actions are supposed to happen after reaching the corresponding final data state and thus no activities are succeeding this placeholder activity. As final step of the control flow node analysis, node n is marked as *checked for combined transitions*.

After analyzing all unchecked control flow nodes, activities are created and added to the process model to cover the actions imposed by the identified combined transitions. For each identified combined transition, one activity is created. Activities that are created based on transitions that are not combined but having the same action label within a single object life cycle (cf. transitions *archive* in Figure 66) are merged with the subsequent step being applied multiple times to this merged activity. For each transition being contained in the combined transition, the source states and their corresponding data classes are mapped to data nodes that are associated as input data nodes to the created activity while the target states and their corresponding data classes are mapped to data nodes that are associated as output data nodes to the created activity (SAD-5). The created activity also gets assigned an activity label that is the concatenation of all actions' names involved in the combined transition. In post processing, a stakeholder may adapt

the activity label. Considering above example with combined transition *collect* being the only enabled one, an activity is added to the activity-centric process model that has a data node of type *CO* in state *init* as input and a data node of the same type in state *received* as output while the activity is labeled *collect*. In postprocessing, the label can be changed to *Collect order* to adhere to the verb-object-style of activity labeling.



(a) Activity-centric process model. (b) Object life cycle.

Figure 67: Activity-centric process model and object life cycle that show an example case where SAD-6 must be applied.

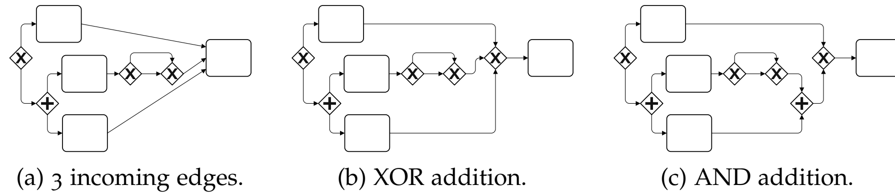
Each non-final data state referring to an output data node of n will eventually occur in some input data node of some subsequent activity. Otherwise, it would have been a final state. In case, an object life cycle contains multiple paths where one consists of directly succeeding transitions being shared with all other exclusive paths (see Figure 67a where the lower path shares transitions *A* and *C* with the upper path which also contains of unique transition *B* while the lower path does not have such unique transition), then this path would be omitted by steps SAD-4 and SAD-5. Consider the OLC in Figure 67a, after execution of combined transition *A*, only transition *B* is enabled resulting in a corresponding sequence. Since data state *s1* is no final state, no placeholder activity leading to an XOR gateway is added such that after execution of *B*, the combined transition *C* is enabled resulting in a sequence of activities *A*, *B*, and *C* with *C* having two data nodes of class *D* with states *s3* and *s1* as input. However, transition *B* and thus the corresponding activity *B* is required to be executed in each case although the actual behavior is an optional execution of *B* as shown in Figure 67b. Therefore, step SAD-6 handles these exceptional cases. For each non-final state that is not utilized as source in some enabled combined transition, a placeholder activity with no label but a data node with the corresponding data state as output is created. This placeholder activity is not marked as checked (in contrast to the other type of placeholder activity). In the example in Figure 67, this is an activity having a data node of class *D* in state *s1* as output. The example can be directly transferred to data state transitions *Analyze* and *Archive* in Figure 66 for transitions *A* and *C* respectively. Transition *B* refers to the combination of transitions *Ship* and *Handle payment*. After adding activity *Analyze order* to the activity-centric process model derived from the OLC, only combined transition *check stock* is

enabled that would result in a single subsequent path although *Analyze order* has two data nodes of the same class *CO* that require alternative paths. Since data state *rejected* is not final and not part of the mentioned combined transition, a corresponding placeholder activity with a data node of class *CO* in state *rejected* is created.

Next, the control flow edges need to be added to the process model to connect the newly added activities (control flow nodes of the process model without incoming nor outgoing control flow edges) with the control flow nodes existing before (SAD-7). Let n denote a control flow node that was checked for combined transitions in this iteration. If the sum of created activities – placeholder activities as well as activities based on combined transitions – in the context of n equals 1, a sequence occurs and thus a control flow edge from n to the created activity is added to the process model. If the sum of created activities is greater than 1, additional gateways need to be added. In case, at least one placeholder activity was created, an XOR gateway is added to the process model. In case, all created activities write data nodes of distinct data classes, an AND gateway is added to the process model. In case, two created activities write data nodes of the same data class, an XOR gateway is added to the process model. Then, control flow edges are added from n to the added gateway and from this gateway to each created activity.

For XOR gateways, data conditions are required to be added to the respecting outgoing control flow edges by assigning a data node to a control flow edge. The class of the data node is determined by identifying the class of nodes that are utilized by some output data node of the checked activity and by some input data nodes if each created non-placeholder activity. The data node corresponding to a determined data class and being output to the activity preceding the XOR gateway and being input to the created activity targeted by a control flow edge originating from the XOR gateway is added as data condition to that control flow edge. For control flow edges targeting a placeholder activity, depending on the type, either a data node being output to the activity preceding the XOR gateway and having the state in the label and having the corresponding final state in the corresponding OLC or the data node being output to the placeholder activity is added as data condition to that control flow edge. Finally, the added gateway gets marked as *checked for combined transitions* as all paths have been processed properly. Again considering the example for n equaling the single start event, a single activity is added to the ACP such that a single control flow node is created connecting the initial start event and the added activity.

Following these steps, in some iteration, an activity may have multiple incoming control flow edges. This is handled before the next iteration of checking control flow nodes for combined transitions takes place. If an activity n has multiple incoming control flow edges, all first



(a) 3 incoming edges. (b) XOR addition. (c) AND addition.

Figure 68: Visualization of iterative addition of join and merge gateways to handle activities with multiple incoming control flow nodes. In (a), the right-most activity has three incoming control flow edges. In the first iteration, the left-most XOR gateway is identified as latest common gateway for all paths. Therefore, in (b) an XOR gateway is added as join for the three paths and becomes a direct predecessor to the mentioned activity. In the second iteration, each pair of paths is analyzed and for the lower pair, the AND gateway is identified as latest common gateway. Therefore, in (c) a corresponding AND gateway is added as merge to the process model.

preceding control flow nodes are determined – one for each incoming control flow edge. Then, for each such determined control flow node, one random trace through the process model starting from the single start event and ending at that control flow node is identified. In these traces, only gateways and the final control flow node are of interest. Therefore, they are reduced accordingly. The reduced traces are analyzed for overlappings in gateways. First, the last gateway existing in all traces is identified. This gateway determines the type of gateway that is added to the process model as predecessor to the control flow node with multiple incoming control flow edges, i. e., n. The control flow edges targeting n are rerouted with the added gateway as new target. Further, a control flow edge connecting the added gateway and n is added. Then, each set of $k-1$ reduced traces (k equals the number of traces analyzed in the last iteration) is analyzed for overlappings in gateways. If there exists another last overlapping gateway that does not precede a gateway already utilized in an earlier iteration, a gateway of the same type is added to the process model as direct successor of the control flow nodes denoting the ends of the affected traces; i. e., let g_1 and g_2 be two gateways where g_1 is the gateway that was identified in a previous iteration and g_2 is the gateway identified in the current iteration, then g_1 must precede g_2 on all paths through the process model. The control flow edges originating from these nodes are rerouted accordingly and a new one is added to connect the newly added gateway with the previous successor of these nodes. These steps are repeated until each pair of reduced traces was analyzed. Figure 68 shows an abstract addition of gateways to a process model based on the described iterative procedure. The addition of join and merge gateways completes step SAD-7.

Above paragraphs describe the steps SAD-3 to SAD-7 undertaken repeatedly until all data state transitions of the given synchronized object

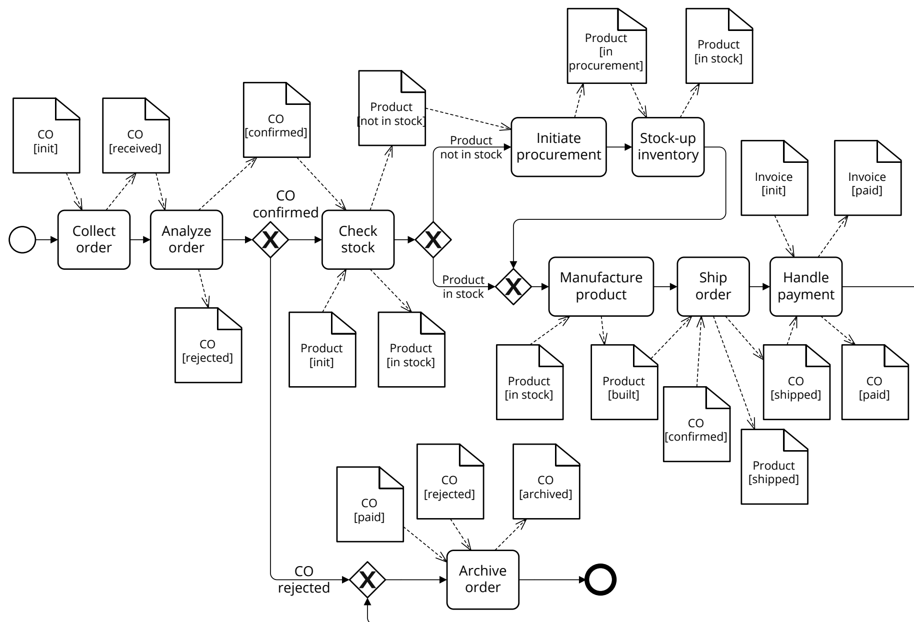


Figure 69: Activity-centric process model derived from the object life cycle given in Figure 66.

life cycle are mapped to activities with corresponding data and control flow additions. The remaining steps now finalize the activity-centric process model and ensure structural soundness by adding a single end event to the process model (SAD-8) to which all paths are routed (SAD-9). If there exists exactly one control flow node with no outgoing control flow edge apart from the added end event and it is no placeholder activity, then this node and the end event are connected by a control flow edge. If the single control flow node is a labeled placeholder activity, its incoming control flow edge is rerouted to the end event. If there exist multiple control flow nodes with no outgoing control flow edge apart from the added end event, then an XOR gateway is added to the process model. Afterwards, control flow edges targeting an activity with *nop:state* label are rerouted such that the added gateway is the new target. The remaining control flow nodes (except the end event) are connected to the added gateway by new control flow edges. Finally, step SAD-10 comprises some model finalization actions. A control flow edge connecting the added gateway and end event is added to the process model. Activities labeled with *nop:state* are removed from the process model. They can be removed safely as none of them has an incoming or outgoing control flow edge. Unlabeled placeholder activities, the associated output data nodes, and control flow edges targeting them are removed while a control flow edge originating from such activity is rerouted such that it connects the preceding XOR gateway (source of removed edge previously targeting the activity) and the succeeding control flow node (original target of edge).

Figure 69 shows the activity-centric process model resulting from the synchronized object life cycle given in Figure 66. After adding activity *Collect order* as discussed above, activity *Analyze order* is added in sequence as well with a data node of type *CO* in state *received* as input and two data nodes of the same type in states *confirmed* and *rejected* respectively as output. After termination of activity *Analyze order*, the combined transition comprising all transitions labeled *check stock* is enabled. This and the two output data nodes of *Analyze order* result in two activities; one is labeled *Check stock* while the other one is an unlabeled placeholder activity. Activity *Check stock* gets a data node of class *Product* in state *i* as input since this is the source of the combined transition and it gets two data nodes of the same class in states *in stock* and *not in stock* as output. Additionally, due to to the currently-typed synchronization edges from *confirmed* to *in stock* and *not in stock* respectively, a data node of class *customer order CO* in state *confirmed* is added as input to activity *Check stock*. Since two activities have been created for the checking of activity *Analyze order* from which one is a placeholder activity, an XOR gateway is added as direct successor of activity *Analyze order*. This XOR gateway has two outgoing control flow edges: one leading to the placeholder activity and one leading to activity *Check stock*. The data conditions are retrieved as follows: Activity *Check stock* as only created activity has a data node of type *CO* in state *confirmed* as input while the same data node is output to activity *Analyze order*. Thus, the control flow edge connecting the XOR gateway and activity *Check stock* is associated with that data node as data condition. For the control flow edge leading to the placeholder activity, its output data node is associated to this control flow edge as data condition. Subsequent steps processing leads to the activity-centric process model as presented in Figure 69.

Attribute information

Object-centric process models also present information about attribute utilization (cf. functions `instate` and `defined`) as part of the process specification. In activity-centric process models, the data state function maps values to attributes of data objects and therewith, it states implicitly which attributes must be defined in conjunction with which data state. However, it does not show the attributes explicitly required to exist in specific circumstances, i. e., the ones of major importance for initializing or completing a task. Therefore, it does not represent the attribute utilization information specifically; see Section 8.3 for a proposal in this regard. Considering BPMN as example for a description language of an ACP, it allows to represent the process model using XML through a mapping specified in the standard specification. Listing 1 shows the representation of a data node (design-time) respectively data object (run-time). For each data node (object), a unique identifier, its class, and its multiplicity status is described in the tag’s properties.

```
1 <dataObject id="" class="" isCollection="">
2   <dataState id="" name="" />
3 </dataObject>
```

Listing 1: Condensed XML representation of a data node (object) in BPMN.

Introducing the attribute information into BPMN requires an XML extension as shown in Listing 2. Therefore, BPMN provides an extension mechanism called *extension points* that allows to specify for each XML tag *extension elements*. For each given attribute, we add an according tag to the extension elements of a data node (object) and specify the attribute's unique identifier, its name, and the data type as properties while the actual value is the content. Generally, an empty value indicates that an attribute is not defined while an existing value indicates successful attribute definition. The properties refer to the concepts defined in Section 4.1 for data nodes and objects that are considered in the transformation process.

```
1 <dataObject id="" class="" isCollection="">
2   <extensionElements>
3     <attribute id="" name="" type="">value</attribute>
4     <attribute id="" name="" type="">value</attribute>
5     <attribute id="" name="" type="">value</attribute>
6     ...
7   </extensionElements>
8   <dataState id="" name="" />
9 </dataObject>
```

Listing 2: Extended XML representation of a data node (object) comprising attribute information.

This added attribute information comprises all attributes defined in the corresponding data model. To explicitly visualize the attributes required for one process step (task execution and one data state transition), we additionally add the Boolean property *required* to each attribute tag as shown in Listing 3.

```
1 <attribute id="" name="" type="" required="">value</attribute>
```

Listing 3: Specifying data attribute changes required for data state transitions.

The information about attribute requirements is not captured in the synchronized object life cycle. Therefore, enriching an activity-centric process model with attribute information requires input from the corresponding OCP to extract the attribute information from there. In the future, this information could be attached to the data state transitions of the OLC and the synchronization edges between single OLCs in case an attribute of a data class, having the source of the synchronization edge, is required although the state of that class will not be changed by the respecting action or task. This would also require a slight but straightforward extension of the algorithm discussed in Section 7.1. For this chapter, we extract the information from the OCP. This information is only used if it is present. Otherwise, the corresponding parts of the algorithm are ignored resulting in an unchanged activity-centric process model that is incomplete from the point of view that attribute information should be part of the model, i. e., for all data nodes, there does not exist any attribute tag with property *required* set to *true*. For attribute information extraction (AIE), three steps have to be undertaken.

(AIE-1) Extract XML representation of activity-centric process model,
(AIE-2) for each data node utilized in the ACP, extract the attribute types from business rules of the corresponding object-centric process model, and
(AIE-3) extend the XML representation of each data node according to above introduced structure (structure is shown for BPMN but can be generalized accordingly).

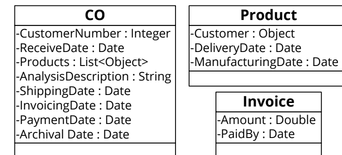


Figure 70: Data model.

For step AIE-1, we use standard XML extractions provided by most graphical process description languages. For BPMN, we utilize the one described in the specification [243] and add the general attribute information as given in the data model (see Figure 70). For each data node (object) given in the XML structure, the required attributes are extracted from business rules of the corresponding object-centric process model. A read data node refers to the precondition and a written data node refers to the postcondition of the business rule containing the corresponding data state in the respecting condition. The correct business rule is identified through the data nodes and their data states as well as the type of access, i. e., read or write. For instance, activity *Analyze order* reads a data node of class *CO* in state *received* and writes data nodes of this class in states *confirmed* and *rejcted* respectively (see Figure 69). Therefore, the business rules shown in Table 4 are analyzed for business rules that require data class *CO* in state *received* as precondition and that have the other two data nodes contained in the postcondition. In the example, this holds true for business rule *b2*. Based on business rule *b2* from Table 4, the data node of class *CO* that is read by activity *Analyze order* in Figure 69 requires the attribute *CustomerNumber* to be newly defined; on model level, it has an empty value. Both data nodes written by activity *Analyze order* require the attribute *AnalysisDescription* to be defined. Listing 4 shows the resulting XML structure for the read data node.

```
1  <dataObject id="uuidDO" class="CO" isCollection="false">
2    <extensionElements>
3      <attribute id="uuidA1" name="CustomerNumber" type="Integer" required
         ="true">1235</attribute>
4      <attribute id="uuidA2" name="ReceiveDate" type="Date" required="
         false">12.01.2015</attribute>
5      <attribute id="uuidA3" name="Products" type="List<Object>" required
         ="false"></attribute>
6      <attribute id="uuidA4" name="AnalysisDescription" type="String"
         required="false"></attribute>
7      ...
8    </extensionElements>
9    <dataState id="uuidDS" name="received" />
10 </dataObject>
```

Listing 4: Example XML representation of a data node (object) of class *customer order CO* read by activity *Analyze order* upon activity enablement.

7.3 ACTIVITY-CENTRIC PROCESS MODEL TO OBJECT LIFE CYCLE

Given an activity-centric process model, a synchronized object life cycle can be derived following eight intertwined steps. The detailed algorithm of this transformation is presented in Algorithm 8 on page 324.

(ASD-1) Identify all data classes utilized in the ACP,
(ASD-2) initialize the synchronized object life cycle,
(ASD-3) initialize all single object life cycles,
(ASD-4) extract all traces through the activity-centric process model,
(ASD-5) analyze the control flow nodes of each trace (specifically activities and XOR splits) and add the input and output data nodes distinctly to the corresponding single OLC,
(ASD-6) specify data state dependencies by adding synchronization edges to the synchronized OLC,
(ASD-7) set final data states in the single OLCs, and
(ASD-8) add these object life cycles to the synchronized one.

First, all distinct data classes utilized in the activity-centric process model are identified (ASD-1) before the synchronized OLC (ASD-2) as well as a single object life cycle consisting of the initial state for each identified data class (ASD-3) are initialized. Additionally, as part of step ASD-3, the distinct data states are determined for each data class and associated with (added to) the corresponding sets of data states of object life cycles. The next step requires to extract all traces through the activity-centric process model from the start event to the end event; loops are reduced to a single trace (ASD-4). Then each trace is handled separately by steps ASD-5 and ASD-6. First, for each data class utilized on the chosen trace, an object life cycle specific *collection* is created that will be used to store data states relating to data nodes of the corresponding data class. The initial state of each single OLC is added to the corresponding collection. Then, all control flow nodes of the trace are checked for their type and get processed as follows.

Activity

If the control flow node is an activity, all input data nodes are received. Afterwards, if not existing yet, the data state of an identified input data node is added to the corresponding single object life cycle and the data state transitions are specified; one transition from each entry of the corresponding OLC specific collection to the added data state is added to the object life cycle if source and target are different data states. This is repeated for all input data nodes. Each data state transition added for the current activity gets assigned τ as action. This requires adaptation from the stakeholders in post processing, because these transitions cover implicit data state transitions of the activity-centric process model. Then, the content of the OLC specific collection of data states for each data class is replaced with the data states of the input data nodes of the current activity. We finalize the input data node processing by adding a synchronization edge to the synchronized OLC between each two data state transitions that belong to different object life cycles and

that were added to a single object life cycle while analyzing the input data nodes of the current activity. Data state transitions connected via a synchronization edge are referred to as combined transition.

Next, the output data nodes of the activity are processed by determining their data states. These data states and the corresponding data nodes need to be filtered whether they are used within a data condition assigned to an outgoing control flow edge of a directly succeeding XOR split gateway or not. Data states being used in data conditions are not valid on all traces of an activity-centric process model. Therefore, the data states being valid for the current trace must be determined. A data state is valid for a given trace if it is part of the data condition that is assigned to the control flow edge having the XOR split gateway as source and the directly succeeding control flow node as target in the current trace. Furthermore, a data state is valid if it is not used in any such data condition. All valid data states are added to the corresponding single object life cycles. Then, the respecting data state transitions are added as well – one data state transition for each pair of data state of the OLC specific collection and valid data state if such transition is not yet existing. Each newly added data state transition gets assigned the activity label as action. Data state transitions being skipped for addition due to existence get their action extended by the label of the current activity. Again, the action labeling can be adapted by the stakeholder in post processing. Then, the content of the OLC specific collection of data states for each data class is replaced with the data states of the output data nodes of the current activity. Finalizing the output data node processing, synchronization edges are added to the synchronized OLC between each two data state transitions that were added or skipped and that belong to different single object life cycles.

Further synchronization edges are added if there exist multiple data nodes being output to the currently handled activity but referring to different data classes and if at least one input data node does not have a correspondent output data node of the same class. Given such input data node without a corresponding output data node, a synchronization edge having the state of this data node as source is added for each output data node referring to some other data class with that other data node's state as target. The type of all these synchronization edges is set to *currently*. Summarized, such synchronization edge specifies that the "output state" can only be reached if the "input state" is currently reached.

XOR split

If the control flow node is an XOR split gateway, the OLC specific collection of data states for the data class used in the data condition on the respecting outgoing control flow edge is adapted. The content of the collection is set to the value of the state of the data node used in the data condition for the current trace.

After processing all traces following steps ASD-5 and ASD-6, the final data states for each single OLC are set. A data state without an

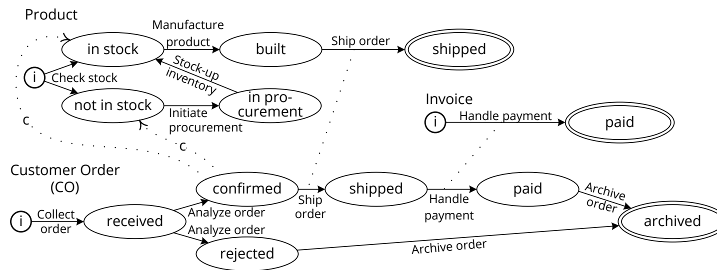


Figure 71: Synchronized object life cycle derived from the activity-centric process model given in Figure 69.

outgoing data state transition becomes a final state of the respecting OLC (ASD-7). Finally, in step ASD-8, the single OLCs are added to the synchronized object life cycle.

Activity *Check stock* from the activity-centric process model with attribute definition (visual representation in Figure 69) adds the data states *init*, *in stock*, and *not in stock* with transitions from *init* to the other two states to the *Product* object life cycle. Both data state transitions get assigned the action *checkStock*. Additionally, due to the input data node of type *CO* in state *confirmed*, two currently-typed synchronization edges targeting states *in stock* and *not in stock* respectively and both originating from state *confirmed* are added to the synchronized object life cycle. Figure 71 shows the synchronized OLC that is extracted from the ACP. In the given example, the activity-centric process model is fully specified because we show the roundtrip and thus, it was extracted from the synchronized object life cycle before. But the presented algorithm is also capable of handling underspecified activity-centric process models and models with subprocesses. Assume activity *Stock-up inventory* is missing in the activity-centric process model, the algorithm would add a transition from data state *in procurement* to data state *in stock* in the *Product* OLC automatically that is labeled with τ, since *in procurement* and *in stock* are directly succeeding data states in the ACP with the first one referring to an output and the second one referring to an input date node.

7.4 OBJECT LIFE CYCLE TO OBJECT-CENTRIC PROCESS MODEL

Given a synchronized object life cycle, an object-centric process model can be derived following five steps. The detailed algorithm of this transformation is presented in Algorithm 9 on page 325.

(SOD-1) Group data state transitions executed together into combined transitions,

(SOD-2) create data classes and add corresponding data states including the initial and final data states,

(SOD-3) extract attribute information from activity-centric process model,

(SOD-4) create business rules utilizing combined transitions and attribute information, and

(SOD-5) initialize object-centric process model with all business rules, the set of tasks utilized in the business rules, and a schema comprising all data classes.

SOD-3 states that this transformation requires the attribute information from the corresponding activity-centric process model pm'. As discussed in Section 7.2, this information could also be attached to the data state transitions of the synchronized OLC and the synchronization edges between single OLCs. However, for this chapter, we extract the information from the ACP. This information is only used if it is present. Otherwise, the corresponding parts of the algorithm are ignored resulting in an incomplete object-centric process model. The set of attributes is empty for all data classes and subsequently, no *defined* function exists in any business rule.

First, data state transitions of the synchronized object life cycle that are executed together are grouped into *combined transitions* (SOD-1). Two data state transitions are executed together if they are connected by a synchronization edge. Next, for each single OLC of the synchronized one, one data class is created. This data class gets assigned all data states including the initial and the final ones from the corresponding object life cycle (SOD-2). Additionally, for each single OLC, an empty map is initialized to store for each data state concatenated with a unique identifier (e. g., *created + 1234*) – representing the keys – a collection of fully qualified attribute types to be used in the *defined* functions of the OCP – representing the values for a key. Then, the attribute information gets extracted from the activity-centric process model pm' (SOD-3). Therefore, the XML structure gets parsed. For each data node, the attribute types marked as required are extracted and added via data class, data state, and identifier mapping to the corresponding collection of the respecting initialized map. Furthermore, all attribute types identified in the XML structure are added to the corresponding data class's attribute set. Already existing attribute types are skipped to avoid duplicates.

In step SOD-4, the business rules are created by processing the combined transitions and by utilizing the map with attribute types for the *defined* functions. For each combined transition, one business rule is created. The task affected by the business rule is derived from the action of the data state transitions (all transitions of a combined transition share the same action such that one action label can be taken randomly) and added to the business rule. Next, the *in-state* and *defined* functions are derived for both the pre- and postcondition of the business rule. An in-state function consists of the data class the respecting data state transition refers to and the source (precondition) respectively target (postcondition) data state of that transition. For each data state transition of the combined transition, one *in-state* function is derived. The *defined* functions are computed from above created maps. Given

the data class, data state, and identifier for a transition being part of the combined transition, for each attribute value stored for the given key, one *defined* function is added to the business rule as precondition if the given data state is the source of the corresponding data state transition or as postcondition if it is the target. This is repeated for all data state transitions of the current combined one. Finally, the *in-state* and *defined* functions of the pre- and postcondition need to be combined via logical *and*s and *or*s separately. *In-state* functions referring to the same data class are combined via a logical *or* $\vee$. Groups of *in-state* functions referring to different data classes are combined via a logical *and* $\wedge$. All *defined* functions are combined via logical *and* operators $\wedge$. The group of all *in-state* functions and the group of all *defined* functions are connected via a logical *and* as well. Then, the combination of functions is added to the pre- respectively postcondition of the business rule. After creating all business rules, the actual object-centric process model is initialized with these business rules, the set of tasks utilized in the business rules, and a schema comprising all data classes (SOD-5).

Table 5 shows an extract of the object-centric process model in business rule representation that is derived from the synchronized object life cycle represented in Figure 71. Showing a model transformation roundtrip, the OCP is equivalent to the one shown in Table 4 with some minor deviations since the roundtrip is not structurally isomorphic but behaviorally isomorphic. During OCP derivation, for the combined transition comprising both single transitions with the action label *Han-*

Table 5: Extract of object-centric process model in business rule representation visualizing the differences to Table 4. These are the labels of the tasks and business rule br8 specifically since both tasks from Table 4 are now represented in a single task *Handle payment* that acts on objects of both data classes.

Data classes:	CO, Product, Invoice
Set of tasks:	Collect order, Analyze order, Check stock, initiate procurement, Stock-up inventory, Manufacture product, Ship order, Handle payment, Archive order
Business rules:	b1, b2, b3, b4, b5, b6, b7, b8, b9
b1: Computer retailer receives order from customer	
...	
b8: Computer retailer invoices the order	
Precondition:	$\mathtt{instate}(\text{CO}, \text{shipped}) \wedge \mathtt{instate}(\text{Invoice}, \text{init})$
Tasks:	Handle payment(CO, Invoice)
Postcondition:	$\mathtt{instate}(\text{CO}, \text{paid}) \wedge \mathtt{instate}(\text{Invoice}, \text{paid}) \wedge \mathtt{defined}(\text{CO}, \text{InvoicingDate}) \wedge \mathtt{defined}(\text{CO}, \text{PaymentDate}) \wedge \mathtt{defined}(\text{Invoice}, \text{Amount})$
b9: Computer retailer archives order in information system	
...	

dle payment, a new business rule, *b8*, is created with both source states being added to the precondition and both target states being added to the postcondition for the affected data classes *customer order CO* and *Product*. Additionally, regarding attributes, $\mathtt{defined}(CO, InvoicingDate)$, $\mathtt{defined}(CO, PaymentDate)$, and $\mathtt{defined}(Product, Amount)$ are extracted from the corresponding data nodes of the activity-centric process model with attribute definition in XML representation. The task name is changed compared to Table 4 because of stakeholder interference and because the algorithm discussed in this subsection creates a single task per business rule and comprises all actions within. The task now is labeled *Handle payment* as the corresponding transition instead of *receivePayment* and *setPaid* in Table 4. Alternatively, the algorithm could be changed to create a single task per data modification. This would result in a change of business rule *b3* to two single tasks except one manipulating two objects of different data classes. In fact, this change would prevent that a task is created that manipulates objects of multiple classes. The remaining business rules are the same as shown in Table 4 except for the task labels because of stakeholder interference.

7.5 PROCESS MODEL REFINEMENT

Enacting process models requires detailed specifications of the overall allowed process behavior including exception handling and rarely taken options. In large software systems, e. g., in the SAP Business Objects Suite, this behavior is usually specified by object life cycles that show all manipulations allowed to be performed on objects of some data class. Coverage of all situations – including exceptional and rarely used behavior – leads to large OLCs, which easily include more than one hundred data states per data class. Process models, which shall be enacted, need to cover all data state transitions described in these OLCs while also weak conformance must hold, i. e., process model and object life cycles must be aligned in both directions.

A notion to check the first property is object life cycle coverage [176] that checks whether a process model explicitly models all data state transitions for a given OLC of one data class. Thus, this check must be repeated for each class' OLC being of interest. Thereby, the check abstracts from synchronization edges probably specified between single OLCs such that this must be checked separately. However, satisfied weak conformance checks ensure correct synchronization and allow safe skipping of this additional check. Generally, all data classes utilized within a process model are of interest but in case the process expert needs to focus on specific classes, the set can be reduced. As weak conformance and OLC coverage are defined for a single process model only, we assume a single, consolidated process model as shown, for instance, in Figure 69 on page 167 or in Figure 7 on page 30 for the subsequent paragraphs. There, we describe the corresponding re-

finement approach before we afterwards discuss the application of the presented refinement approach to cases where multiple process models represent a business process as given in the build-to-order and delivery process in Section 2.4.

Single process model

Given one process model, if the coverage check returns a positive result for each data class, refinement of the process model is not necessary as all data state transitions are already explicitly covered by the process model. Otherwise, two types of refinements may be applied to the process model. If there is a gap within the process model data specifications (e. g., in the process model in Figure 7 on page 30, activity *Check stock* writes a data node of class *Product* in state *not in stock* while activity *Stock-up inventory* writes a data node of this class in state *in stock* with no further activity in a path betweeen these two accessing some node of this class "bypassing" data state *in procurement* which is written by some other activity that is not part of the process), new control flow nodes are added to the process model to close the gap. If there are "jumps" in the process model data specifications (e. g., in the process model in Figure 7, activity *Handle payment* transitions a data node of class *CO* from *shipped* directly to state *paid* including the transition to state *invoiced* as intermediate result), corresponding activities are replaced with connected control flow nodes covering the data state transitions. Alternatively, the activity might be typed as subprocess and the corresponding control flow nodes are inserted into the subprocess. In case, the activity is already a subprocess containing control flow nodes, the OLC is tailored with respect to the identified "jump" (see Section 7.6) and the coverage check is repeated. Then, the subprocess gets refined towards object life cycle coverage regarding the initial version of the tailored part of the OLC.

The process model refinement comprises four main steps executed in sequence:

1. Preparation of the given process model for refinement,
2. identify gaps and "jumps" in the process model's data specification,
3. handle the gaps in data specifications, and
4. handle the "jumps" in data specifications.

1—Process model preparation. For process model preparation, two tasks must be performed. First, the given process model is checked for weak conformance and gets adapted until the notion is satisfied for all object life cycles of all data classes utilized within the process model and that are of interest for the stakeholder. Second, to reduce the computation complexity, all activities not reading or writing at least one data node referring to a data class of interest and all data nodes not referring to a data class of interest get marked as superfluous, because they do not change the result of the OLC coverage check. Next, the actual refinement take place (steps 2 to 4). Thereby, we assume that each step in an OLC, i. e., each data state transition, is performed through

exactly one activity.

2—Gap and "jump" identification. Summarized, we step through the process model node by node and check whether each relevant control flow node's input and output data nodes correspond to the ones expected based on the OLCs. In the beginning, we mark the control flow edge originating from the source node in the process model and we mark all initial data states in the OLCs. Iteratively, from the set of marked control flow edges in the process model, one is chosen randomly. If the node targeted by this edge is a join or merge gateway, all control flow edges targeting this gateway are checked for being marked. If all these edges are marked, all control flow edges originating from this gateway get marked while all control flow edges targeting the node get unmarked. Otherwise, another control flow edge is randomly chosen from the set of the marked ones. If the node targeted by the chosen edge is a split or a fork gateway, all control flow edges originating from this gateway get marked while all control flow edges targeting the gateway get unmarked. If the node targeted by the chosen edge is an activity, the edge gets unmarked, the data check is performed for this node, and the control flow edge originating from the activity gets marked. If the node targeted by the chosen edge is an end event, the edge gets unmarked and no new edge gets marked such that the set of marked control flow edges is empty and thus, the iteration is finished. The data check validates that (i) each read data node is in a data state being marked in the corresponding OLC, (ii) the states of data nodes read and written are source and target of a data state transition respectively, and (iii) each write of a data class requires a read of the same class except the state of the written data node is a direct successor of the initial data state in the corresponding OLC. If the data check validates to true for an activity, i. e., no violation exists, states of written data nodes get marked after states of read data nodes of the same data class got unmarked.

Gap in data specifications

3—Gap handling. If the state of a read data node is not marked (cf. violation of (i)) or (iii) is violated, then there exists a gap in the process model data specifications and therefore, control flow nodes covering all possible paths need to be added. To do so, in all affected OLCs, all execution sequences from each currently marked data state to the data state of the input data node are determined. These build subnets of the OLCs; the subnets then get connected via synchronization edges as given in the synchronized object life cycle. This (synchronized) OLC is transformed into a process model roughly following the algorithm discussed in Section 7.2 – with slight adaptations after transformation (see PMD-2 below). From the resulting process model, the source node, the sink node, and the control flow edges originating respectively targeting these two nodes are removed. Finally, the control flow edge targeting the activity causing the gap in the data specifications is rerouted to the first control flow node of the derived process model while the last con-

trol flow node of this process model is connected with the gap causing activity via a new control flow edge.

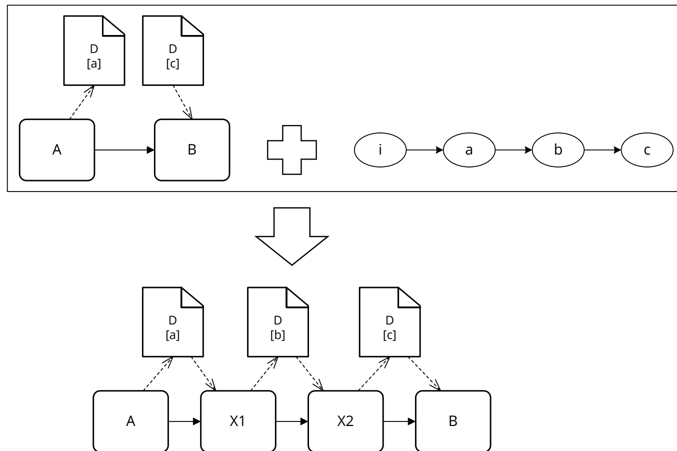


Figure 72: The gap in the process model in the upper part is fixed by refining the process model based on object life cycle information (right upper part) towards the process model in the lower part.

Figure 72 shows how the refinement works for an abstract example. Activities *A* and *B* both process a data node of class *D* with the input of the succeeding one not matching the output of the preceding one. Applying the notion of weak conformance (see Chapter 6), correctness can be determined such that there must exist some set of activities ensuring correct data manipulation. Assuming that this shall be covered by the given process, we can refine it following above procedures resulting in the process model given in the lower part of Figure 72. Activities *X1* and *X2* are added to transition the data node (object) from state *a* via state *b* to state *c* as required by the original process model. In case, activity *B* in the model in the upper part of Figure 72 would write a data node of class *D* in state *c* instead of reading it, only one activity would be added and a data node with state *b* would become input to activity *B*.

Summarized, the process model derivation (PMD) follows seven rules:

(PMD-1) A data state transition maps to an activity having a data node of the corresponding class in the source state as input and a data node in the target state of the data state transition as output,

(PMD-2) two transitions being connected by an untyped synchronization edge are mapped into the same activity which input and output data nodes are added correspondingly,

(PMD-3) two transitions originating from the same state or targeting the same state are handled by the same activity,

(PMD-4) an activity having multiple outputs of the same data class is succeeded by a split gateway,

(PMD-5) an activity having multiple inputs of the same data class is preceded by a join gateway,

(PMD-6) activities manipulating distinct data classes without synchronization dependencies for the affected data states are put into AND-blocks, and

(PMD-7) the control flow nodes are connected with respect to the object life cycle specifications while additionally needed gateways as, for instance, joins and merges, are added.

Assuming two transitions are connected via an untyped synchronization edge, PMD-2 is also enforced, if one transition was already present in the process model and the other transition gets added to the process model through above steps. In such case, for the second transition, no new activity is created but the already existing one is reused. Further added activities might need to be shifted to different places within the process model to align with the order of data states in the OLCs.

Jump in data specifications

4—"Jump" handling. If a single activity comprises multiple data state transitions of one data class, e. g., an activity changes the state of class *customer order CO* from *confirmed* via *accepted*, *shipped*, and *invoiced* to *paid*, a "jump" in the process model data specifications exists (cf. violation of (ii)). Analogously to the gap handling, a subnet of the synchronized object life cycle is determined and the corresponding process model is derived (cf. Section 7.2). The activity causing the "jump" is replaced with this set of connected control flow nodes. Alternatively, the activity can be retyped into a subprocess. Thereby, two cases must be distinguished. If the activity is of type *task* or *multi-instance task* respectively, the source and sink node of the derived process model are not removed and the complete process is inserted into the scope of the subprocess. If the activity is already of type *subprocess* or *multi-instance subprocess* respectively and contains some control flow nodes (otherwise the reaction of the first case take places), a copy of the synchronized OLC is tailored with respect to the control flow nodes within the subprocess (cf. Section 7.6 with preserving the intermediate data states). The alignment is performed for this tailored OLC and the process model contained in the subprocess and finally resolves the "jump". Though, applying the OLC coverage check would still identify an issue because of multiple state transitions within the subprocess. However, we ensured that the hierarchical structure aligns with the OLC and thus, the issue can now be ignored.

The upper part of Figure 73 shows an activity *A* that takes a data node of class *D* as input and manipulates it such that the processing result will be state *d*. Thereby, the OLC in the upper right of Figure 73 shows that activity *A* must also include manipulations that put the corresponding object in states *b* and *c* during activity execution. Subsequently, *A* can be replaced by a set of three activities *X1*, *X2*, and *X3* performing the mentioned data state transitions (see lower part of Figure 73).

Refinement results discussion

The refinement approach discussed in this section does not provide an optimal solution but provides one solution such that process model and synchronized object life cycle are smoothly aligned to each other.

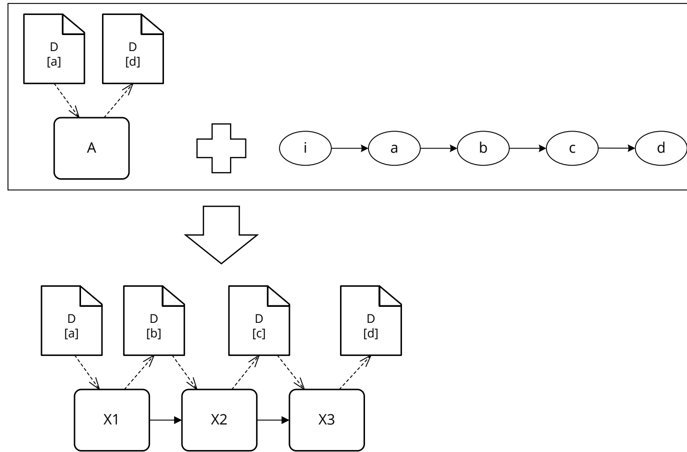


Figure 73: The "jump" in the process model in the upper part is fixed by refining the process model based on object life cycle information (right upper part) towards the process model in the lower part.

This includes that there may exist multiple solutions to fix the gaps and "jumps" in the process model data specifications. Thus, we implement above refinements not as automatic approach directly changing the process model but propose the single changes to the stakeholder or process expert which she has to approve. We envision this to happen by marking refinement changes either after each iteration step, a predefined number, of steps or after all steps during the gap and "jump" handling respectively. All made changes to the process model are highlighted, for instance, by color-coding and the process expert decides for each change.

Then, the algorithm starts over with the current process model as basis until the coverage checks succeeds and no refinement proposals are made. The algorithm may also be catered to the process expert's needs by skipping parts of the algorithm. For instance, if aggregated activities or subprocesses are tolerated, the corresponding part about "jumps" in the process model data specifications can be omitted.

Multiple process models

The main challenges for business processes being represented by multiple process models is that gaps in one process model may be represented by another one or that two process models present the same part of the business process but on different abstraction levels or that a process model represents the subprocess being contained in another process model, i. e., the "jump" in one process model is detailed by another one. In this chapter, we assume that two process models do not comprise the same part of the business process except for subprocess relationships. Identifying relations between process models, that are distinguished through business process model abstraction techniques [96, 131, 165, 209, 255, 309], are out of scope. Business process architectures (BPAs) [79, 86, 287] can be used for relation visualization while BPA creation techniques [85, 88] can help with their identification. Above presented refinement approach builds the basis for the multi

process approach. In contrast to the introduced algorithm, changes cannot be undertaken just because of the requirement induced by one model, it must be checked whether the found inconsistency is an actual one or whether it is resolved by another process model of the same business process. Therefore, identification of a violation (i), (ii), or (iii) for some activity in some process model triggers a check of the remaining process models whether there exists a subnet solving the issue, i. e., whether the output of some subnet fits the input of the activity causing the violation. Correctness of the subnet is validated once the corresponding process model is checked. Subsequently, correctness (OLC coverage) can only be judged after analysis of all involved process models. Considering the overall-example of this thesis introduced in Section 2.4, for activity *Process order* with data nodes of class *customer order CO* in states *confirmed* and *paid* as input and output respectively (see Figure 5 on page 29), violation (ii) – a "jump" – is corrected by existence of the process model presented in Figure 7. Thus, no further correction is required.

7.6 OBJECT LIFE CYCLE TAILORING

Object life cycles founding the basis for process execution easily comprise more than one hundred data states including all exceptional and rarely occurring ones. Usually, a business process covers a certain case and thus, it only utilizes a subset of these data states. For instance, the running example introduced in Section 2.4 consists of five process models from the computer retailer's point of view describing the built-to-order and delivery process while further business processes of the retailer as accounting and customer support are not described although they do utilize further data states. Each of the five process models describes a different part of the overall business process on different levels of abstraction. Each process model utilizes a different subset of data states and thus, a different part of the synchronized object life cycle – in fact, each part is represented by a subgraph of the synchronized object life cycle for the build-to-order and delivery business process[1]. This shows, that different views on the synchronized OLC are required.

If the stakeholder's focus is set to one process model, the readability and understandability of OLCs can be increased by omitting not covered data states and transitions. Thereby, two options arise. Implicit data state transitions as induced by activities causing a "jump" can either be preserved in or removed from the OLC depending on the use case. Consider the process refinement approach discussed in the previous subsection. There, it is required to know the OLC of a process model representing a subprocess with preservation of implicit

1 Please note that the synchronized object life cycle utilized in this thesis is in turn a subgraph of a synchronized object life cycle comprising all business processes executed by the given software system.

state transitions to reason about OLC coverage. In contrast, in practice, to get the big picture about the manipulations done within a process model, the implicit data state transitions are of rather low importance and can be omitted.

Omitting data states and state transitions allows a reduction of the OLC to the smallest set of data states and state transitions, for which the process model still satisfies weak conformance with respect to the appropriate data classes. An OLC comprising only data state transitions utilized in the process model while omitting implicit state transitions can be achieved using the transformation algorithm presented in Section 7.3. Analogously to the refinement algorithms above, this only provides one solution with the option that the derived OLC differs from the synchronized object life cycle. To avoid such behavior, we introduce algorithms, which adapt the given synchronized object life cycle preserving its basic structure.

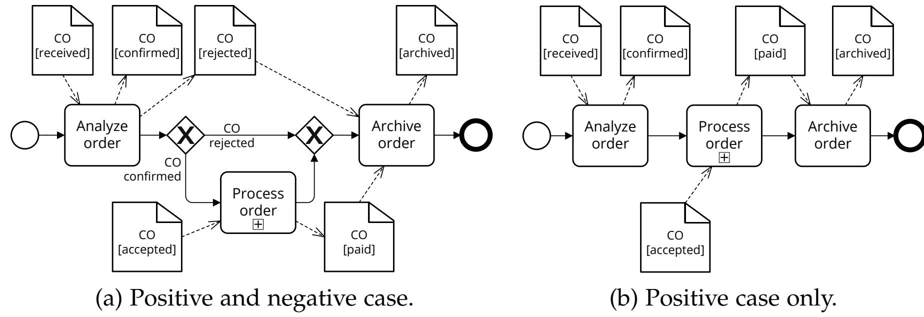
Single process model

First, we discuss the tailoring of a synchronized OLC with respect to a single process model. Discussions for multiple involved process models follow below. The algorithm comprises four main steps executed in sequence:

1. Determine all explicit data state transitions in the process model,
2. determine all implicit data state transitions resulting from gaps in the process model,
3. tailor the single object life cycles of the synchronized OLC accordingly, and
4. handle the synchronization edges which source or target may need to be adapted through removal of data states respectively data state transitions.

Process scenarios, the algorithm is applied to, are required to satisfy the notion of weak conformance. If they do not, correction mechanisms are applied. Details are given in Chapter 6.

1—Explicit data state transitions. For coping with the first step, each activity of the given process model is taken and for each distinct data class read and written, the Cartesian product of states corresponding to input data nodes and states corresponding to output data nodes is created. Considering the process model in Figure 74a, for data class *customer order CO*, the data state pairs (*received*,*confirmed*) and (*received*,*rejected*) are determined for activity *Analyze order* while the data state pair (*accepted*,*paid*) is determined for activity *Process order*. The process model in Figure 74b shows the positive case of the model in Figure 74a. Thus, the data state pair (*received*,*rejected*) for activity *Analyze order* is not determined. Each such data state pair will represent a data state transition (omit implicit data state transitions resulting from "jumps" in OLCs) or will summarize an execution sequence (preserve implicit data state transitions resulting from "jumps" in OLCs) in the corresponding object life cycle.



(a) Positive and negative case. (b) Positive case only.

Figure 74: Both process models show condensed representations of the subprocess given in Figure 5 as part of the build-to-order and delivery process utilized in this thesis. While (a) shows both possible outcomes for activity *Analyze order*, (b) only shows the positive case that confirms the *customer order CO*.

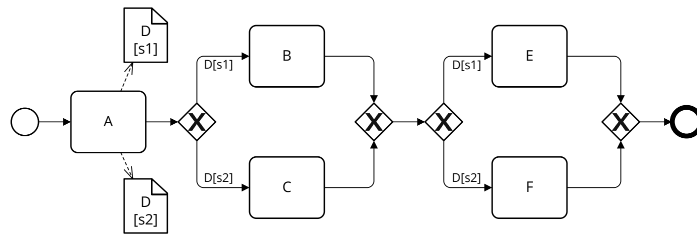


Figure 75: Abstract example where data dependencies restrict behavioral control flow relations. Activities *C* and *E* are in weak order relation $C \succ E$ but the data conditions annotated to the control flow edges originating from the XOR gateway splits reveal exclusiveness behavior $C + E$.

2—Implicit data state transitions resulting from gaps. Second, implicit data state transitions resulting from gaps in the process model are determined. Therefore, we first utilize the transitive closure over the control flow relation $\mathfrak{C}^+$ to compute the behavioral relations between each pair of activities of the given process model. All pairs of activities in weak order relation are captured for further processing. From the set of these pairs, we keep all pairs of activities, where there exists at least one data class that is read or written (in any combination) by both activities. The others get removed. Furthermore, weak order relations are transitive, e.g., $a_1 \succ a_2$ and $a_2 \succ a_3$ imply that $a_1 \succ a_3$. To determine the implicit data state transitions resulting from gaps in the process model, we are only interested in pairs of activities accessing the same data class with the smallest distance, i.e., a pair of activities, e.g., (a_1, a_3), specifying a path that is subsumed by the combination of several paths represented by pairs of activities, e.g., (a_1, a_2) and (a_2, a_3), can be removed if a_1, a_2, and a_3 handle data nodes of the same data class.

Due to data dependencies on control flow edges, such pair of activities (a_i, a_j) may be in weak order relation based on $\mathfrak{C}^+$ but during process execution, the path $a_i \mathfrak{C}^+ a_j$ can never occur. Figure 75 shows an

abstract example where $C \succ E$ (weak order relation such that there exists a path through the process model in which activity C occurs before activity E) based on control flow but data prohibits each of these execution sequences such that $C + E$ (both activities are exclusive to each other); execution of activity *C* requires data state *s2* for a data node of class *D* while activity *E* requires data state *s1* for a data node of the same class without data state changes of nodes of class *D* after termination of activity *A*. Therefore, we eliminate such pairs (a_i, a_j) from the captured list. For each remaining pair of activities (a_i, a_j) in weak order relation such that $a_i \succ a_j$, we check for the Cartesian product of data states of appropriate data nodes, whether there exists an execution sequence from the first to the second data state in the corresponding OLCs. Given a data class c and thus object life cycle l, appropriateness is specified as follows. If activity a_i writes nodes of the given data class, these are appropriate. Otherwise, the nodes of that class read by activity a_i are considered appropriate. If activity a_j reads nodes of the given data class, these are appropriate. Otherwise, the nodes of that class written by activity a_j are considered appropriate. Data states of activity a_i are considered first data states (s_1) and states of activity a_j are considered second data states (s_2). If there does not exist an execution sequence $s_1 \Rightarrow_{l_c} s_2$ for any data class c read or written by both activities, the pair is removed. Otherwise, it is kept for further processing and the corresponding existing data state transitions are stored for tailoring.

Considering the process models given in Figure 74, activities *Analyze order* and *Process order* are in weak order relation; both handle data nodes of class *CO*. In Figure 74a, reachability of activity *Process order* from activity *Analyze order* at run-time is ensured for state *confirmed* of data nodes of class *CO* (cf. data condition on corresponding control flow edge). The output data nodes of activity *Analyze order* and the input data nodes of activity *Process order* are considered appropriate with respect to above criteria. Thus, the Cartesian product leads to sets *confirmed,accepted* and *rejected,accepted* of data states. In the corresponding object life cycle, there does not exist a path $\mathrm{rejected} \Rightarrow_{l_{CO}} \mathrm{accepted}$ but there does exist a path $\mathrm{confirmed} \Rightarrow_{l_{CO}} \mathrm{accepted}$ (cf. Figure 76 on page 187 or Figure 28 on page 68). Subsequently, the pair of activities (*Analyze order*,*Process order*) is kept for further processing and data states are stored for tailoring.

While a last access to a data class is definitely comprised within the data state transitions, a first access in the process model is not. Therefore, data state transitions from the initial data state to the one first accessed in the process model must be determined. This is done by stepping through each execution sequence (path) of the process model. Given a data class c, all activities are checked first whether they read data nodes of class c and second whether they write data nodes of class c. If data class c is recognized in an activity (first match), the states of all data nodes of class c read or written respectively by the correspond-

ing activity are taken and further processing of this path is stopped. For each such data state, a data state transition from the initial state of the corresponding OLC to the determined one is stored for tailoring. The complete procedure is repeated for each data class utilized in the process model to determine the implicit state transitions resulting from gaps in the process model for all single OLCs.

3—Tailoring. Utilizing the data state transitions determined in the previous steps and based on the fact whether implicit state transitions resulting from "jumps" shall be preserved or omitted in the OLCs, the single object life cycles get tailored. First, we discuss the case, the implicit data state transitions shall be omitted to minimize the sizes of the OLCs. For a given data class c, all determined data state transitions are checked whether they map to an execution sequence σ_{l_c} in the corresponding OLC l such that $|\sigma_{l_c}| = 1$. If it is the case, that data state transition is marked in the object life cycle as *required*. Afterwards, the remaining (unmarked) data state transitions with $|\sigma_{k,l_c}| > 1$ are processed.

First, they are checked for overlappings, i.e., whether there exist data state transitions that are subsumed by multiple others as, for instance, a data state transition in data class *CO* from state *received* to state *shipped* subsumes a state transition from state *accepted* to *shipped* (cf. OLC in upper part of Figure 76). For each state transition $s_i \xrightarrow{\sigma_p} s_j$ identified to subsume another transition $s_m \xrightarrow{\sigma_q} s_n$, it is replaced by data state transitions $s_i \xrightarrow{\sigma_u} s_m$ and $s_n \xrightarrow{\sigma_v} s_j$. If $i = m$ or $j = n$, the corresponding data state transition is omitted. In the given example, the transition from *received* to *shipped* is replaced by transition *received* to accepted while the second one is omitted due to state equality (*shipped*) such that transitions from *received* to *accepted* and from *accepted* to *shipped* are the intermediate result for the overlapping handling. A created data state transition is marked in the OLC, if $|\sigma_u| = 1$ or $|\sigma_v| = 1$. Otherwise, it will get processed as the other unmarked execution sequences of size greater 1. Then, the subsumption checks are continued until all subsumptions are resolved.

Each remaining data state transition of length greater 1 is checked whether it subsumes a marked transition in the object life cycle. If not, the data state transition is added to the OLC while all subsumed state transitions are removed. The added state transition is marked. If the transition subsumes marked transitions, it is replaced as described above until it does not subsume any marked transition. Then, the transition is added to the OLC. During subsumption resolution, it may happen that the created transitions all are of length 1 such that no more transition must be added to the OLC. Finally, data states not being connected to the OLC graph anymore get removed from the object life cycle.

In case preservation of implicit data state transitions resulting from "jumps" in the OLCs is required, above described procedure is altered

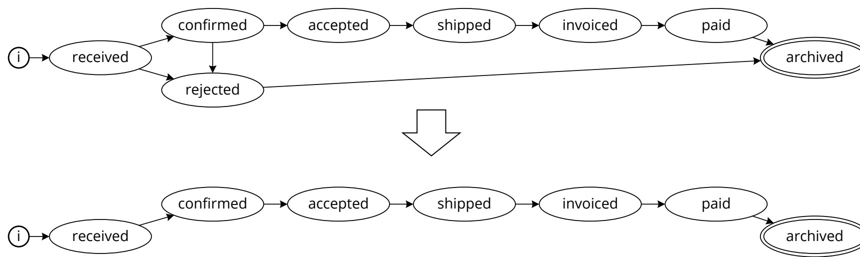


Figure 76: Object life cycle tailoring with respect to the process model given in Figure 74b from the upper OLC to the lower one under the assumption that data state transitions resulting from "jumps" shall be kept as execution sequence.

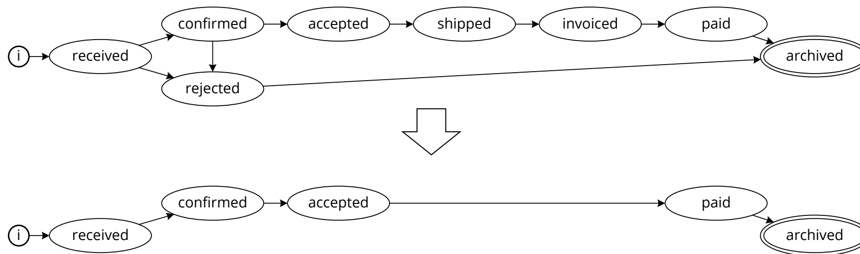


Figure 77: Object life cycle tailoring with respect to the process model given in Figure 74b from the upper OLC to the lower one under the assumption that data state transitions resulting from "jumps" shall be reduced to a single transition.

for data state transitions that map to an execution sequence in the OLC such that $|\sigma_{k,l_c}| > 1$. Instead of adding these to the OLC and deleting all subsumed state transitions after subsumption resolution, the complete execution sequence is preserved, i. e., all contained transitions are marked required. This leads to an object life cycle where all unused paths are omitted. In the build-to-order and delivery example in this thesis (cf. Section 2.4) as well as in the process model given in Figure 74a, the state transition confirmed $\xrightarrow{\sigma_2}$ rejected is removed from the OLC of data class *customer order CO*. In the process model given in Figure 74b, the state transitions received $\xrightarrow{\sigma_1}$ rejected, confirmed $\xrightarrow{\sigma_2}$ rejected, and rejected $\xrightarrow{\sigma_3}$ archived are removed from the OLC of data class *customer order CO* as presented in Figure 76. Data state *rejected* is not connected anymore to the object life cycle graph and thus, it gets removed. These examples hold true if preservation of implicit data state transitions resulting from "jumps" in the OLCs is required. Ignoring this property, for instance, for the process model given in Figure 74b, the data state transitions from *accepted* to *shipped* to *invoiced* and to *paid* are replaced by a direct transition from *accepted* to *paid* as presented in Figure 77.

After processing all determined data state transitions, we verify correctness of the tailoring by applying the notion of projection inheritance [337] to ensure that the tailored object life cycle allows the same

behavior as the initial OLC, if unused data states are hidden. If projection inheritance is not satisfied, the tailored OLC would introduce new behavior, i. e., new state transitions not comprised in the initial OLC.

4—Synchronization Edges. Adaptation of the synchronization edges follows a set of seven rules. Thereby, the result cannot represent all given constraints as specified. Instead, the result will represent an approximation being usually more restrictive than the original constraints. For instance, in the synchronized object life cycle given in Figure 28 on page 68, the OLC of class *Invoice* contains two state transitions from *i* (for initial) to *created* and from *created* to *sent*, where the second state transition is to be executed together with the transition from *shipped* to *invoiced* of the corresponding OLC of class *customer order CO*. Assuming, after tailoring, there exists only one state transition which subsumes both mentioned ones such that $i \xrightarrow{\sigma} sent$. Then, the question arises how to include the given dependency. The solution presented next connects the given state transition in the OLC of class *CO* with the new one in the tailored OLC of class *Invoice* ensuring that the sending of the invoice is synchronized with the billing of the customer order given the tradeoff that invoice creation cannot be performed before order shipping anymore although it was allowed before. This example shows how rules restrict process execution since we assume ensuring compliance and organizational guidelines more critical than freedom of execution order for single process participants. The synchronization edge adaptation rules (SER) are:

(SER-1) If source and target of a synchronization edge exist in the tailored object life cycles, the synchronization edge is added as given.

(SER-2) If a state transition specified as source or target is missing in the tailored OLCs, the transition subsuming the missing one replaces it as source respectively target of a synchronization edge.

(SER-3) If a missing state transition is not subsumed by some newly added one, the synchronization edge is not added to the tailored OLC.

(SER-4) If a data state specified as target is missing in the tailored OLCs and if it is not subsumed by some newly added data state transition, the synchronization edge is not added to the tailored OLC.

(SER-5) If a data state specified as source is missing in the tailored OLCs and if it is not subsumed by some newly added data state transition, the source is replaced by each final state of the tailored OLC from which the missing source state is reachable in the original OLC; if multiple such final states fulfill the condition, multiple synchronization edges are added with the respective source data states.

(SER-6) If the source state is missing but subsumed by some data state transition, the target of that state transition gets the source of the synchronization edge.

(SER-7) If the target state is missing but subsumed by some data state transition, the target of that state transition gets the target of the synchronization edge.

While the application of rules SER-1 and SER-2 are clear respectively discussed in the example above, we now discuss the remaining rules. If a data state transition is not subsumed by some other (SER-3), it is neither modeled explicitly nor implicitly and therefore, cannot influence the tailored synchronized OLC. If a target data state is missing and not subsumed by some state transition (SER-4), the corresponding synchronization edge is not required since the state, affected by this condition, will never be reached. In contrast, if a source data state is missing and not subsumed (SER-5), the target should still be affected by this condition. Such missing and not subsumed data state may only be on an execution sequence in the original OLC starting from some state that is a final one in the tailored OLC. Therefore, each final data state in the tailored object life cycle from which the missing state is reachable in the original OLC becomes the source of a synchronization edge. SER-5 is the only rule that relaxes the original dependency constraints, because after adaptation, the target may be reached although the originally expected state is not yet valid. However, taking the final state is the closest approximation.

SER-6 and SER-7 replace the missing data state with an appropriate succeeding one that is the data state the transition subsuming the missing state targets. In case of a missing source (SER-6) for a currently typed synchronization edge, the state transition t leading to the target s' in the OLC of the target shall only take place, if the source is an active data state. Taking the state preceding the source means that t may be executed before the source is reached in turn meaning that some required task may not have been done. Thus, taking the succeeding state ensures that all required tasks have been done. Second, this succeeding state is the one reached after executing the required tasks although some more tasks may have been performed; i. e., the succeeding state is the first state capable of guaranteeing the source state is respectively was reached. This is analogous for previously typed synchronization edges since they exactly give the mentioned guarantee that a specific state was reached at some point in time. Similarly, the missing targets (SER-7) are resolved by succeeding data states, because the state transition leading to the target in the respective OLC is subsumed by the one leading to the successor such that the latter one may only be executed if the original one might have been executed.

Multiple process models

Tailoring a synchronized object life cycle with respect to multiple process models utilizes the same steps as described for single process models with some extensions. First, steps 1 to 3 are performed for each process model resulting in a single tailored OLC per data class and source process model. Then, for each data class, the tailored OLCs are combined using state chart integration techniques [108, 235, 258, 328]. State chart integration is done via data state name matching followed by a data state analysis as described in step 2 for single process models – determination of implicit data state transitions resulting from gaps. This

harmonizes the data state transitions and avoids that state transitions exist in the integrated OLC which subsume the same state transition from the original synchronized OLC.

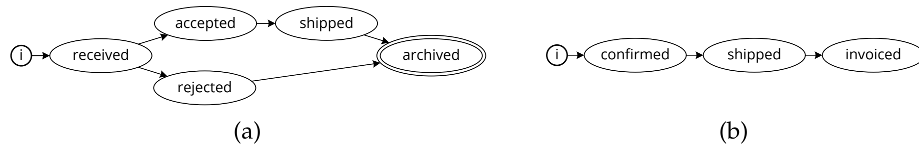


Figure 78: Initially tailored object life cycles.

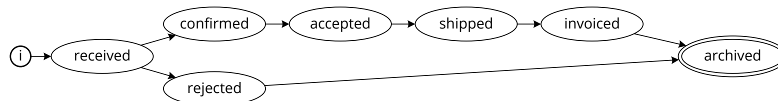


Figure 79: Integrated tailored object life cycle.

For instance, let the object life cycles given in Figures 78a and 78b be two tailored OLCs from the synchronized OLC presented in Figure 28 on page 68 for data class *customer order CO*. Then, the transition from state *accepted* to state *shipped* in Figure 78a is subsumed by the transition from state *confirmed* to state *shipped* in Figure 78b which in turn is interwoven with the transition from state *received* to state *accepted* in Figure 78a, because both transitions subsume one state of each other (cf. data states *confirmed* and *accepted* respectively). Applying the transition splitting as described above leads to an integrated tailored OLC as shown in Figure 79.

If preservation of implicit data state transitions in the OLC resulting from "jumps" is required, then state chart integration by name matching is sufficient since no two data states or transitions are subsumed by each other after integration. Finally, step 4, the synchronization edge handling takes place resulting in one tailored synchronized object life cycle, which satisfies the notion of weak conformance for each of the process models and is minimal in size, if preservation of implicit state transitions resulting from "jumps" is not required.

7.7 RELATED WORK

Activity-centric business process modeling emerged from workflow modeling and is described extensively in several works, e. g., [370], with BPMN being the widely used industry standard [243]. The object-centric modeling paradigm was initiated by IBM research [237] and further formalized by several researchers, e. g., [54, 384]. Additionally, deviations of this paradigm have been developed and process engines executing such process models have been established [173, 231]. Both paradigms compete each other although they only provide different views on the same business processes putting either activities or data

in focus. Instead of keeping both paradigms separated, we combined them with a set of transformation algorithms.

First steps towards an integration have been taken by extracting unsynchronized [97, 98, 292] and synchronized [186] object life cycles from activity-centric process models as well as by extracting activities statically from processing paths through data objects [363] or goals [356] or dynamically from data dependencies on missing data [347]. Closely related to our concept of utilizing object life cycles as mediator between both paradigms, [189] introduces a Petri net based approach representing the exchanged "artifacts", i. e., data nodes (objects), in the context of inter-organizational communication; however, the concepts can also be applied to intra-organizational communication. Both, the object-centric and the activity-centric representations utilize Petri nets with a single net per data class on the object-centric side and an integrated net showing the actions applied to data nodes (objects) of all classes.

The mentioned approaches provide means to partly support one of the seven transformations introduced in this chapter. Liu et al. [186] provide an approach closely related to our ACP-to-OLC-transformation (see Section 7.3) but they omit attribute consideration. Additionally, they also assume that each data node (object) written in a specific state is also read in this state, if the node (object) gets read again. Besides considering data attributes, we also allow the read of a previously written node (object), if there exists a path between the respecting states in the object life cycle (cf. Chapter 6 where we discussed the issue of underspecification in process models).

Model transformations are a general challenge in computer science often discussed in the area of model-driven engineering (MDE) [158], since there *everything is a model* [206]. Thus, MDE aims to develop, maintain and evolve software by performing model transformations [207]. [57, 206, 207] provide overviews on types of model transformations, e. g., source code to structural model and vice versa but also transformations preserving the modeling type like migration, simplification, or optimization. Targeting the challenge of keeping two models in sync, the research field on bidirectional model transformations provides "mechanism for maintaining the consistency of two (or more) related sources of information" [58]. This stream of research can be combined with our algorithms allowing real-time adaptations to multiple views and directly seeing the impact of one change in all required views. This gets especially important in the context of collaborative modeling of multiple stakeholders.

7.8 conclusion

We distinguish between inter-view transformations and intra-view transformations. For both, we provided algorithms allowing the stakeholder to see the business process in multiple views each providing different

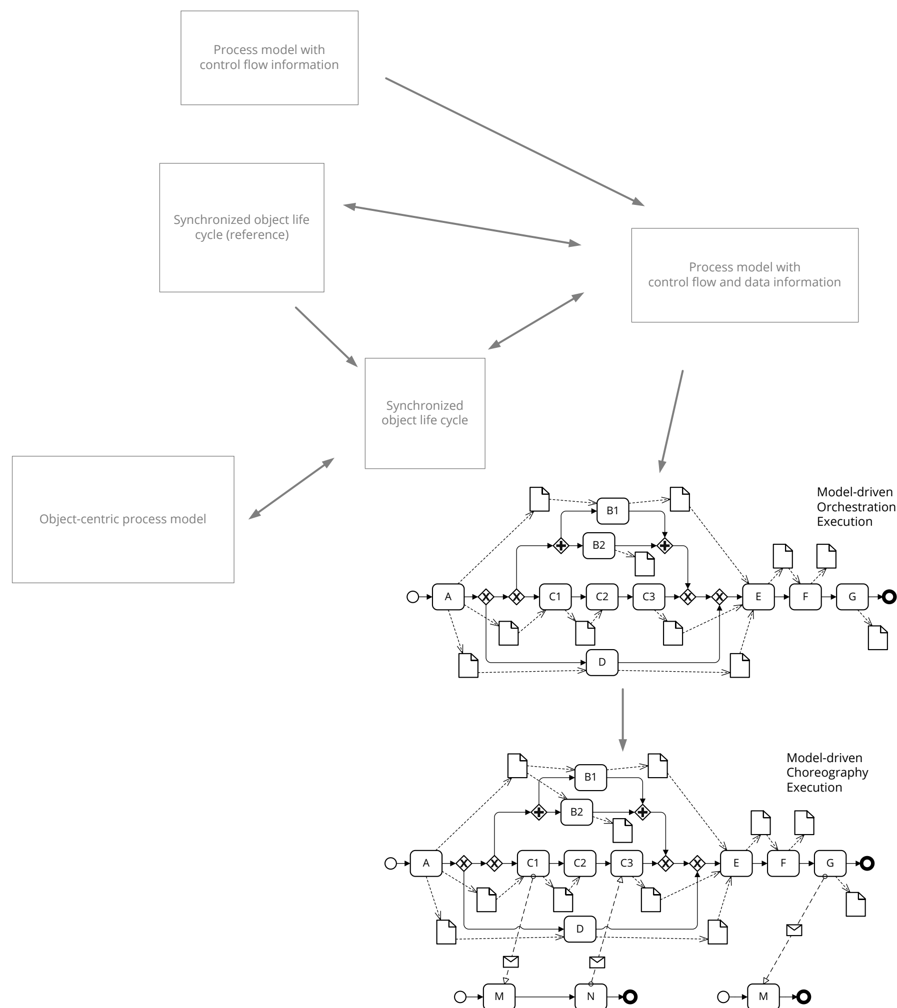
insights. In the context of inter-view transformations, we introduced algorithms allowing roundtrip transformations between activity-centric process models and object-centric process models with object life cycles as mediator between both modeling paradigms. An ACP or an OCP is transformed into an OLC which in turn can be transformed in an ACP or an OCP. In the context of intra-view transformations, we introduced two algorithms to tailor an object life cycle with respect to a given process model or to refine a process model with respect to a given object life cycle towards an executable process model. Both algorithms utilize the notion of weak conformance to ensure correctness of data specifications by removing or adding details to the corresponding model. We showed applicability of all algorithms by applying them to an extract of the build-to-order and delivery scenario of this thesis.

Part III

AUTOMATED PROCESS MODEL EXECUTION

8 MODEL-DRIVEN BUSINESS PROCESS EXECUTION

This chapter is based on results published in [218, 219, 220, 221, 223, 224].



EXECUTION OF BUSINESS PROCESSES in process-aware information systems or process engines comprises four dimensions: (i) control flow to manage the order of tasks and ensure enablement of a task only after termination of the preceding one, (ii) resource coordination, (iii) execution of services and applications (automatic tasks) as well as steering manual execution, and (iv) data that is queried, transformed, and provided to process stakeholders. Process engines such as Activiti [2], Bonita [32], AristaFlow [177], or the camunda BPM platform [45] are able to execute the control flow of a business process and to allocate required resources based on a given process model in an automated

fashion (i, ii). Service execution is generally supported, for instance, by web service calls that are configured for each task separately (iii). For execution, data also plays an important role because it specifies pre- and postconditions of tasks (see Chapter 1) that are connected by control flow, it shows dependencies between multiple objects, it shows actually processed objects, and it specifies information exchanged between multiple organizations (iv). In current process engines, data is not supported on the model level but only, if at all, on the configuration level.

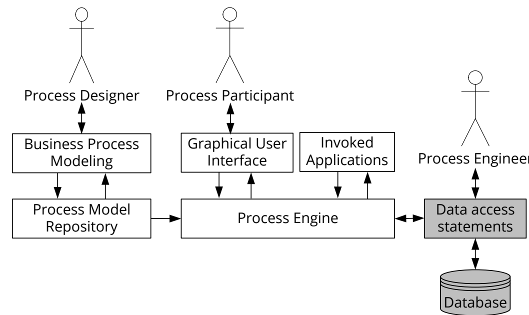


Figure 80: Classical architecture of a workflow management system from [370].

Figure 80 presents an abstract classical architecture of a workflow management system (WFMS). Generally, a process engine has access to a repository containing the process models describing business processes that may be executed. As soon as a start event of a particular process occurs, e. g., the stakeholder starts the process, the process engine creates a new instance of this process and executes the control flow as specified in the corresponding process model. Thereby, the process engine is able to allocate specified user tasks to process participants via a graphical user interface or to invoke an application for execution of service tasks. Send and receive tasks are also executed automatically – they send a previously created message or wait for retrieval of a message which then can be processed in upcoming tasks. A process engineer manually specifies data access statements (see shaded elements in Figure 80) incorporating the data dependencies. These statements are then utilized from the process engine during process execution. In the scope of this chapter, we assume that each organization has a single process engine handling all process executions.

During process execution, some processes require interaction with process participants from other organizations via message exchange, e. g., to conclude contracts. Considering the build-to-order and delivery process from Section 2.4, the computer manufacturer, for instance, needs to receive orders from customers and needs to prepare the ordering of corresponding components for manufacturing via request-response interactions with suppliers. The interaction between business processes of multiple organizations via message exchange is called *pro-*

cess choreography. Each organization has its own, private process engine such that interaction between business processes of different organizations is not controlled by one central agent and requires communication between the corresponding process engines, e. g., via web services. While modern process engines allow enforcement of the correct ordering of messages from a given process model, the process internal message handling, e. g., message creation, has to be implemented manually – similarly to data dependencies in process orchestrations.

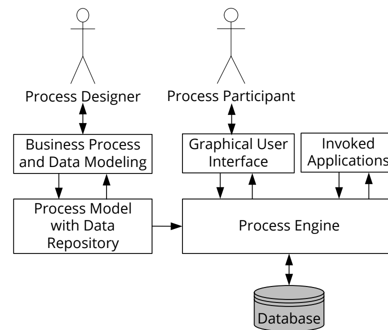


Figure 81: Improved architecture of a workflow management system.

In this chapter, we introduce concepts that allow execution of data aspects in business processes entirely model based. In detail, we ensure (i) task enablement only if all data objects required for the specific task instance exist in the required data states, (ii) correct data states for objects written by a task upon task termination, (iii) retrieval and storage of data objects including all their attributes in a relational database, and (iv) automatic message, i. e., data, exchange between business processes of different organizations. We describe the concepts generically based on the formalisms given in Chapter 4 and utilize the Business Process Model and Notation (BPMN) to describe these concepts by example. First, we introduce a concept to execute complex data dependencies in process orchestrations in Sections 8.1 and 8.2 (i, ii). Second, we introduce a concept to handle actual process data during process execution based on process model information only in Section 8.3 (iii). Third, we introduce a concept to automate the data exchange in process choreographies in Section 8.4 (iv). The combination of these concepts enables model-based data execution in activity-centric process models (ACPs). Considering BPMN, only few modeling extensions to the standard are required for application of these concepts. Feasibility is shown in our implementation discussed in Section 8.6. Figure 81 shows an abstract architecture for a WFMS based on the concepts that will be introduced below. The manual definition of data access statements during process configuration is not necessary anymore. Instead, data dependencies, data accesses, and message exchanges are modeled during the business process design.

8.1 COMPLEX DATA DEPENDENCIES IN ORCHESTRATIONS

Activity-driven process description languages such as BPMN [243] allow the modeling of simple data dependencies; for example, that an activity can only be executed if a particular data node (object) is in a particular state. Information about these simple data dependencies may be used for process execution from a process model. However, when *m:n relationships* arise between processes, activities, and data nodes (objects), modeling and execution becomes more difficult.

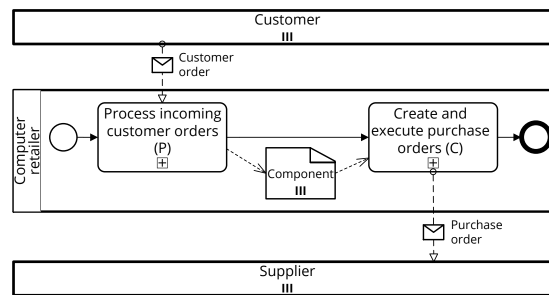


Figure 82: Consolidated part of the build-to-order and delivery process from Section 2.4 where subprocess *P* collects multiple *Orders* from several *Customers* in an internal loop and where subprocess *C* sends multiple *Purchase orders* to several *Suppliers* using a multi-instance subprocess internally.

The running example in Section 2.4, a build-to-order and delivery process, shows a typical scenario with m:n relationships which we refer to as *complex data dependencies*. Figure 82 presents a reduced view on that business process highlighting the complex dependencies. Here, the components required to build the ordered products are not in stock and will be purchased upon request. For an incoming *Customer order CO*, the computer retailer identifies all *Components* needed to build the product. Then, the retailer creates and executes a number of *Purchase orders PO* to be sent to various *Suppliers* to procure the required *Components*. To reduce costs, *Components* of multiple *Customer orders* are bundled in joint *Purchase orders*. The two subprocesses in Figure 82 handle complex m:n relationships between the different orders: one *Purchase order* contains *Components* of multiple *Customer orders* and one *Customer order* utilizes *Components* of multiple *Purchase orders*.

In fact, activity-driven process description languages do not provide sufficient modeling concepts for capturing m:n relationships between data nodes (objects), activities, and processes. As a consequence, in modern process engines, as the ones mentioned above, actual data dependencies – simple and complex ones – are often not derived from a process model. They are rather implemented manually in services and application code which yields high development efforts and is prone to errors. Explicitly adding data dependencies to process models provides multiple advantages. In contrast to having data only specified inside

services and applications called from the process, an integrated view facilitates *communication with stakeholders* about processes and their data manipulations; there are *no hidden dependencies*. With execution semantics, one can *automatically execute* processes with complex data dependencies from a model only. An integrated conceptual model allows for *analyzing control and data flow combined* regarding their consistency (see Chapter 6) and correctness. Finally, *different views on a process can be generated automatically* (see Chapter 7); for instance, models showing how a data object evolves throughout process execution [97, 186, 210, 292], i. e., deriving an object life cycle (OLC) from a process model as discussed in Section 7.3.

Existing techniques for integrating data and control flow follow the object-centric process modeling paradigm [54, 173, 175, 231, 232, 245, 344]: a process is modeled by its involved objects; each one has a life cycle and multiple objects synchronize on their state changes (see Definition 3.7). This paradigm is beneficial when process flow follows from process objects, e. g., in manufacturing processes [231]. However, there are many domains where processes are rather activity-centric such as accounting, insurance handling, or municipal procedures. In these, execution follows an explicitly prescribed ordering of domain activities, not necessarily tied to a particular object life cycle.

Orchestration Execution Requirements

Requirements

For such processes, changing from an activity-centric view to an object-centric view for the sake of data support has disadvantages. Besides having to redesign all existing processes in a new paradigm and to train process modelers, one also has to change the IT infrastructure in terms of switching to new process engines. Additionally, one may no longer be supported by existing and broadly used standards although the Case Management Model and Notation (CMMN) standard was recently released; business acceptance cannot be evaluated yet. This gives – as already indicated above – rise to the first orchestration execution requirement **(OER-1—Activity)**: Processes can be modeled in an activity-centric way using well-established industrial standards for describing process dynamics and data dependencies.

We address the problem of modeling and executing *activity-centric* processes with complex data dependencies. The problem itself was researched for more than a decade revealing numerous requirements as summarized in [173]. The following requirements of [173] have to be met to execute activity-centric processes with complex data dependencies directly from a process model:

(OER-2—Data Integration) The process model refers to data in terms of object types, defines pre- and postconditions for activities (cf. requirements R01 and R14 in [173]), and

(OER-3—Object Behavior) expresses how data objects change (cf. R04 in [173])

(OER-4—Object Interaction) in relation and interaction with other data objects; objects are in 1:1, 1:n, or m:n relationships. Thereby, process execution depends on the state of its interrelated data objects (cf. R05 in [173]), and

(OER-5—Variable Granularity) an activity changes a single object, multiple related objects of different types, or multiple objects of the same type (cf. R17 in [173]).

In the remainder of this section, we introduce a technique that addresses the requirements (OER-1)-(OER-5). This technique combines classical activity-centric modeling, e. g., in BPMN [243], with relational data modeling as known from relational databases [305].

Solution

Proposed solution: Extending ACPs

To this end, we introduce few extensions to activity-centric process modeling data objects referred to as data nodes in this thesis: Each data node gets dedicated life cycle information, an object identifier, and fields to express any type of correlation, even m:n relationships, to other objects with identifiers. The utilization of the data semantics introduced in Section 4.7 ensures compatibility to many process description languages; both the ones having the concept of input and output set as BPMN and the ones which do not have this concept; data nodes of the same class being input or output to an activity build disjunctions and such sets of data nodes of different classes build conjunctions. Utilizing these concepts, we show how to automatically derive Structured Query Language (SQL) queries [147] from annotated data nodes that check and implement the conditions on data stored in a relational database. This effectively gives an activity-centric process model data-aware execution semantics in a technology that is well-understood and standardized.

Extending BPMN

Our concepts can be applied to all process description languages that adhere to the generic definition given throughout this section as subset to the one introduced in Definition 4.10. BPMN is such process description language. To introduce the new concepts into BPMN, we build on BPMN's extension mechanism called *extension points* to ensure conformance to the specification. Data annotations define pre- and postconditions of activities with respect to data, i. e., required input for activity enablement and achieved results upon activity termination. BPMN allows the definition of input and output sets representing different input requirements and termination results respectively, which, however, are not visually represented in the model. Since we rely on a graphical representation of data as well as input and output data sets, we restrict data modeling as introduced in Chapter 4. In fact, the corresponding semantics is a subset of the actual BPMN semantics and can thus easily be applied. In future work, the concepts introduced in this chapter can also be extended to support more complex input and output sets of data for activity enablement. To do so, the process model representa-

tion must allow graphical modeling of input and output sets[1], where different *input sets* respectively *output sets* represent alternative pre- respectively postconditions of activities.

State of the Art Data Modeling in BPMN

BPMN capabilities

Next, as representative for activity-centric process models, we discuss BPMN's existing capabilities for data modeling and its shortcomings with respect to the requirements introduced above. BPMN provides the concept of *data objects* – we refer to as data nodes – to describe different types of data in a business process. Data flow edges describe which activities read or write which data nodes. The same data node may be *represented* multiple times in the process model distinguishing distinct read or write accesses. A data flow edge from a data node to an activity describes a read access to a data object at run-time, which has to be present in order to execute the activity. A data flow edge from an activity to a data node describes a write access, which creates a data object at run-time, if it did not exist, or updates the data object, if it existed before. Figure 83 shows two data nodes of data class *D*, one is read by activity *A* and one is written. Data nodes can be modeled *single-instance* or *multi-instance*. A multi-instance data node is indicated by three parallel bars at the middle bottom and comprises a set of data nodes of one data class. Further, at run-time, a data object can be either *persistent* (stored in a database) or *non-persistent* (exists only while the process instance is active). We focus on persistent single- and multi-instance data nodes (objects).

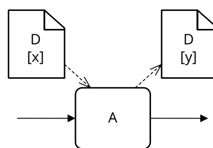


Figure 83: Part of implicit "object life cycle" of data class *D* with two data nodes in data states *x* and *y* respectively.

BPMN supports the notion of an *object life cycle* that gives data nodes (objects) a behavior implicitly. To express data behavior, BPMN provides the concept of *data states* which allows to annotate each data node with a *[state]*. Figure 83 shows an example. Activity *A* may only be executed when the respective data object is indeed in data state *x*; after activity termination, this object is in data state *y*. This refers to the data state transition $x \xrightarrow{A} y$ in the OLC of data class *D* (cf. Definition 3.8 and following paragraphs).

1 The current BPMN data flow representation can be extended by a grouping functionality such that activities have multiple sets of data nodes as input and output where each set belongs to one group and contains multiple data nodes. The existence of one such input set enables the activity from the data perspective and one such output set is required as execution result.

BPMN shortcomings

The BPMN semantics is not sufficient to express all data dependencies in a process model because of the following four aspects.

(BSC-1) The annotations to data nodes shown in Figure 83 do not allow distinction of different data objects of class *D* in the same process instance, e. g., two different customer orders.

(BSC-2) Likewise, we cannot express how several data nodes (objects) of different data classes relate to each other.

(BSC-3) Further, the type of a write access on data nodes (objects), e. g., create or update, is not clear from the annotations shown above.

(BSC-4) Finally, the correlation between a process instance and its data objects is not supported.

Concepts for Data Modeling in Activity-centric Process Modeling

The shortcomings identified for BPMN also hold true for further activity-centric process description languages. Next, we introduce few extensions to data nodes in common ACPs to address the shortcomings BSC-1 to BSC-4 derived from requirements OER-1 to OER-5. Afterwards, we illustrate the extensions on parts of the build-to-order and delivery process from Section 2.4.

New concepts

To distinguish and reference data objects (targeting BSC-1 and BSC-2), we utilize proven concepts from relational databases: *primary* and *foreign keys* [305]. We introduce object identifiers as an annotation to data nodes that describes the attribute by which different data objects can be distinguished; i. e., primary keys. Along the same lines, we introduce attributes which reference identifiers of other objects, i. e., foreign keys in [305]).

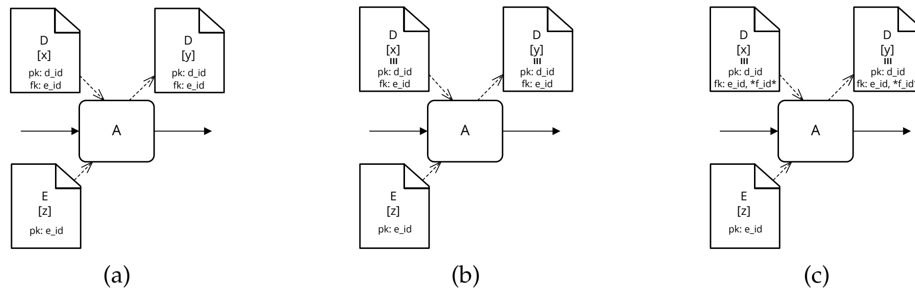


Figure 84: Describing data node interactions in (a) 1:1, (b) 1:n, and (c) m:n cardinality.

Figure 84 shows annotations for primary key (pk) and foreign key (fk) attributes in data nodes *D* and *E* that relate to data classes of respective names. At run-time, data objects of class *D* are distinguishable by primary key attribute *d_id* and objects of class *E* by attribute *e_id*. Based on Figure 84a, each data object d of class *D* is related to one object e of class *E* by the foreign key attribute *e_id*, i. e., objects of classes *D* and *E* are in a 1:1 relationship. Activity *A* can only execute when there exists one data object e of class *E* in data state *z* and when there exists one

object d of class *D* in state *x* that is also related to e. Upon execution, the object d enters data state *y* whereas e remains unchanged. A *multi-instance* representation of data node *D* expresses a 1:n relationship from objects of data class *E* to objects of class *D* as shown in Figure 84b, e. g., several computer components (*D*) for one customer order *E*. To execute activity *A*, *all* objects d_i of class D related to object e of class *E* have to be in data state *x*; the execution will put all objects d_i into state *y*.

We allow *multi-attribute foreign keys* to express m:n relationships between data nodes (objects) as follows. Assume, data nodes *D*, *E*, and *F* have primary keys *d_id*, *e_id*, *f_id*, respectively, and *D* has the foreign key attributes *e_id* and *f_id*. Each object of class *D* (e. g., a component) refers to one object of class *E* (e. g., a customer order the component originated from) and one object of class *F* (e. g., a purchase order in which the component is handled). Different objects of data class *D* may refer to the same object of class *E* (e. g., all components of the same customer order) but to different objects of class *F* (e. g., handled by different purchase orders) and vice versa. This yields an m:n relationship between objects of classes E and F via objects of class D. We allow *all-quantification* over foreign keys by enclosing them in asterisks, e. g., **f_id** in Figure 84c. Here, activity *A* updates *all data objects* of class *D* from state *x* to state *y* that are related to one object e of class *E* and that are related to any object of class *F*, i. e., we quantify over **f_id**. A foreign key attribute can be *null* indicating that the specific reference is not yet set. A data node (object) may have further attributes, however, these are not specified in the node itself but in a data model, possibly given as UML class diagram [244], accompanying the process model as indicated in Chapter 4. These are then mainly used for process data handling (see Section 8.3). Furthermore, the data model also contains primary and foreign key information such that the relationships modeled in the process model must adhere to the data model (see Section 8.5).

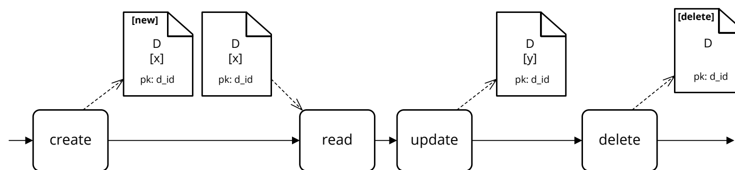


Figure 85: Describing create, read, update, and delete of a data node (object) through annotations in the upper left corner of a data node.

Tackling shortcoming BSC-3 regarding the derivation of data dependencies from a process model, we need to be able to express the four major data operations: *create*, *read*, *update*, and *delete* (CRUD operations) for a data node (object) (see Figure 85). Read and update are already provided through an ACP's data flow edges. To express create or delete operations, we add two annotations shown in the upper left corner of a data node: *[new]* and *[delete]*. At run-time, *[new]* expresses the creation of a new data object having a completely fresh identifier and *[delete]*

expresses its deletion. Note that one activity can apply several data operations to different data objects indicated by multiple read or written data nodes in the process model. For example, activity *A* in Figure 84a reads and updates one object of class *D* and reads an object of class *E*.

The introduced extensions require that a data node contains a *name* and a set of attributes from which one needs to describe a *data state*, one an *object identifier* (primary key), and multiple ones a *set of relations* to other data objects (foreign keys). Figure 86 summarizes these extensions for a data node. Based on these considerations, we formally define such extended data node as given in Definition 4.2 on page 62. The set J of data node attributes may be omitted here, since these are not considered for data dependency derivation. However, they will be required for process data handling (see Section 8.3). Recall, function $\mathtt{type}_{\mathtt{op}} : D \rightarrow \{\mathtt{new}, \mathtt{delete}, \perp\}$ specifies the data operation. $\perp$ as result of function $\mathtt{type}_{\mathtt{op}}$ refers to a blank data operation description for which the data access is derived from the data flow: an input data flow requires a read operation while an output data flow requires an update operation.

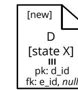


Figure 86: Extended data node.

To let a specific *process instance* create or update specific data objects, we need to link these two (BSC-4). For this, we adopt an idea from business artifacts [237] that each process instance is "driven" by a specific data object. We call this object *case object*; all other objects have to be related to it directly or indirectly by means of foreign keys. This idea naturally extends to instances of subprocesses or multi-instance activities. Each of them defines a *scope* which has a dedicated instance id. An annotation in a scope defines which class of data objects acts as case object. At run-time, a case object is either freshly created by its scope instance based on a *new* annotation (the object gets the id of its scope instance as primary key value). Alternatively, the case object already exists and is passed to the scope instance upon creation (the scope instance gets the id of its case object). By all means, a case object is always *single-instance*.

Data object persistence is achieved by storing them in a database. We utilize the concept of *data stores* to represent databases in the process model; data stores can be visualized multiple times where identical labels reference the same database. For each distinct data store, connection details to the corresponding database (address, username, password) must be specified. Additionally, a standard database should be defined. The persistence relation (cf. Definition 4.11 on page 71) allows the specification of database accesses. In case, a data node is not connected to a data store, the standard database is used as target for the access. In the remainder of this chapter, we assume all data objects to be stored in the same database. Thus, we omit representation of data stores in the process model as well as read and write data from/to this

single database, the standard database. Note that the concept of data stores is borrowed from BPMN with two major differences. First, we require the specification of database details (see above). Second, BPMN utilizes data stores as abstract replacement for data nodes such that data stores are connected to activities instead of data objects as in our case.

Given these additions to process modeling concepts, a process model $pm = (N, D, DS, Q, R, bp, \mathfrak{C}, \mathfrak{F}, type_a, type_t, type_g, \mu, \beta, DCF)$ with data-specific configuration $DCF_{pm} = (\mathfrak{P}, \xi, case, \varkappa)$ is sufficiently annotated to derive complex data dependencies for process execution. Next, we apply the introduced concepts to parts of the build-to-order and delivery process from Section 2.4 consisting of activities *Start processing cycle*, *Collect orders* detailed in Figure 6, *Request quotes* detailed in Figure 9, *Decide quotes*, and *Handle purchase order* detailed in Figure 10.

Modeling Example

We present these activities with two process models in BPMN enriched with the required concepts for model-driven execution of data dependencies. Figure 88 presents the initialization of the build-to-order and delivery process and the collection of *customer orders CO*. Figure 89 presents the purchase order preparation comprising activities *Request quotes* and *Correlate quote information to CPs and PO* with sending a single request per *purchase order PO* instead of multiple ones as introduced in Section 2.4. Further, the received quote is accepted. Thus, the process of deciding for a quote and canceling the remaining ones is omitted in this modeling example. For simplicity reasons, we assume that all data is persisted in the same database, e. g., all data nodes would be connected to the same data store. Thus, we omit representation of the data store in both process models. Each customer order can be fulfilled by a set of purchase orders and each purchase order consolidates the components required for several customer orders. This m:n relationship and all other relationships between data classes utilized in these two parts of the overall build-to-order and delivery process are expressed in the data model in Figure 87.

Data model. The *Processing Cycle (ProC)* contains information about *Customer Orders (CO)* being placed by customers and *Purchase Orders (PO)* used to organize the purchase of components within a particular time frame. Data class *Component (CP)* links *CO* and *PO* in an m:n-fashion, i. e., *CP* has one foreign key to *CO* and one to *PO*. *CO* and *PO* each have one foreign key to *ProC*. For each purchase order, one *Request* gets created that is also linked to the affected customer orders in an m:n-fashion by the components involved in the request. Therefore, a *Request* has a foreign key to *PO* and a component additionally has one foreign key to *Request*. Each request gets responded with a *Quote* that consists of *Quote Details (QD)* and possibly multiple *Quote Items (QI)*. A *Quote* has

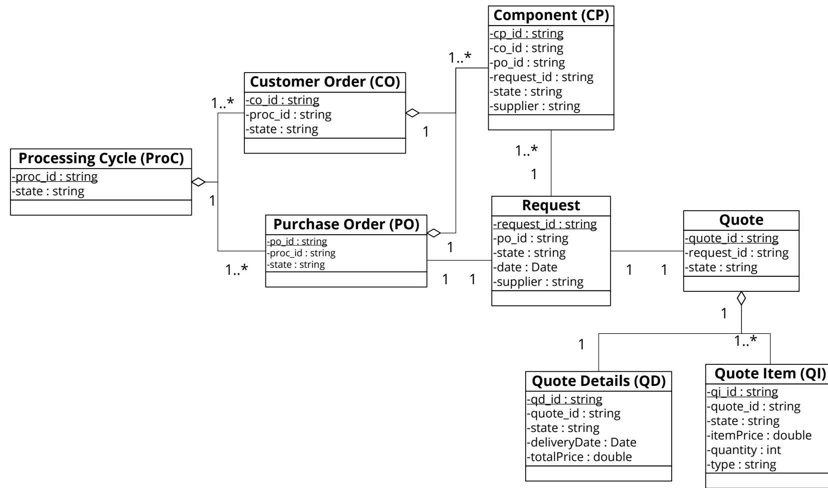


Figure 87: Data model containing required information about data classes utilized in the modeling example comprising Figures 88 and 89.

a foreign key to *Request* and *QD* as well as *QI* have a foreign key to *Quote*. Besides primary and foreign keys, each data class has a *state* attribute. Classes *CP* and *Quote* also have an attribute named *supplier* that is utilized during data dependencies execution. Furthermore, each data class has additional attributes that are omitted with respect to data dependencies. In Section 8.3, we discuss the process of data handling and, thus, we will extend this data model by these additional attributes.

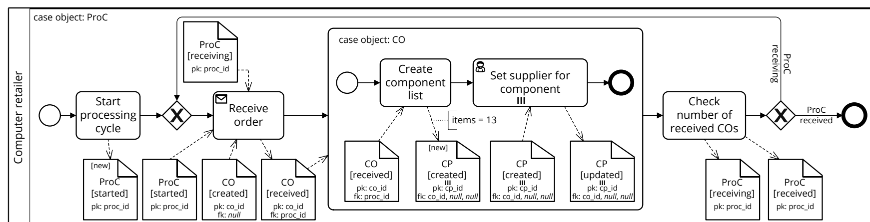


Figure 88: Customer order collection comprising activities *Start processing cycle* and *Collect orders* from the running example introduced in Section 2.4.

Customer order collection process. In Figure 88, the first task starts a new processing cycle allowing customers to send in orders for computers, for instance. By annotation *new*, a new *ProC* object is created for each task execution. As this is the case object of the process (see upper left corner of the process model), the primary key *proc_id* gets the id of the process instance as value. Next, customer orders *CO* are collected in a loop structure until a sufficient number of *CO*s have been successfully received and analyzed. Task *Receive customer order* receives one *CO* from a customer and correlates this *CO* object to the *ProC* object of this process instance (annotation *fk: proc_id*) before it is analyzed in a subprocess. *CO* is the case object of the subprocess which gets its instance id from the

primary key of the received *CO* object. Task *Create component list* determines the components *CP* needed to handle the customer order – several *CP* objects are created (annotation *new* on a multi-instance data node). The number of created objects is defined by the data flow edge annotation. Each *CP* object has a unique primary key value. The foreign key attribute *co_id* referring to the corresponding customer order is set to the primary key of the current *CO* object. The foreign key attributes referring to a corresponding purchase order and to a request are still *null*. The number of *CP* objects to create is given in the expression on the data output flow edge. Here, we give an explicit number (*items = 13*), but it could also be a regular expression to be evaluated or a process variable holding the result of the task execution, e. g., user input or the result of a service invocation. For instance, an ordered computer may consist of multiple configurations where also the number of components may vary as sometimes an external graphics card is used and sometimes an internal one resulting in one component to order less. Next, a user specifies for each component *CP* from which supplier the component will be purchased using, for instance, a form; i. e., for each component, the attribute *CP.supplier* gets updated. The actual data update is not in the scope of this section, but will be discussed and solved in Section 8.3 for possibly multiple attribute updates by one activity. In case only one attribute get updated, a solution presented in Section 8.2 can be used by labeling the activity correspondingly. For this example of supplier specification, the activity would be required to be labeled *Update supplier to $supplier*, where process variable $supplier specified the corresponding supplier. As last step, the current number of received customer orders is checked. If the computer retailer received a sufficient number of customer orders (state *received* for object *ProC*), the order retrieval is finished and thus, the process model reaches the end event. Otherwise (state *receiving* for object *ProC*), the computer retailer waits for additional customer orders prior order processing. Thereby, the specific number must be given in the task execution, e. g., hard coded for this task or a link where the number is provided dynamically.

Purchase order preparation process. The process model in Figure 89 describes how components that where extracted from different customer orders in Figure 88 are associated to purchase orders (*PO*s), building an m:n relationship between purchase orders and customer orders. The purchase order preparation can only start when there is a *PO* object in state *created* for that no process instance has been started yet.

Activity *Create purchase order* that precedes the process model shown in Figure 89 creates multiple *PO* objects correlated to the *ProC* object. These *PO* objects are handled separately in the shown process model: for each *PO* object, one instance of this process model is created having the *PO* object as case object and the corresponding *po_id* value as instance identifier. Based on the given purchase order *PO*, one request is created (annotation *new* for data node *Request*). For this *Request*, one

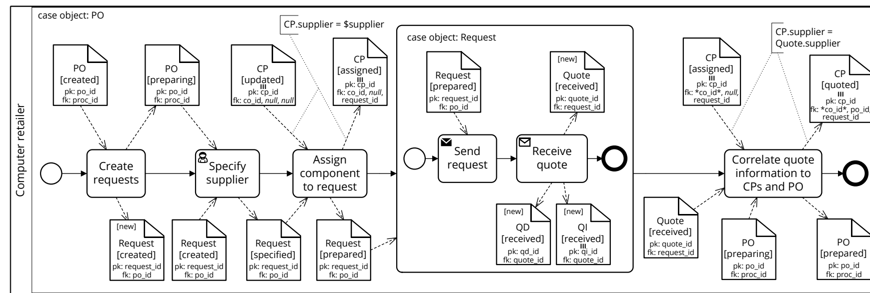


Figure 89: Purchase order preparation comprising activities *Request quotes* and *Correlate quote information to CPs and PO* from the running example (see Section 2.4) reduced to send a single request per purchase order *PO*. Thus, the activities regarding the decision taking with respect to quotes are omitted and the received quote is directly accepted.

supplier is selected to which the request is supposed to be sent; here we assume that the task *Specify supplier* sets a process variable *$supplier* local to the process instance that we can utilize for data object filtering. Task *Assign component to request* relates to the *Request all CP* objects in state *updated* that have no *request_id* value yet and where attribute *CP.supplier* equals the chosen *$supplier*. The relation is built by setting the value of *CP.request_id* to the primary key *Request.request_id*. The update quantifies over all values of *co_id* as indicated by the asterisks. The foreign key referencing the *PO* remains *null* in this step.

The execution of task *Assign component to request* results in a *CP* subset being related to one specific *Request*. The subsequent subprocess comprises a request-response scenario for the created *Request*. The *Request* along with the contained information about the *CP*s is sent to the corresponding supplier. Subsequently, a response containing the *Quote* including further *Quote Details QD* and multiple *Quote Items QI* is received from that supplier. *QD* and *QI*s refer to the *Quote* which in turn refers to the respecting *Request* by the shown foreign keys. Finally, information retrieved from the *Quote* are correlated to the purchase order *PO* prepared in this process model and the affected components *CP*. The affected components also get assigned the key *po_id* relating them to *PO*. Affectedness is determined through the data flow annotation *CP.supplier = Quote.supplier* indicating that only *CP*s are modified where the supplier – that was set in task *Set supplier for component* in Figure 88 – matches the supplier who sent the *Quote*. This task also concludes the purchase order preparation and sets the status of *PO* accordingly to *prepared*.

Object life cycle. Altogether, our extension to data nodes in activity-centric process models increases the expressiveness of an ACPs with information about process-data-correlation on instance level. As such, it does not interfere with the semantics discussed in Section 4.7. In addition, our extension is compatible with the object life cycle oriented techniques that allow to derive OLCs from sufficiently annotated process models (see [97, 186] and Section 7.3). Considering the two subparts of

the build-to-order and delivery process presented in Figures 88 and 89, we can derive the object life cycles shown in Figure 90 for data classes processing cycle *ProC*, component *CP*, and purchase order *PO* with a synchronization edge between the latter two for data state transition *Correlate quote information to CPs and PO*. We omit the presentation of the remaining utilized data classes for visualization purposes.

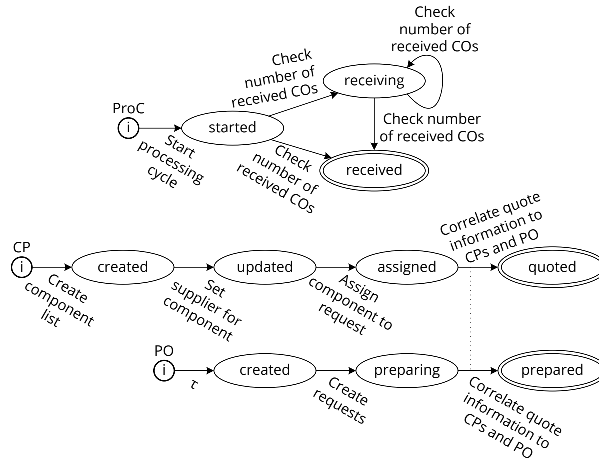


Figure 90: Object life cycles of data classes *ProC*, *CP*, and *PO* derived from the process models in Figures 88 and 89.

Extended Execution Semantics for Execution

After discussing the modeling of concepts to sufficiently annotate a process model for automatic execution of data dependencies, we introduce the corresponding *operational execution semantics*. Therefore, we *refine* the execution semantics described in Section 4.7 with SQL database queries that are derived from annotated input and output data nodes. Since the SQL query derivation only bases on single data nodes or set of data nodes but abstracts from data node coordination, for instance, in terms of input and output sets, the derivation concept covers both the standard BPMN semantics [243, Section 13] and the generic data semantics from Section 4.7. The difference gets important to identify for which data nodes, the concept is applied. While for the strict BPMN case, the derivation is applied to all data nodes within a single input respectively output set and the result is evaluated, the generic data semantics requires an application to all modeled input respectively output data nodes. Altogether, the refinement is standard compliance preservative.

For the extended semantics, we distinguish control flow and data flow aspects of a process model pm as discussed in Section 4.3. A state $m_{pm} = (m_{\mathfrak{C}}, m_{\mathfrak{F}})$ of pm consists of a control flow state $m_{\mathfrak{C}}$ describing a distribution of tokens on sequence flow edges and activities and a data flow state $m_{\mathfrak{F}}$ that is represented – differently to Chapter 4 – by a database db storing the data objects of pm in tables. The representation of data in a database is required to actually enable processing of the

data through SQL queries. The token semantics from Chapter 4 can directly be retrieved from the data nodes modeled in pm, the states of the corresponding data objects stored in db, and the correlation of data objects to the process instance. To distinguish the states of different process instances, each token in $m_{\mathfrak{C}}$ has an identifier id. The data model of the process is implemented in a set of relational databases DB (shared by all processes). Each data object is represented in some database $db \in DB$ as a table, where columns represent attributes. Each table has at least columns for primary key, foreign keys (if any), and data state. Each row in a table describes a single data object with concrete values, i. e., one instance. The concrete database db is determined via the data store a corresponding data node is associated with.

An activity a has several input and output data nodes. Data nodes referring to the same data class are considered as disjunction while sets of such data nodes referring to different classes are considered as conjunction separated for input and output to the activity. The database storing the persistent data objects has one table per data class which holds all corresponding data objects. In this thesis, we assume consistent labeling of table and data class such that identical labels refer to matching. An object o represented by a data node d being read from a is *available* in process instance i with identifier pid if the corresponding table in the respecting database db holds a particular row. We can define a *select* query $\Delta_d(pid)$ on db and a guard $\nu_d(pid)$ that compares the result of $\Delta_d(pid)$ to a constant or to another select query; $\nu_d(pid)$ is *true* if and only if o is available in i with identifier pid. An object o represented by a data node d being written by a has to become available when activity a completes. We operationalize this by executing an *insert*, *update*, or *delete* query $\Delta_d(pid)$ on db depending on d.

Activity a is *enabled* in process instance i with identifier pid in state $m_{pm} = (m_{\mathfrak{C}}), (m_{\mathfrak{F}})$ if and only if a token with identifier pid is on the input edge of a and for some input set $\{d_1, \ldots, d_n\}$ of a, each guard $\nu_{d_i}(pid)$ is *true*[2]. If A is enabled in m_{pm}, then A gets *started*, i. e., the token pid moves "inside" A in step $(m_{\mathfrak{C}}, m_{\mathfrak{F}}) \rightarrow (m'_{\mathfrak{C}}, m_{\mathfrak{F}})$ and depending on the type of activity corresponding services are called, forms are shown, etc. When this instance of a *completes*, the outgoing edge of a gets a token pid and the database gets updated in a step $(m'_{\mathfrak{C}}, m_{\mathfrak{F}}) \rightarrow (m''_{\mathfrak{C}}, m'_{\mathfrak{F}})$, where $m'_{\mathfrak{F}}$ is the result of executing queries $\Delta_{d_1}(pid), \ldots, \Delta_{d_m}(pid)$ for some output set $\{d_1, \ldots, d_m\}$ of a[3]. The semantics for gateways and events is extended correspondingly. If activity a is a subprocess with a case object of class c and a has a data node d of class c as data input, then we create a *new instance* of subprocess a for each entry returned by query $\Delta_d(pid)$. Each subprocess instance is

2 In the generic case, this is the single set of all specified input data nodes and the guards for one data node of each utilized data class must be true

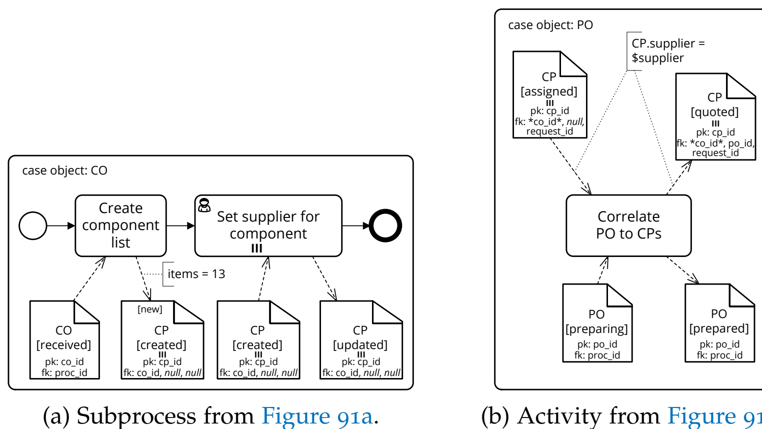
3 In the generic case, this is the single set of all specified output data nodes and for one data node of each utilized data class, one query is executed

identified by the primary key value of the corresponding row of object o being represented by d in the process model; $o \in \delta_D(d)$ and $d \in \delta_O(o)$.

Deriving Database Queries from Data Annotations

The annotated data nodes describe pre- and postconditions for the execution of activities. Next, we show how to derive from a sufficiently annotated data node d (and its context) a guard ν_d or a query Δ_d that realizes this pre- or postcondition. Both are then executed in the context of the database referred to by the corresponding data store or the standard data base if no data store is given.

In a combinatorial analysis, we considered the occurrence of a data object as *case object*, as single dependent object with 1:1 relationship to another object, and as multiple dependent object with 1:n or m:n relationship in the context of a *create*, *read*, *update*, and *delete* operation. Additionally, we considered process instantiation based on existing data and reading/updating object attributes other than state. The latter one only with respect to setting data dependency. A fully supported handling of attribute reading/updating will be discusses as process data handling in Section 8.3. Altogether, we obtained a complete collection of 46 *parametrized patterns* regarding the use of data objects as pre- or postconditions. For each of these patterns, we defined a corresponding database query or guard. During process execution, each input or output data node of an activity is matched against the patterns. The guard/query of the matching pattern is then used as described in the semantics discussion above.



(a) Subprocess from Figure 91a. (b) Activity from Figure 91b.

Figure 91: Fragments from process models in Figures 88 and 89 with incorporated adaptations due to direct dependency assumption. Modeled data nodes refer to the given scope instance case objects *CO* and *PO* while the process instance reference is given by the surrounding scope (omitted for readability reasons) with case object *ProC*.

Here, we present the seven patterns that are required to execute (i) the subprocess in the model in Figure 88 on page 206 and (ii) the task

succeeding the subprocess in the model in Figure 89 on page 208. These parts of the example process models have been chosen since they show (i) basic data operations and (ii) m:n relationship handling. Tables 6, 7, and 8 list the abstract patterns and their formalization that we explain next. The data model utilized in all seven patterns is presented in Table 8 on page 215. For presentation reasons, we assume that given a scope with a case object only data nodes are used in this scope which are either the case object or directly dependent to it, i.e., they relate to the case object via a explicit foreign key. Thus, for activity *Correlate quote information to CPs and PO* of the model in Figure 89, we need to remove the input data node *Quote* (since it is not directly related to *PO*) and assume that the supplier is provided through a process variable *$supplier*; the label is changed to *Correlate PO to CPs* to align activity labeling with the mentioned adaptations. The data node annotation is updated accordingly to *CP.supplier = $supplier*. Figure 91 presents both fragments with the required changes incorporated. Additionally, some of the presented patterns get slightly adapted (simplified) to satisfy the assumption of direct dependencies. All 46 patterns and their generic formalizations (including removal of above direct dependency assumption) are given in Section 8.2.

Each scope, e.g., subprocess, is driven by a particular case object. Each scope instance has a dedicated instance id. The symbol $*ID* refers to the instance id of the directly enclosing scope; $*PID* refers to the process instance id. Note, that the patterns assume modeling on the base process level; in our example this is the one in Figure 5 with *ProC* being the case object and thus referencing the process instance id. For the given fragments, additional scope are assumed to surround these fragments. For both, this additional scope has *ProC* as case object.

P1—Read single data node. Pattern P1 describes a read operation of an activity on a single data node *D1* that is also the case object of the scope. The activity is only enabled when this case object is in the given state *s*. The guard shown below P1 in Table 6 operationalizes this behavior: it is *true* if and only if table *D1* in the database has a row where the state attribute has the value *s* and the primary key *d1_id* is equal to the scope instance id *$ID*. This pattern is applied to activity *Create component list* in the subprocess in Figure 91a.

P2—Create multi-instance data node. Pattern P2 describes a write operation by an activity on a multi-instance data node *D2* that did not exist before (creation operation) and that depend on another data node *D1* read from the same activity in the same process instance. Thereby, data node *D1* acts as case object while data nodes *D2* are not. Successful termination of the activity creates multiple data nodes with the given state *t*, a unique identifier *d2_id*, and a reference *d3_id* to the read data node *D3*. The number is specified through a statement at the data flow edge. In the example, 13 components *CP* are created. The query shown below P2 in Table 6 operationalizes this behavior: it inserts multiple

Table 6: SQL queries for patterns P1 to P3 utilized in the subprocess and in the task *Correlate PO to CPs* in Figure 91.

P1	P2	P3
guard: (SELECT COUNT(D1.d1_id) FROM D1 WHERE D1.d1_id = $ID AND D1.state='s') ⩾ 1	INSERT INTO D2 (D2.d2_id, D2.d1_id, D2.state) VALUES (DEFAULT, fk, 't') ... (DEFAULT, fk, 't') //#items times fk = SELECT D1.d1_id FROM D1 WHERE D1.d1_id=$ID	UPDATE D1 SET D1.state = 's2' WHERE D1.d1_id = $ID

Table 7: Mapping a multi-instance task to a subprocess (pattern P4) and SQL query for pattern P5 both utilized for task *Set supplier for component* in Figure 91a.

P4	P5
	  For each D2.d2_id ∈ (SELECT D2.d2_id FROM D2 WHERE D2.d1_id = $ID) start subprocess with id D2.d2_id

rows into table *D2* in the database where the state attribute has the value *t*, the value for the primary key *d2_id* gets generated, and the foreign key has the value *d1_id* where *d1_id* is equal to the scope instance id *$ID*. This pattern is applied to activities *Create component list* and *Set supplier for component* (after application of P4) in the subprocess in Figure 91a. Further, this pattern is applied to activity *Correlate PO to CPs* in Figure 91b.

P3—Update single instance data node. Pattern P3 describes a write operation by an activity on a single data node *D1* that is also the case object of the scope and that did exist in a different state before activity execution (update operation). Successful termination of the activity updates the state attribute of data node *D1* from *s1* to the given state *s2*. The unique identifier remains unchanged. The query shown below P3 in Table 6 operationalizes this behavior: it updates a row in table *D1* in the database where the state attribute has the value *s1* and primary key *d1_id* is equal to the scope instance id *$ID*. The value of the state attribute is changed to *s2*. This pattern is applied to activity *Create component list* in the subprocess in Figure 91a.

P4—Mapping multi-instance tasks to subprocesses. Pattern 4 shows the mapping of a multi-instance tasks with a multi-instance data node as input to a corresponding subprocess providing the same behavior. The multi-instance activity is inserted as single-instance activity in a multi-instance subprocess. The multi-instance input data node is set as input to the multi-instance subprocess and a single-instance data node of the same data class with the same attributes (state, primary key, and foreign key) is added as input to the single-instance activity. The multi-instance output data node is also added as Pattern P4 is used to allow instantiation of activity *Set supplier for component* in the subprocess in Figure 91a as described next.

P5—Instantiate subprocesses from data. Pattern P5 deals with the instantiation of a multi-instance subprocess combined with a read operation on the dependent multi-instance data node *D2*. As described in the extended semantics paragraph, we create a new instance of the subprocess for each id returned by the query shown below P5 in Table 7. Each subprocess gets the primary key of a corresponding data node *D2* as scope instance id. In the given example, the subprocess containing activity *Set supplier for component* is instantiated 13 times since there exist that many *CP* objects in the database satisfying the query below P5 (see pattern P3). In each subprocess instance, the control flow reaches activity *Set supplier for component* for which pattern P1 applies. Since each subprocess instance gets a different scope instance id, the guard of pattern P1 can only get true for one object in the database such that successful subprocess execution can take place.

P6,P7—Transactional properties. Patterns P6 and P7 illustrate how our approach updates m:n relationships. Pattern P6 describes a read oper-

Table 8: SQL queries for patterns P6 and P7 utilized in task *Correlate PO to CPs* in Figure 91b.

Data model	P6	P7

P6:

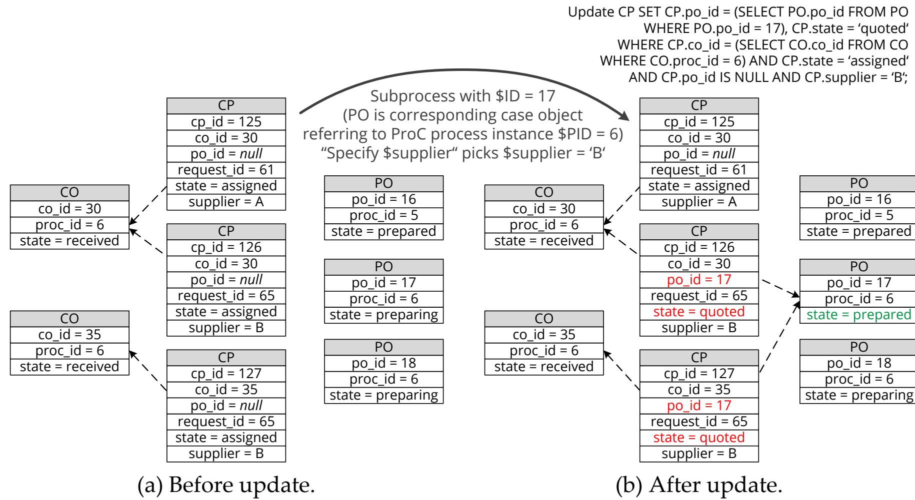
```
guard:
  (SELECT COUNT(D2.d2_id)
  FROM D2
  WHERE D2.state = 't'
  AND D2.d4_id IS NULL
  AND D2.attr2 = $var
  AND D2.d3_id = (
    SELECT D3.d3_id
    FROM D3
    WHERE D3.d1_id = (
      SELECT D4.d1_id
      FROM D4
      WHERE D4.d4_id = $ID
      )
    )
  ) >= 1
```

P7:

```
UPDATE D2
SET D2.d4_id = (
  SELECT D4.d4_id
  FROM D4
  WHERE D4.d4_id = $ID),
state = 'r'
WHERE D2.state = 't'
AND D2.d4_id IS NULL
AND D2.attr2 = $var
AND D2.d3_id = (
  SELECT D3.d3_id
  FROM D3
  WHERE D3.d1_id = (
    SELECT D4.d1_id
    FROM D4
    WHERE D4.d4_id = $ID
    )
  )
```

ation on multiple data objects *D2* that share a particular attribute value and are *not* related to the case object yet (in contrast to pattern P2). We have to ensure that another process instance does not interfere with reading (and later updating) these objects of class *D2*, that is, we have to provide *basic transactional properties*. We achieve this by accessing only those *D2* objects that are in some way related to the current process instance. Therefore, this read operation assumes a data model as shown in Table 8 (left): *D2* defines an m:n relationship between *D3* and *D4* via foreign keys *d3_id* and *d4_id* with *D4* being the case object of the process model. Assume, *D1* is the case object of the top process model and thus, *D1* contains the reference to the process execution through its primary key. *D3* and *D4* both have foreign keys to *D1* which must reference the same *D1* object to indicate that they are belong to the same process execution while *D1* is not part of the current instance; again, the general case is shown in Section 8.2. The guard shown below P6 in Table 8 is true if and only if there is at least one object of class *D2* in state *t*, with a particular attribute value, not linked to *D4*, and where the link to *D3* points to an object that itself is linked to the same *D1* object as *D4* is; i. e., foreign keys of *D3* and *D4* have the same value. The link to *D3* ensures that the process instance only reads *D2* objects and no other process instance can read them.

In our example, the pattern occurs at task *Correlate PO to CPs* reading all objects of class component *CP* that are not yet assigned to a purchase order *PO* (i. e., *null* value as foreign key) and where *CP.supplier* =



(a) Before update. (b) After update.

Figure 92: Setting missing foreign key relation of m:n object component *CP*: Concrete update query of subprocess with ID 17 to relate all *CP*s referring to supplier *B* to the purchase order *PO* with ID 17 indicated by arrows.

$supplier (see change of data flow edge annotation as discussed above). Figure 92a shows an extract of data objects with their current values after termination of the subprocess in the process model in Figure 89 and before enablement of task *Correlate PO to CPs* (referring to task *Correlate quote information to CPs and PO* in the process model in Figure 89; see also Figure 91b). For the processing cycle *ProC* with primary key 6, multiple purchase orders *PO* have been created. In the subprocess instance with *$ID* 17, the purchase order with the same value for the primary key is processed. Assume, the supplier *B* was chosen as partner for this purchase order in task *Specify supplier*. In this state, the queries of pattern P6 returns two rows (referring to components *CP*) having a *null* value for *po_id*, *B* as supplier value, and *assigned* as state value: the activity is enabled.

P7—Updating m:n relationships. Finally, pattern P7 describes an update operation on multiple data objects of class *D2* which sets the foreign key *d4_id* that is not set yet and moves them from state *t* to state *r*. All data objects of class *D2* get as value for *d4_id* the instance id of the current case object of class *D4*. Semantically, this turns the select statement of pattern P6 into an update statement that sets attributes *d4_id* and *state* for all rows where the pre-condition holds; see the SQL query of pattern P7 in Table 8.

In our example, pattern P7 occurs at task *Correlate PO to CPs* for assigning a specific set of components *CP* to a purchase order *PO* based on the chosen supplier. As assumed above for handling the *PO* with primary key 17, the process variable *$supplier* has the value *B*. The entire derived query is shown in Figure 92b (top right); executing the query gives components *CP* with primary key 126 and 127 respectively concrete

references to the *PO* (*po_id* = 17) and the state *quoted* (see red colored highlights for objects of class *CP* in Figure 92). The resulting state of the database in Figure 92b shows the m:n relationship that was set. Additionally, the corresponding *PO* object gets updated with respect to its state through some further derived but not shown query following pattern P3; the data state after activity termination is *prepared* (see green highlight).

Relation of patterns P1 to P7 to generic patterns. Pattern P1 refers to the generic pattern CR1. Pattern P2 refers to the generic pattern $D^{1:n}C1$. Pattern P3 refers to the generic pattern CU2. Pattern P4 refers to the generic pattern MI1. Pattern P5 refers to the generic pattern I4. Pattern P6 refers to the generic pattern $D^{m:n}R4$. Pattern P7 refers to the generic pattern $D^{m:n}U4$.

8.2 PATTERNS FOR SQL-QUERY DERIVATION

This section is dedicated to introduce all patterns required to derive data dependencies from a process model and execute them on a process engine. A relational database is used for persistence. For each pattern, the corresponding generic SQL pattern is presented; Table 9 provides an overview about the assignment of each pattern to the fol-

Table 9: Pattern classification overview.

Data operation	Case object (page 219)	Dependent 1:1 (page 221)	Dependent 1:n (page 226)	Dependent m:n (page 231)
select	CR1	$D^{1:1}R1$	$D^{1:n}R1$	$D^{m:n}R1$
	CR2	$D^{1:1}R2$	$D^{1:n}R2$	$D^{m:n}R2$
		$D^{1:1}R3$	$D^{1:n}R3$	$D^{m:n}R3$
				$D^{m:n}R4$
insert	CC1	$D^{1:1}C1$	$D^{1:n}C1$	$D^{m:n}C1$
	CC2	$D^{1:1}C2$	$D^{1:n}C2$	$D^{m:n}C2$
update	CU1	$D^{1:1}U1$	$D^{1:n}U1$	$D^{m:n}U1$
	CU2	$D^{1:1}U2$	$D^{1:n}U2$	$D^{m:n}U2$
		$D^{1:1}U3$	$D^{1:n}U3$	$D^{m:n}U3$
				$D^{m:n}U4$
delete	CD1	$D^{1:1}D1$	$D^{1:n}D1$	$D^{m:n}D1$
				$D^{m:n}D2$
instantiation (page 239)	I1, I2, I3, I4			
attribute (page 240)	A1, A2			
multi-instance task (page 242)	MI1, MI2, MI3			

lowing classification schema. The patterns are classified with respect to two dimensions: (i) type of data condition (horizontally) and (ii) data node function (vertically). Horizontally, they are classified into *pre-* and *postconditions* of an activity with respect to the data operation. While the fulfillment of pre-conditions decides about the enablement of an activity, postconditions must apply at termination of an activity. Pre-conditions are logical expressions that consist of one or more *select* statements. Postconditions are subdivided into *insert*, *update*, and *delete* statements.

Vertically, the patterns are classified regarding whether the operation is executed on the case object (that is bound to the process instance), on a single dependent data object (being in 1:1 relationship with another object), or multiple dependent data objects (being in 1:n or m:n relationship with another object). In the example in Section 8.1, *processing cycle ProC* is the case object, *customer order CO* directly depends on *ProC* in 1:1-fashion, *Request* data objects indirectly depend on *ProC* in 1:n-fashion via *purchase order PO* objects, and *Component* objects indirectly depend on *ProC* in m:n-fashion via *PO* and *CO* objects (cf. data model in Figure 87).

Furthermore, we need patterns to distinguish different cases of instantiation, i. e., which object is used as a case object and how identifiers of case object and process instance are set. Data nodes may also contain more attributes than those shown in a data node in the previous section, i. e., attributes other than primary key, foreign keys, and data state (cf. J in Definition 4.2). Thus, we also provide means to use data object attributes for control flow decisions and to automatically update these attributes. A solution based on the data model accompanying the process model is given in Section 8.3. In this section, we provide means to update a data object based on information from the activity label; e. g., label *Update supplier to $supplier* of an activity having data node *component CP* as input indicates that the supplier of component shall be set to the value of the given process variable. Additionally, the state attribute is used for decision taking at split gateways. This is also covered by some pattern under the umbrella of attribute handling. Third, an activity might be of type multi-instance task. Each multi-instance task be remodeled as multi-instance subprocess containing the activity and the corresponding data nodes in their single type. This way, the patterns for single-instance activities can be applied to multi-instance activities as well.

The remainder of this section presents the patterns for each vertical category as well as the three additionally mentioned ones starting with the patterns for the case object. For each vertical category, we first present the required data model. Then, we provide a two-column table showing process fragments and corresponding SQL queries followed by a textual discussion on each single row of this table in separate para-

graphs. The additional categories are presented analogously without a data model.

Patterns for Case Object

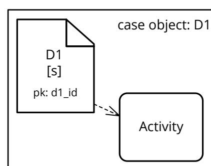
The data model required to derive SQL queries for the case object is presented in Figure 93. Except for the data class of the case object *D1*, no data class is considered for query generation. Following our approach, this class requires two attributes: a primary key attribute *d1_id* and a state attribute *state*. Table 10 shows the seven patterns identified for case object handling.

```
D1
-d1_id
-state
```

Figure 93: Data model for case object; no other data class is of relevance for these patterns.

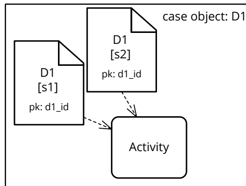
Table 10: Patterns for case object.

CR1 – Read single state



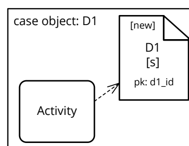

```
guard:
  (SELECT COUNT(D1.d1_id)
  FROM D1
  WHERE D1.d1_id = $ID
  AND D1.state = 's') ⩾ 1
```

CR2 – Read multiple states



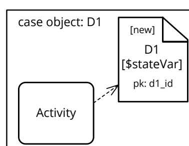

```
guard:
  (SELECT COUNT(D1.d1_id)
  FROM D1
  WHERE D1.d1_id = $ID
  AND D1.state = ('s1' OR 's2')) ⩾ 1
```

CC1 – Create single state




```
INSERT INTO D1
(d1_id, state)
VALUES ($ID,'s')
```

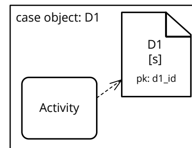
CC2 – Create multiple states




```
INSERT INTO D1
(d1_id, state)
VALUES ($ID,$stateVar)
```

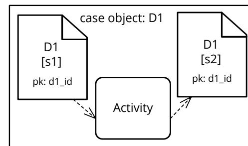
Table 10: Patterns for case object (ctd.).

CU1 – Update



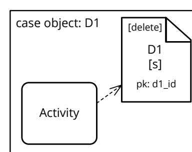

```
UPDATE D1
SET state = 's'
WHERE D1.d1_id = $ID
```

CU2 – Update with required input




```
UPDATE D1
SET state = 's2'
WHERE D1.d1_id = $ID
AND D1.state = 's1'
```

CD1 – Delete




```
DELETE FROM D1
WHERE D1.d1_id = $ID
AND D1.state = 's'
```

CR1 – READ SINGLE STATE. The pattern describes a read operation on the case object of the surrounding scope. *Read* requires that the corresponding instance (i. e., the case object instance, which is related to the current scope instance via its primary key value) is available in state *s*. This is checked by the SQL query to the right that returns all rows of the respective database table for the case object which are related to $\$ID$ and have state *s*. The guard ensures that the activity is only enabled if the result set of the query returns 1 or more rows.

CR2 – READ MULTIPLE STATES. The pattern describes a read operation on the case object similar to CR1, but it allows that the data node can be present in different states. In the pattern, the corresponding instance has to be available either in state *s1* or in state *s2*. As described in the SQL query, all rows of the case object table are counted which are related to $\$ID$ and have *s1* or *s2* as state value. The activity is enabled if one or more rows are returned.

CC1 – CREATE SINGLE STATE. The pattern describes a create operation on the case object. *Create* results in a new entry in the case object table with $\$ID$ of the current instance as primary key value and *s* as state value. This is achieved by executing the given SQL query at the termination of the activity.

CC2 – CREATE MULTIPLE STATES. The pattern describes a create operation on the case object similar to CC1, but the state is not statically given by the process model; it is dynamically set during activity execution by means of a process variable. Executing the given SQL query, a new entry is added to the case object table with $*ID* as primary key value and the value of the process variable $*stateVar* as state value.

CU1 – UPDATE. The pattern describes an update operation on the case object. At the termination of the activity, a new state is set for the corresponding case object instance. In terms of database design, the state value of the corresponding row in the case object table related to $*ID* is updated to *s* as shown in the SQL query. Alternatively, also the process variable $*stateVar* can be used in the update statement for dynamically setting the state during activity execution as done in pattern CC2.

CU2 – UPDATE WITH REQUIRED INPUT. The pattern describes an update operation on the case object similar to pattern CU1, but it additionally requires that the current case object instance is in the given state of the data input. The corresponding SQL query selects only the row related to $*ID* with the state value *s1* and updates it to *s2*.

CD1 – DELETE. The pattern describes a delete operation on the case object. At the termination of the activity, the corresponding case object instance is deleted, whereby the instance has to be in the given state. This is covered by the SQL query, which considers the given state *s* in the WHERE-clause in order to avoid the deletion of wrong data objects.

Patterns for Dependent$^{1:1}$ *Objects*

Next, we describe the patterns and their SQL queries for single-instance data objects, which are in 1:1 relationship with another object and which are dependent to the case object. These patterns consider the generalized case where the dependent data object of class *D2* has no foreign key directly pointing to the case object of class *D1* but rather to another data object of class *D3*, which itself points to *D1* directly or indirectly through further data objects of different classes. The data dependencies are expressed in the data model shown in Figure 94. In the data model, the case object is required to have the following attributes: a primary key attribute *d1_id* and a state attribute *state*. The dependent single-instance data object of class *D2*, which is in the focus of the subsequent queries, has besides the primary key attribute *d2_id* and the state attribute *state* also a foreign key attribute *d3_id* pointing to *D3*. This holds analogously for the other dependent data objects of classes *D3*, ..., *Dn*. From the data model, we can find a path *D2*, *D3*, ..., *Dn*, *D1* of data classes (or tables) from *D2* to *D1* along the foreign key relations. In terms of a database design, the inner join on all tables *D2*,

D3, ..., Dn, D1 connects entries in *D2* with entries in *D1* using the respective identifiers as join attribute. We define the *JOINALL* statement to build this join for our queries, e. g., *JOINALL(D2, D3, D4, D1)* resolves to *(((D2 INNER JOIN D3 USING d3_id) INNER JOIN D4 USING d4_id) INNER JOIN D1 USING d1_id)*. Table 11 shows the nine patterns identified for handling dependent objects in 1:1-fashion.

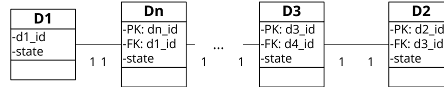
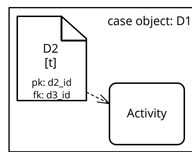


Figure 94: Data model for dependent$^{1:1}$ objects.

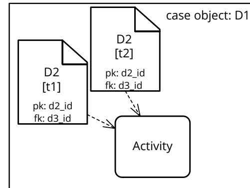
Table 11: Patterns for dependent$^{1:1}$ objects.

$D^{1:1}$R1 – Read single state



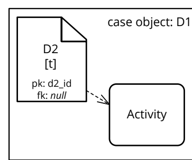

```
guard:
  (SELECT COUNT(D2.d2_id)
  FROM JOINALL(D2, D3, ..., Dn, D1)[4]
  WHERE D1.d1_id = $ID
  AND D2.state = 't') ⩾ 1
```

$D^{1:1}$R2 – Read multiple states



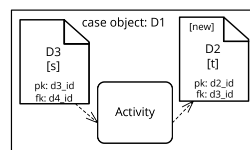

```
guard:
  (SELECT COUNT(D2.d2_id)
  FROM JOINALL(D2, D3, ..., Dn, D1)
  WHERE D1.d1_id = $ID
  AND D2.state =
  ('t1' OR 't2')) ⩾ 1
```

$D^{1:1}$R3 – Read without foreign key




```
guard:
  (SELECT COUNT(D2.d2_id)
  FROM D2
  WHERE D2.d3_id IS NULL
  AND D2.state = 't') ⩾ 1
```

$D^{1:1}$C1 – Create single state

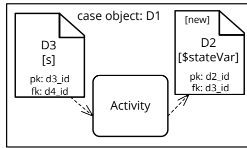



```
INSERT INTO D2
(d2_id, d3_id, state)
VALUES (DEFAULT, (SELECT D3.d3_id
FROM JOINALL(D3, ..., Dn, D1)
WHERE D1.d1_id = $ID), 't')
```

4 *JOINALL(D2, D3, D4, D1) = (((D2 INNER JOIN D3 USING d3_id) INNER JOIN D4 USING d4_id) INNER JOIN D1 USING d1_id)*

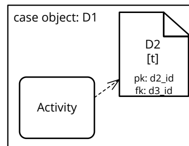
Table 11: Patterns for dependent$^{1:1}$ objects (ctd.).

$D^{1:1}C2$ – Create multiple states



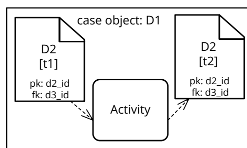

```
INSERT INTO D2
(d2_id, d3_id, state)
VALUES (DEFAULT,(SELECT D3.d3_id
FROM JOINALL(D3, ..., Dn, D1)
WHERE D1.d1_id = $ID),
$stateVar)
```

$D^{1:1}U1$ – Update



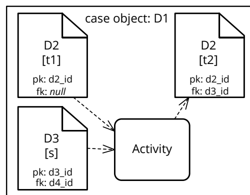

```
UPDATE D2
SET state = 't'
WHERE D2.d3_id = (
  SELECT D3.d3_id
  FROM JOINALL(D3, ..., Dn, D1)
  WHERE D1.d1_id = $ID)
```

$D^{1:1}U2$ – Update with required input



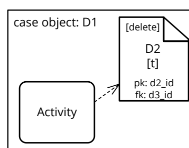

```
UPDATE D2
SET state = 't2'
WHERE D2.D3_id = (
  SELECT D3.d3_id
  FROM JOINALL(D3, ..., Dn, D1)
  WHERE D1.d1_id = $ID)
AND D2.state = 't1'
```

$D^{1:1}U3$ – Update missing foreign key




```
UPDATE D2
SET d3_id = (
  SELECT D3.d3_id
  FROM JOINALL(D3, ..., Dn, D1)
  WHERE D1.d1_id = $ID),
state = 't2'
WHERE D2.d3_id IS NULL
AND D2.state = 't1'
```

$D^{1:1}D1$ – Delete




```
DELETE FROM D2
WHERE D2.d3_id = (
  SELECT D3.d3_id
  FROM JOINALL(D3, ..., Dn, D1)
  WHERE D1.d1_id = $ID)
AND D2.state='t'
```

$D^{1:1}R1$ – READ SINGLE STATE. This pattern describes a read operation on a dependent single-instance data object. *Read* requires that

the respective data object of class *D2* is available in state *t*. Using the statement *JOINALL(D2, D3, ..., D1)*, we can build the join-table between *D2* and *D1* by means of the foreign key relations. In the join-table, each row with *d1_id* = $*ID* describes an instance of class *D2* that is related to the case object instance of the corresponding scope instance. This is used by the SQL query to return all rows of the respective database table for *D2* which are related to $*ID* and have state *t*. The guard ensures that the activity is only enabled if the result set of the query returns 1 or more rows.

$D^{1:1}R2$ – READ MULTIPLE STATES. The pattern describes a read operation on a dependent single-instance data object similar to $D^{1:1}R1$, but it allows that the corresponding data node can be present in different states. In the pattern, the corresponding instance has to be available either in state *t1* or in state *t2*. As described in the SQL query, all rows of the data object table of *D2* are counted which are related to $*ID* and have *t1* or *t2* as state value. The activity is enabled if one or more rows are returned.

$D^{1:1}R3$ – READ WITHOUT FOREIGN KEY. The pattern describes a read operation on a dependent single-instance data object for which the foreign key value is not yet set, i. e., the data object is not yet correlated to a scope instance. The activity is enabled, if any instance of class *D2* exists with an empty foreign key relationship and being in state *t*. Covered by the corresponding SQL query, all rows of the data object table of *D2* are counted which have a *null*-value for the foreign key *d3_id* and *t* as state value. If one or more rows are returned, the activity can be started.

$D^{1:1}C1$ – CREATE SINGLE STATE. The pattern describes a create operation on a dependent single-instance data object. *Create* results in a new entry in the data object table of class *D2* with a default primary key value, the respective primary key value of the *D3* object as foreign key value, and *t* as state value. The respective primary key value of the *D3* object is extracted by joining the table of *D3* with the case object table *D1* by the *JOINALL* statement and selecting the *d3_id* value of the row with *d1_id* = $*ID*, which is related to the current scope instance. This select statement is considered from the SQL query inserting a new row for *D2* at the termination of the activity.

$D^{1:1}C2$ – CREATE MULTIPLE STATES. The pattern describes a create operation on a dependent single-instance data object similar to $D^{1:1}C1$, but the state is not statically given by the process model; it is dynamically set during activity execution by means of a process variable. A new entry is added to the table of class *D2* with a default primary key value, the respective primary key value of the *D3* object as foreign key

value, and the process variable value of $*stateVar* as state value covered by the corresponding SQL query.

$D^{1:1}U1$ – UPDATE. The pattern describes an update operation on a dependent single-instance data object. At the termination of the activity, a new state is set for the corresponding data object. In terms of database design, the state value of the corresponding row in the data object table of class *D2* related to $*ID* is updated to *t* as shown in the SQL query. For the update, the row is selected where the foreign key value *d3_id* points to an entry in the table of class *D3* which is related to $*ID* determined by means of the *JOINALL* statement from *D3* to the case object *D1*. Alternatively, also the process variable $*stateVar* can be used in the update statement for dynamically setting the state during activity execution as done in pattern $D^{1:1}C2$.

$D^{1:1}U2$ – UPDATE WITH REQUIRED INPUT. The pattern describes an update operation on a dependent single-instance data object similar to pattern $D^{1:1}U1$, but it additionally requires that the respective data object is in the given state of the data input. The corresponding SQL query only selects the row related to $*ID* with the state value *t1* and updates it to *t2*.

$D^{1:1}U3$ – UPDATE MISSING FOREIGN KEY. The pattern describes an update operation on a dependent single-instance data object, which has a not yet specified foreign key. Goal of this pattern is to link an uncorrelated data object of class *D2* to a scope instance by setting the corresponding foreign key value. The assignment is done randomly: The uncorrelated instance is taken and processed by this scope instance which is currently running. Thereby, the foreign key value is extracted by selecting the primary key value of the corresponding data object of class *D3* shown as input data node. For the select statement, the *JOINALL* statement is used to join the table of *D3* with the case object table *D1* and to choose the *d3_id* value of the row with *d1_id* = $*ID*, which is related to the current scope instance. For the update, the row of the table of class *D2* is selected which has currently a *null*-value for *d3_id* and *t1* as state value. Then, the foreign key *d3_id* is set to a concrete value and the state is set to *t2*. This is covered by the corresponding SQL query, which is executed at the termination of the activity.

$D^{1:1}D1$ – DELETE. The pattern describes a delete operation on a dependent single-instance data object. At the termination of the activity, the corresponding data object is deleted, whereby the instance has to be in the state given by the data node. This is covered by the SQL query, which also considers the given state *t* in the WHERE-clause in order to avoid the deletion of wrong data objects. For the deletion, the table row of class *D2* is selected where the foreign key value *d3_id* points to an

entry in the table of *D3* which is related to *$ID* determined by means of the *JOINALL* statement from *D3* to the case object *D1*. This is covered by the corresponding SQL query.

Patterns for Dependent$^{1:n}$ Objects

Next, we describe the patterns and their SQL queries for multi-instance data objects, which are in 1:n relationship with another object and which are dependent to the case object. These patterns consider the generalized case where the dependent data object of class *D2* has no foreign key directly pointing to the case object of class *D1* but rather to another data object of class *D3*, which itself points to *D1* directly or indirectly through further data objects of different classes. The data dependencies are expressed in the data model shown in Figure 95. In the data model, the case object is required to have the following attributes: a primary key attribute *d1_id* and a state attribute *state*. The dependent multi-instance data object of class *D2*, which is in the focus of the subsequent queries, has besides the primary key attribute *d2_id* and the state attribute *state* also a foreign key attribute *d3_id* pointing to *D3*. This holds analogously for the other dependent data objects of classes *D3*, ..., *Dn*. We again utilize the *JOINALL* statement as means for joins with the case object. Table 12 shows the nine patterns identified for handling dependent objects in 1:n-fashion.

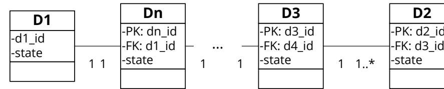
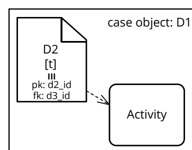


Figure 95: Data model for dependent$^{1:n}$ objects.

Table 12: Patterns for dependent$^{1:n}$ objects.

$D^{1:n}R1$ – Read single state

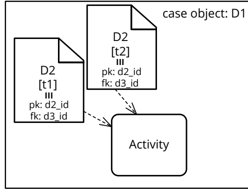



```
guard:
  (SELECT COUNT(D2.d2_id)
  FROM JOINALL(D2, D3, ..., Dn,D1)[5]
  WHERE D1.d1_id = $ID
  AND D2.state = 't') =
  (SELECT COUNT(D2.d2_id)
  FROM JOINALL(D2, D3, ..., Dn, D1)
  WHERE D1.d1_id = $ID )
```

5 *JOINALL(D2, D3, D4, D1) = (((D2 INNER JOIN D3 USING d3_id) INNER JOIN D4 USING d4_id) INNER JOIN D1 USING d1_id)*

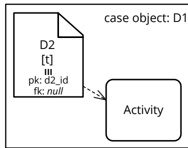
Table 12: Patterns for dependent$^{1:n}$ objects (ctd.).

$D^{1:n}R2$ – Read multiple states



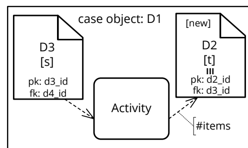

```
guard: (
  (SELECT COUNT(D2.d2_id)
  FROM JOINALL(D2, D3, ..., Dn, D1)
  WHERE D1.d1_id = $ID
  AND D2.state= 't1') =
  (SELECT COUNT(D2.d2_id)
  FROM JOINALL(D2, D3, ..., Dn, D1)
  WHERE D1.d1_id = $ID ))
  xor (
  (SELECT COUNT(D2.d2_id)
  FROM JOINALL(D2, D3, ..., Dn, D1)
  WHERE D1.d1_id = $ID
  AND D2.state = 't2') =
  (SELECT COUNT(D2.d2_id)
  FROM JOINALL(D2, D3, ..., Dn, D1)
  WHERE D1.d1_id = $ID ))
```

$D^{1:n}R3$ – Read without foreign key



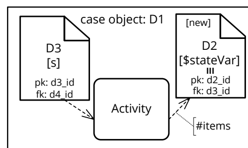

```
guard:
  (SELECT COUNT(D2.d2_id)
  FROM D2
  WHERE D2.d3_id IS NULL
  AND D2.state = 't') ⩾ 1
```

$D^{1:n}C1$ – Create single state




```
INSERT INTO D2
(d2_id, d3_id, state) VALUES
(DEFAULT, fk, 't')
...
(DEFAULT, fk, 't')
//#items times

fk = SELECT D3.d3_id
FROM JOINALL(D3,...,Dn,D1)
WHERE D1.d1_id=$ID
```

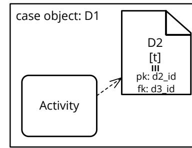
$D^{1:n}C2$ – Create multiple states




```
INSERT INTO D2
(d2_id, d3_id, state) VALUES
(DEFAULT, fk, $stateVar)
...
(DEFAULT, fk, $stateVar)
//#items times

fk = SELECT D3.d3_id
FROM JOINALL(D3, ..., Dn, D1)
WHERE D1.d1_id = $ID
```

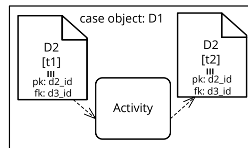
Table 12: Patterns for dependent$^{1:n}$ objects (ctd.).

$D^{1:n}U1$ – Update



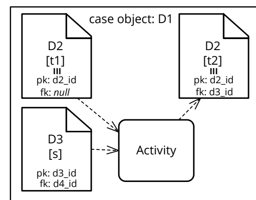

```
UPDATE D2
SET state = 't'
WHERE D2.d3_id = (
  SELECT D3.d3_id
  FROM JOINALL(D3, ..., Dn, D1)
  WHERE D1.d1_id = $ID)
```

$D^{1:n}U2$ – Update with required input



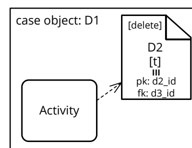

```
UPDATE D2
SET state = 't2'
WHERE D2.d3_id = (
  SELECT D3.d3_id
  FROM JOINALL(D3, ..., Dn, D1)
  WHERE D1.d1_id=$ID)
AND D2.state = 't1'
```

$D^{1:n}U3$ – Update missing foreign key




```
UPDATE D2
SET d3_id = (
  SELECT D3.d3_id
  FROM JOINALL(D3, ..., Dn, D1)
  WHERE D1.d1_id = $ID),
state = 't2'
WHERE D2.d3_id IS NULL
AND D2.state = 't1'
```

$D^{1:n}D1$ – Delete




```
DELETE FROM D2
WHERE D2.d3_id = (
  SELECT D3.d3_id
  FROM JOINALL (D3, ..., Dn, D1)
  WHERE D1.d1_id = $ID)
AND D2.state='t'
```

$D^{1:n}R1$ – READ SINGLE STATE. This pattern describes a read operation on dependent multi-instance data objects. *Read* requires that the respective set of data objects of class *D2* is available in state *t*. Using the statement *JOINALL(D2, D3, ..., D1)*, we can build the join-table between *D2* and *D1* by means of the foreign key relations. In the join-table, each row with *d1_id* = $*ID* describes an instance of class *D2* that is related to the case object instance of the corresponding scope instance. This is used by the SQL query to return all rows of the respective database table for *D2* which are related to $*ID* and have state *t* (first select) and to return

all rows of this table which are related to $*ID* independently from the state attribute (second select). The guard returns true if and only if both selects return the same number of entries; i. e., the guard ensures that the activity is only enabled if all objects related to $*ID* are in state *t*.

$D^{1:n}R2$ – READ MULTIPLE STATES. The pattern describes a read operation on dependent multi-instance data objects similar to $D^{1:n}R1$, but it allows that the data objects can be present in different states. In the pattern, the corresponding instances have to be available either in state *t1* or in state *t2*; a mixture of states is not allowed (cf. execution semantics in Section 4.7). This is ensured by the guard expression that connects two (or more) SQL queries as given for $D^{1:n}R1$ by *XOR* constructs. For state *t1*, all rows of the data object table of class *D2* are counted which are related to $*ID* and have *t1* as state value. These are compared to all rows being related to $*ID* independently from the state. If both return the same number, the condition holds true and the activity gets enabled. The same check is done for each other potential state where all rows related to $*ID* and having that state (*t2* in this pattern) are compared to all rows being related to $*ID* independently from the state. The activity is enabled as soon as one of the conditions holds true.

$D^{1:n}R3$ – READ WITHOUT FOREIGN KEY. The pattern describes a read operation on dependent multi-instance data objects for which the foreign key value is not yet set, i. e., the data objects are not yet correlated to a scope instance. The activity is enabled, if any set of instances of class *D2* exists, where each object has the same empty foreign key relationship and is in state *t*. Covered by the SQL query, all rows of the data object table of class *D2* are counted which have a *null*-value for the foreign key *d3_id* and *t* as state value. If one or such entries exist, the activity is enabled.

$D^{1:n}C1$ – CREATE SINGLE STATE. The pattern describes a create operation on dependent multi-instance data objects. *Create* results in new entries in the data object table of class *D2*, each with a default primary key value, the respective primary key value of the *D3* object as foreign key value, and *t* as state value. The respective primary key value of the *D3* object is extracted by joining the table of *D3* with the case object table *D1* by the *JOINALL* statement and selecting the *d3_id* value of the row with *d1_id* = $*ID*, which is related to the current scope instance. This query is executed first and the returned foreign key value is saved in the variable *fk* for application in the create statement. The variable *fk* is utilized for each insertion of a new row for the objects of class *D2* at the termination of the activity. The number objects to be created is determined by the expression *#items*, which is attached to the output data flow edge.

$D^{1:n}C2$ – CREATE MULTIPLE STATES. The pattern describes a create operation on dependent multi-instance data objects similar to $D^{1:n}C1$, but the state is not statically given by the process model; it is dynamically set during activity execution by means of a process variable. Each new entry is added to the data object table of class *D2* with a default primary key value, the respective primary key value of the *D3* object as foreign key value, and the value of the process variable $*stateVar* as state value covered by the given SQL query. Similar to $D^{1:n}C1$, the number of data objects to be created is determined by the expression *#items*, which is attached to the output data flow edge of the activity.

$D^{1:n}U1$ – UPDATE. The pattern describes an update operation on dependent multi-instance data objects. At the termination of the activity, a new state is set for each of the corresponding data objects. In terms of database design, the state values of the corresponding rows in the data object table of class *D2* related to $*ID* are updated to *t*. For the update, all rows are selected where the foreign key value *d3_id* points to an entry in the table of class *D3* which is related to $*ID* determined by means of the *JOINALL* statement from *D3* to the case object *D1*. Alternatively, also the process variable $*stateVar* can be used in the update statement for dynamically setting the state during activity execution as done in pattern $D^{1:n}C2$.

$D^{1:n}U2$ – UPDATE WITH REQUIRED INPUT. The pattern describes an update operation on dependent multi-instance data objects similar to pattern $D^{1:n}U1$, but it additionally requires that the respective data objects are in the given state of the data input. The SQL query selects the rows related to $*ID* with the state value *t1* and updates them to *t2*.

$D^{1:n}U3$ – UPDATE MISSING FOREIGN KEY. The pattern describes an update operation on dependent multi-instance data objects, which have a not yet specified foreign key. Goal of this pattern is to link uncorrelated data objects of class *D2* to a scope instance by setting the corresponding foreign key value. The assignment is done randomly: The uncorrelated objects are taken and processed by this scope instance which is currently running. Thereby, the foreign key value is extracted by selecting the primary key value of the corresponding data object of class *D3* shown as input data node. For the select statement, the *JOINALL* statement is used to join the table of class *D3* with the case object table of *D1* and to select the *d3_id* value of the row where *d1_id* = $*ID*, i. e., the row which is related to the current scope instance. For the update, all rows of class *D2* are selected which have currently a *null*-value for *d3_id* and *t1* as state value. Then, the foreign key *d3_id* is set to the selected concrete value and the state is set to *t2*. This query is executed upon termination of the activity.

D$^{1:n}$D1 – DELETE. The pattern describes a delete operation on dependent multi-instance data objects. At the termination of the activity, the corresponding data objects are deleted, whereby the instances have to be in the given state. This is covered by the given SQL query, which also considers the given state *t* in the WHERE-clause in order to avoid the deletion of wrong data objects. For the deletion, all rows of class *D2* are selected where the foreign key value *d3_id* points to an entry in the table of class *D3* which is related to $*ID* determined by means of the *JOINALL* statement from *D3* to the case object *D1*.

Patterns for Dependent$^{m:n}$ Objects

Next, we describe the patterns and their SQL queries for multi-instance data objects, which represent a m:n relationship between two other data objects to which they are dependent. Additionally, these two data objects are in 1:n relationship with another data object. Both are dependent to the case object and may point directly or indirectly to the case object. The data dependencies are expressed in the data model shown in Figure 96. In the data model, the case object of class *D1* is required to have the following attributes: a primary key attribute *d1_id* and a state attribute *state*. The dependent multi-instance data objects of classes *D3* and *D4* have besides the primary key attribute *d3_id* and *d4_id* respectively and the state attribute *state* also a foreign key attribute *d3_id* and *d4_id* respectively pointing to the corresponding class. This holds analogously for the other dependent data objects of classes *D5*, *D6*, ..., *Dn*. The dependent multi-instance data object of class *D2*, which is in the focus of the subsequent queries, has a primary key attribute *d2_id*, a state attribute *state*, and additionally a set of two foreign key attributes *d3_id* and *d4_id*. We again utilize the *JOINALL* statement as means for joins with the case object.

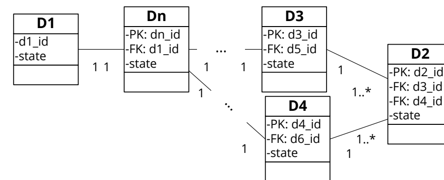


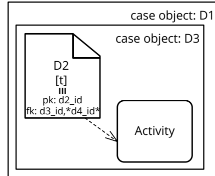
Figure 96: Data model for dependent$^{m:n}$ objects.

As shown in the data model, several data object of class *D3* (and *D4* respectively) are related indirectly to one instance of the case object and in turn, each instance of *D3* (and *D4* respectively) relates to multiple instances of data objects of class *D2*. Thus, several instance subsets of *D2* can be observed each belonging to one instance of *D3* (and *D4* respectively). For queries on such a m:n data object, the process modeler can decide if the set of all data objects or whether specific subsets are needed. We differentiate between all subsets (meaning all data objects) and a specific subset by means of asterisks. If a foreign key is sur-

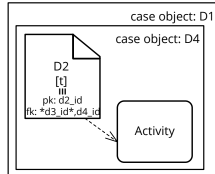
rounded by these asterisks, all subsets are utilized for the query and if not, only one specific subset is utilized (cf. all-quantified foreign keys in Definition 4.2 on page 62). We will use this notation in the following patterns. Table 13 shows the twelve patterns identified for handling dependent objects in m:n-fashion.

Table 13: Patterns for dependent$^{m:n}$ objects.

$D^{m:n}R1$ – Read subset

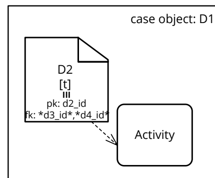



```
guard:
  (SELECT COUNT(D2.d2_id)
  FROM JOINALL (D2, D3)⁶
  WHERE D3.d3_id = $ID
  AND D2.state = 't') =
  (SELECT COUNT(D2.d2_id)
  FROM JOINALL (D2, D3)
  WHERE D3.d3_id = $ID )
```



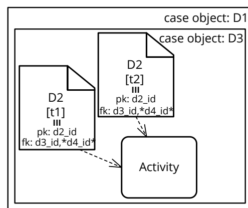

```
guard:
  (SELECT COUNT(D2.d2_id)
  FROM JOINALL (D2, D4)
  WHERE D4.d4_id = $ID
  AND D2.state = 't') =
  (SELECT COUNT(D2.d2_id)
  FROM JOINALL (D2, D4)
  WHERE D4.d4_id = $ID )
```

$D^{m:n}R2$ – Read multiple subset




```
guard:
  (SELECT COUNT(D2.d2_id)
  FROM JOINALL (D2,D3,...,Dn,D1)
  WHERE D1.d1_id = $ID
  AND D2.state = 't') =
  (SELECT COUNT(D2.d2_id)
  FROM JOINALL (D2, D3, ..., Dn, D1)
  WHERE D1.d1_id = $ID)
```

$D^{m:n}R3$ – Read multiple states

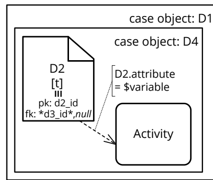



```
guard:(
  (SELECT COUNT(D2.d2_id)
  FROM JOINALL (D2, D3)
  WHERE D3.d3_id = $ID
  AND D2.state = 't1') =
  (SELECT COUNT(D2.d2_id)
  FROM JOINALL (D2, D3)
  WHERE D3.d3_id = $ID))
  xor (
  (SELECT COUNT(D2.d2_id)
  FROM JOINALL (D2, D3)
  WHERE D3.d3_id = $ID
  AND D2.state = 't2') =
  (SELECT COUNT(D2.d2_id)
  FROM JOINALL (D2, D3)
  WHERE D3.d3_id = $ID))
```

6 *JOINALL(D2, D3, D4, D1) = (((D2 INNER JOIN D3 USING d3_id) INNER JOIN D4 USING d4_id) INNER JOIN D1 USING d1_id)*

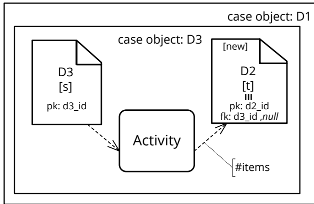
Table 13: Patterns for dependent$^{m:n}$ objects (ctd.).

$D^{m:n}R4$ – Read without foreign key



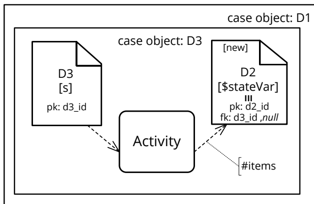

```
guard:
  (SELECT COUNT(D2.d2_id)
  FROM JOINALL (D2, D3, ..., Dn, D1)
  WHERE D1.d1_id = $ID
  AND D2.state = 't'
  AND D2.d4_id IS NULL
  AND D2.attribute = $variable) >= 1
```

$D^{m:n}C1$ – Create single state



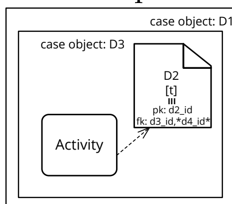

```
INSERT INTO D2
(d2_id, d3_id, d4_id, state) VALUES
(DEFAULT, fk,   NULL, 't')
...
(DEFAULT, fk, NULL, 't')
//#items times

fk = SELECT D3.d3_id
FROM D3
WHERE D3.d3_id = $ID
```

$D^{m:n}C2$ – Create multiple states



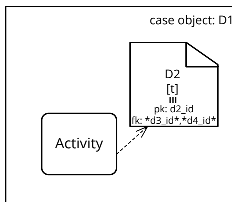

```
INSERT INTO D2
(d2_id, d3_id, d4_id, state) VALUES
(DEFAULT, fk, NULL, $stateVar)
...
(DEFAULT, fk,   NULL, $stateVar)
//#items times

fk = SELECT D3.d3_id
FROM D3
WHERE D3.d3_id=$ID
```

$D^{m:n}U1$ – Update subset




```
UPDATE D2
SET state = 't'
WHERE D2.d3_id = (
  SELECT D3.d3_id FROM d3
  WHERE d3_id = $ID)
```

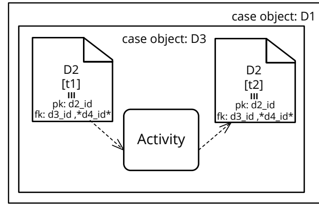
$D^{m:n}U2$ – Update multiple subsets




```
UPDATE D2
SET state = 't'
WHERE D2.d3_id = (
  SELECT D3.d3_id
  FROM JOINALL(D3, ..., Dn, D1)
  WHERE D1.d1_id = $ID)
```

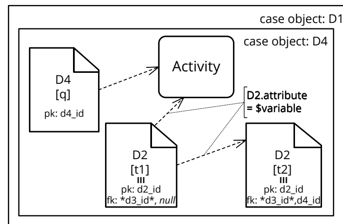
Table 13: Patterns for dependent$^{m:n}$ objects (ctd.).

$D^{m:n}U3$ – Update with required input



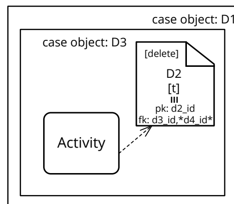

```
UPDATE D2
SET state = 't2'
WHERE D2.d3_id = (
   SELECT D3.d3_id
         FROM D3
   WHERE D3.d3_id = $ID)
AND D2.state = 't1'
```

$D^{m:n}U4$ – Update missing foreign key



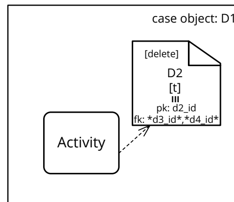

```
UPDATE D2
SET d4_id = (
   SELECT D4.d4_id
   FROM D4
   WHERE D4.d4_id = $ID),
state = 't1'
WHERE D2.d3_id = (
   SELECT D3.d3_id
   FROM JOINALL(D3, ..., Dn, D1)
   WHERE D1.d1_id = $ID)
AND D2.state = 't2'
AND D2.d4_id IS NULL
AND D2.attribute = $variable
```

$D^{m:n}D1$ – Delete subset




```
DELETE FROM D2
WHERE D2.d3_id = (
   SELECT D3.d3_id
         FROM D3
   WHERE D3.d3_id = $ID)
AND D2.state = 't'
```

$D^{m:n}D2$ – Delete multiple subsets




```
DELETE FROM D2
WHERE D2.d3_id = (
   SELECT D3.d3_id
   FROM JOINALL(D3, ..., Dn, D1))
   WHERE D1.d1_id = $ID)
AND D2.state = 't'
```

$D^{m:n}R1$ – READ SUBSET. This pattern describes a read operation on a specific data object subset of dependent m:n data objects. In the upper pattern, the foreign key *d3_id* is not surrounded by asterisks; this indicates that a subset is requested belonging to a particular data object of class *D3* being the case object of the surrounding scope. *Read* requires that the respective subset of data objects of class *D2* is available in state *t*. Using the statement *JOINALL(D2, D3)*, we can build the join-table between

m:n data objects of class *D2* and the case object of class *D3* by means of their foreign key relations. In the join-table, each row with *d3_id* = $*ID* describes an instance of class *D2* that is related to the case object instance of the corresponding scope instance. This is used by the SQL query to return all rows of the respective database table for the m:n data objects of class *D2* which are related to $*ID* and have state *t* (first select) and to return all rows which are related to $*ID* independently from the state attribute (second select). The guard returns true if and only if both selects return the same number of entries; i. e., the guard ensures that the activity is only enabled if all data objects of the specific subset related to $*ID* are in state *t*. The same applies if a subset is requested where the m:n data objects of class *D2* belong to a particular data object of class *D4*; the lower pattern shows this case.

$D^{m:n}R2$ – READ MULTIPLE SUBSETS. This pattern describes a read operation on all subsets of dependent m:n data objects. Both foreign key attributes of the data objects of class *D2* are surrounded by asterisks; this indicates that no specific instance subset of *D2* is requested but rather all subsets relating to the case object of class *D1* of the surrounding scope. *Read* requires that the respective subsets of data objects of class *D2* are available in state *t*. Using the statement *JOINALL(D2,D3,...Dn,D1)*, we can build the join-table between m:n data objects of class *D2* and the case object of class *D1* by means of their foreign key relations. For the join, both foreign key relations of objects of class *D2* can be used supposing that the related data objects of classes *D3* and *D4* are in turn in a direct or indirect relation with the case object of class *D1* as shown in the data model in Figure 96. Thus, the SQL query compares the number of rows in the table of class *D2* where corresponding objects are related to $*ID* and are in state *t* to all rows where objects are related to $*ID* independently from the data state. The activity gets enabled if both select statements return the same number.

$D^{m:n}R3$ – READ MULTIPLE STATES. The pattern describes a read operation on a specific data object subset of dependent m:n data objects similar to $D^{m:n}R1$, but it allows that the data objects can be present in different states (though all objects of one subset must be in the same data state). Thus, all instances of the subset with objects of class *D2* corresponding to a particular object of class *D3* have to be available either in state *t1* or in state *t2* in this pattern; a mixture of states is not allowed (cf. execution semantics in Section 4.7). This is ensured by the guard expression that connects two (or more) SQL queries as given for $D^{m:n}R1$ by *XOR* constructs. For state *t1*, all rows of the data object table of class *D2* are counted which are related to $*ID* and have *t1* as state value. These are compared to all rows being related to $*ID* independently from the state. If both return the same number, the condition holds true and the activity gets enabled. The same check is

done for each other potential state where all rows related to $*ID* and having that state (*t2* in this pattern) are compared to all rows being related to $*ID* independently from the state. The activity is enabled as soon as one of the conditions holds true.

$D^{m:n}R4$ – READ WITHOUT FOREIGN KEY. The pattern describes a read operation on a specific data object subset of dependent m:n data objects where the objects of this subset do not have specified foreign key values yet. Due to the missing foreign key value, a join with the case object of class *D4* of the directly surrounding scope cannot be created. However, we have to ensure that only data objects of class *D2* are counted which belong to the current process execution. The process is the top-level of a scope hierarchy and has in this pattern an object of class *D1* as case object, since *D1* is the root class in the data model in Figure 96. We assume that a data object of class *D3*, to which an object of class *D2* has already an existing second foreign key relation, is directly related to the process case object (as given in the data model in Figure 96). This foreign key relation to *D3* is used to select the respective rows of data object table of class *D2*. In terms of database design, the rows of the table for class *D2* are counted where the foreign key value *d3_id* points to rows in the table of class *D3* which are in turn related to $*PID* – the current process instance – over their foreign key relation to the process case object. Additionally, these rows for class *D2* have to have value *t* for the state attribute and have to have a *null*-value for the second foreign key attribute. Furthermore, the process designer can provide an expression at the input data flow edge restricting the set of data objects with no foreign key relation being read by the activity. This expression compares a given data object attribute with a process variable being set during process execution. All these aspects are captured by the given SQL query; the activity is enabled if one or more rows are in the result set.

$D^{m:n}C1$ – CREATE SINGLE STATE. The pattern describes a create operation on dependent m:n data objects. *Create* results in new entries representing a specific subset of data objects belonging to a particular instance of class *D3* in the data object table of class *D2*, each entry with a default primary key value, the respective primary key value of the *D3* object as foreign key value, and *t* as state value. The m:n relationships presented by the data objects of class *D2* have to be set in two steps because an activity instance can only relate one instance of *D3* (or *D4* respectively) to an instance subset of *D2*. Therefore, in this pattern, a particular value is set for the foreign key attribute *d3_id* and a *null*-value for the other foreign key attribute *d4_id*. The non-empty foreign key value of an object of class *D2* is extracted by selecting the primary key value *d3_id* from the row in the case object table of class *D3* where *d3_id* = $*ID*. This select statement executed at first and the returned foreign key

value is saved in the variable *fk* for application in the create statement. The variable *fk* is utilized for each insertion of a new row for the objects of class *D2* at the termination of the activity. The number of objects to be created is determined by the expression *#items*, which is attached to the output data flow edge.

$D^{m:n}$C2 – CREATE MULTIPLE STATES. The pattern describes a create operation on dependent m:n data objects similar to $D^{m:n}$C1, but the state is not statically given by the process model; it is dynamically set during activity execution by means of a process variable. Each new entry is added to the data object table of class *D2* with a default primary key value, the respective primary key value of the *D3* object as foreign key value, and the value of process variable $*stateVar* as state value covered by the corresponding SQL query. Similar to $D^{m:n}$C1, the number of objects to be created is determined by the expression *#items*, which is attached to the output data flow edge.

$D^{m:n}$U1 – UPDATE SUBSET. The pattern describes an update operation on a specific data object subset of dependent m:n data objects. At the termination of the activity, a new state is set for each object of the subset, where all objects belong to the same particular data object of class *D3*. In terms of database design, the state values of all rows in the data object table of class *D2* related to the current case object instance with *d3_id* = $*ID* are updated to *t*. Alternatively, also the process variable $*stateVar* can be used in the update statement for dynamically setting the state during activity execution as done in pattern $D^{m:n}$C2.

$D^{m:n}$U2 – UPDATE MULTIPLE SUBSETS. The pattern describes an update operation on all data object subsets of dependent m:n data objects. Both foreign key attributes of the data objects of class *D2* are surrounded by asterisks; this indicates that no specific instance subset of *D2* is requested but rather all subsets relating to the case object of class *D1* of the surrounding scope. At the termination of the activity, a new state is set for the respective objects of all subsets. In terms of database design, the state values of the corresponding rows in the data object table of class *D2* related to $*ID*, i. e., the current process instance, are updated to *t*. For the update, the process instance $*ID* is determined by means of the *JOINALL* statement from one class of an object the *D2* object is directly related to via foreign key relationship (in this pattern either class *D3* or class *D4*) with the case object *D1*. Alternatively, also the process variable $*stateVar* can be used in the update statement for dynamically setting the state during activity execution as done in pattern $D^{m:n}$C2.

$D^{m:n}$U3 – UPDATE WITH REQUIRED INPUT. The pattern describes an update operation on a specific data object subset of dependent m:n

data objects similar to pattern $D^{m:n}U1$, but it additionally requires that all objects of the subset are in the given state of the data input. The SQL query only selects the rows related to $*ID* with the state value *t1* and updates them to *t2*.

$D^{m:n}U4$ – UPDATE MISSING FOREIGN KEY. The pattern describes an update operation on a specific data object subset of dependent m:n data objects, which have one not yet specified foreign key value. This pattern is the continuation of pattern $D^{m:n}C1$. Thereby, the foreign key value for this subset of data objects of class *D2* is extracted by selecting the primary key value of the corresponding data object of class *D4* as shown in the SQL query. This pattern uses an expression at the output data flow edge for specifying which subset of all *D2* instances should be assigned to one specific object of class *D4*. This expression compares a given data attribute with a process variable, which is set during process execution. In the SQL statement, it is used for the update WHERE-clause. Additionally, all data objects of class *D2*, which have a missing foreign key, have to be selected in a similar manner as in pattern $D^{m:n}R4$ over the WHERE-clause.

$D^{m:n}D1$ – DELETE SUBSET. The pattern describes a delete operation on a specific data object subset of dependent m:n data objects. At the termination of the activity, all objects of this subset are deleted. The objects are identified by their relationship to the current case object with *d3_id* = $*ID*, whereby all instances of the subset have to be in the given state *t*. This is covered by the SQL query, which also considers the given state *t* in the WHERE-clause in order to avoid the deletion of wrong data objects.

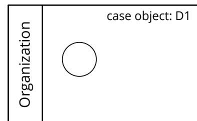
$D^{m:n}D2$ – DELETE MULTIPLE SUBSETS. The pattern describes a delete operation on all subsets of dependent m:n data objects. Both foreign key attributes of data objects of class *D2* are surrounded by asterisks; this indicates that no specific instance subset of *D2* is requested but rather all subsets relating to the case object of class *D1* of the surrounding scope. At the termination of the activity, all data objects being related to the current case object instance with *d1_id* = $*ID* are deleted, whereby all these objects have to be in the given state *t*. This is covered by the SQL query, which also considers the given state *t* in the WHERE-clause in order to avoid the deletion of wrong data objects. The $*ID* is determined by means of the *JOINALL* statement from one class of an object the *D2* object is directly related to via foreign key relationship (in this pattern either class *D3* or class *D4*) with the case object *D1*; the query utilizes *D3*. Thus, for the deletion, all rows of class *D2* are selected where the foreign key value *d3_id* points to an entry in the table of class *D3* which is related to $*ID* through the case object.

Instantiation Patterns

Process and activity instantiation is an essential part of process execution. We specify a set of four instantiation patterns to be able to link the data objects with the process or activity instances from those they are processed. These and the corresponding SQL queries are introduced next. Table 14 shows the four patterns identified for handling process and subprocess instantiation.

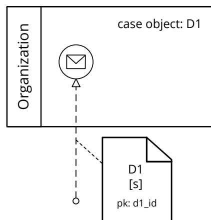
Table 14: Patterns for process and subprocess instantiation.

I1 – Process instantiation without data trigger



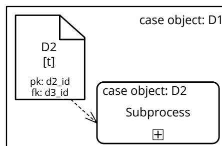

```
Start process instance
with new $ID
```

I2 – Process instantiation with data trigger



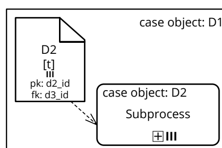

```
Start process instance
with id D1.d1_id
```

I3 – Subprocess instantiation with single data trigger




```
For D2.d2_id ∈ (
  SELECT D2.d2_id
  FROM JOINALL(D2, D3, ..., Dn, D1)
  WHERE D1.d1_id  = $ID)
start subprocess
with id D2.d2_id
```

I4 – Subprocess instantiation with multiple data trigger




```
For each D2.d2_id ∈ (
  SELECT D2.d2_id
  FROM JOINALL(D2, D3, ..., Dn, D1)
  WHERE D1.d1_id  = $ID)
start subprocess
with id D2.d2_id
```

I1 – PROCESS INSTANTIATION WITHOUT DATA TRIGGER. This pattern describes the instantiation of a process without any data trigger by an arbitrary event. The instance of the process gets a unique identifier (id), which is managed by the process engine. As soon as an instance of the case object is created within the process instance, it will receive the id of that process instance as primary key value (see pattern CC1).

I2 – PROCESS INSTANTIATION WITH DATA TRIGGER. This pattern describes the instantiation of a process triggered by a data object, which already exists and is received by the process. At the same time, the class of this data object is specified as case object of the process; in the pattern, it class *D1*. The instantiated process instance gets the primary key value of its case object instance as id in order to correlate these two.

I3 – SUBPROCESS INSTANTIATION WITH SINGLE DATA TRIGGER. This pattern describes the instantiation of a subprocess triggered by its case object *D2*. The instantiated subprocess instance gets the primary key value of the respective case object instance as id. This is captured by the SQL query that selects the primary key value of the row of the database table for *D2* being related to $*ID* of the surrounding scope with *D1* as case object. *D2* and *D1* are joined by using the *JOINALL* statement.

I4 – SUBPROCESS INSTANTIATION WITH MULTIPLE DATA TRIGGERS. This pattern describes the instantiation of a multi-instance subprocess triggered by its multi-instance case object *D2*. For each data object of class *D2* related to the current scope instance, one instance of the subprocess is created, which gets the primary key value of the respective object of class *D2* as id. This is captured by the SQL query that selects the primary key values for all rows of the database table for *D2* being related to $*ID* of the surrounding scope with *D1* as case object. *D2* and *D1* are joined by using the *JOINALL* statement. This pattern also applies for a multi-instance task having a case object (see below for the support of multi-instance tasks).

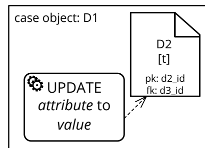
Attribute Patterns

Next, we introduce patterns and their SQL queries to handle database operations on single data object attributes other than primary key, foreign key, and data state although it may be used for them as well. Here, we provide a solution for handling a single attribute (read or update). A solution for multiple attributes based on the data model accompanying the process model is given in Section 8.3. The read of a single attribute is used for checking data conditions on control flow edges originating from a split gateway; in this thesis, the function is limited to the state attribute but may be used for any attribute beyond the scope of this thesis. The update is used to set a single value of some attribute; e. g., the supplier for some component. Table 15 shows the two corresponding patterns identified for handling single attributes of data objects.

A1 – UPDATE ATTRIBUTE. This pattern describes the update of a single data attribute based on activity label information. This attribute may not be represented in the data node, but it is part of the data model that accompanies the process model. Thus, the corresponding attribute and the value (respectively the process variable holding the value), to

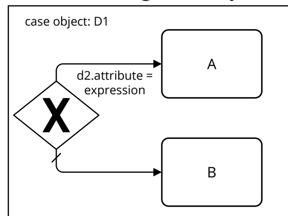
Table 15: Patterns for handling single attributes.

A1 – Update attribute




```
UPDATE D2 SET
attribute = 'value'
WHERE D2.d3_id = (
  SELECT D3.d3_id
  FROM JOINALL (D3, ..., Dn, D1)
  WHERE D1.d1_id = $ID  )
```

A2 – XOR gateway




```
SELECT D2.attribute
FROM JOINALL(D2, D3, ..., Dn, D1)
WHERE D1.d1_id = $ID
```

which it shall be updated, is specified in the label of the task. In the graphical representation, the task is shown as service task to indicate that it is executed automatically without further human interference (after specifying the value to put into the database). Usually, this information is derived dynamically extracted from a process variable. The differentiation whether a process variable or a specific value is given in the task label needs to be done by surrounding code and included into the query accordingly; we propose to tag process variables with a $ sign and to omit tags for exact values of the type as specified in the data model. The output data node, here an object of class *D2* is referenced, indicates on which data object table the update statement is executed. For the update, all data table rows of *D2* are selected where the foreign key value *d3_id* points to an entry in the table of *D3* which is related to $*ID* determined by means of the *JOINALL* statement from *D3* to the case object *D1*.

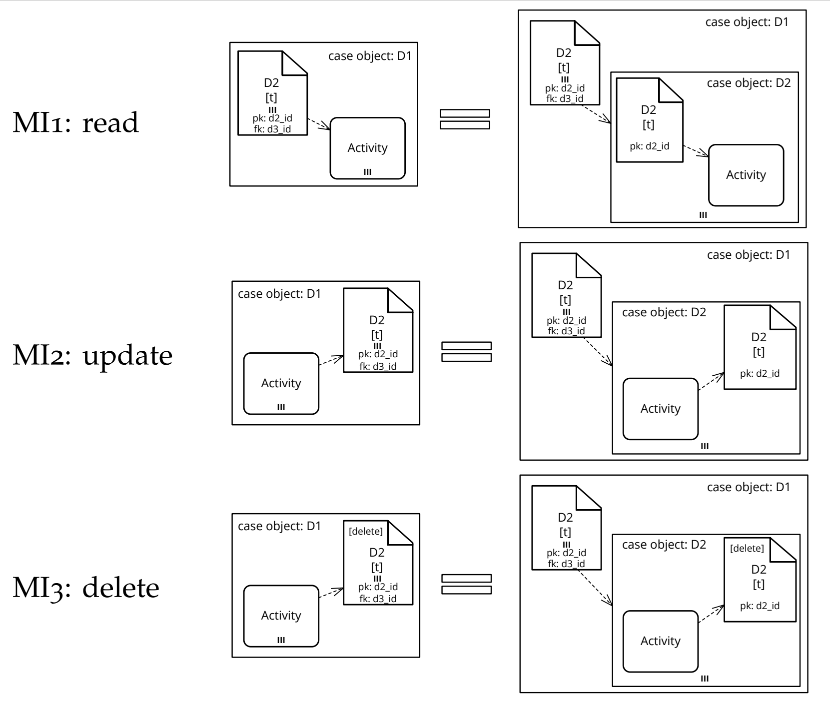
A2 – XOR GATEWAY. This pattern describes how data attributes can be utilized to decide the path to be taken after an exclusive choice (XOR split) in the control flow. The SQL query delivers the current value of the specified attribute belonging to the data object of class *D2*, which relates to $*ID* of the surrounding scope. The correlation to $*ID* is done by means of the *JOINALL* statement with the case object *D1*. The value returned by the query can be checked against the specified expression to reason about the truth value of the condition attached the upper pat. Thereby, "=" is only one possibility, since all comparison operator, e. g., $\leqslant, >$, can be used. In this thesis, the attribute is restricted to be the data state and the expressions is supposed to specify the required state. For shortening the annotation as done throughout all examples in this

thesis, *class[state]* is translated into *class.dataState = state* and handled accordingly as described in the query.

Supporting Multi-Instance Tasks

Above, patterns omitted the existence of multi-instance tasks and covered single-instance tasks as well as single- and multi-instance subprocesses. Next, we describe how multi-instance tasks are covered by above patterns. Multi-instance tasks are very similar to subprocesses from an execution point of view: Several task instances are instantiated from which each is executed independently. Thus, we transform each multi-instance task into a multi-instance subprocess. Thereby, we require that each multi-instance task only contains data associations to exactly one set of multi-instance data nodes; further single-instance data nodes are disregarded from the transformation and copied as defined below. Thus, multi-instance tasks cannot be used for the creation of dependent multi-instance data objects, since they need their related data object being single-instance as input. During the transformation, the multi-instance task is mapped to a single instance task that is then surrounded by a multi-instance subprocess. The associated, multi-instance data node is associated as-is to the subprocess as input (to specify the number of instances to be created; see pattern I4). Additionally, the multi-instance data node is mapped to a single-instance data node and associated with the single-instance activity surrounded by the multi-instance subprocess; input and output properties are not

Table 16: Transforming multi-instance tasks to multi-instance subprocesses.



changed. Table 16 shows the three patterns identified for transforming a multi-instance task into a multi-instance subprocess that can get processed by above discussed patterns.

8.3 process data handling

Sections 8.1 and 8.2 discuss means to identify enablement of activities with respect to the existence of required data objects and to handle the termination of activities by storing corresponding data objects in a database – both including the consideration of complex data dependencies. Enabled activities may be started; upon start of activity execution, the data objects proofed existing need to be gathered from the database. Upon activity termination, the content of the data objects changed during process execution need to be stored in the database. During execution, the user must be able to read and write information from and to the data objects respectively. This leads to the following process data requirements (PDR) to handle the process execution runtime information in terms of data objects.

(PDR-1) Data object retrieval from the database.

(PDR-2) Data object storage in the database.

(PDR-3) Generation of forms from retrieved data objects to present their contents to the user and to retrieve the user's changes for storage.

(PDR-4) Specification of the data nodes' attributes (cf. Definition 4.2 on page 62) in the process model as input to the form generation in PDR-3. These attributes determine the information to be shown and changed by the user.

In the remainder of this section, we introduce means to tackle requirements PDR-1 to PDR-3 and provide. For PDR-4, we provide an example visualization but consider this requirement a tooling challenge. Thus, PDR-4 is out of scope in this section and we omit a final solution. Discussing concept details, we start with PDR-1.

For data object retrieval, we adapt the SQL statements of Section 8.2 for data object reads (CRx and D*Rx) by replacing "`SELECT COUNT (id)` [...] $\geqslant$ 1" of the queries with "`SELECT *`". Considering Figure 97, activity *Correlate PO to CPs* reads data nodes of classes *CP* and *PO*. To retrieve, for example, the object represented by the data node of class *PO*, the SQL query `SELECT * FROM PO WHERE po_id = $ID AND state = 'preparing'` is used. This example query is adapted from pattern CR1 (see Section 8.2): `SELECT COUNT (po_id) FROM PO WHERE po_id = $ID AND state = 'preparing'` $\geqslant 1$.

These adapted read-queries as well as the unchanged create, update, and delete queries are the basis for process data handling. Following the formalization in Chapter 4, the data class of a data node determines the set of attributes which might get utilized throughout process execution; each data node contains a subset of these attributes indicating the attributes required for the current operation (read, write). These

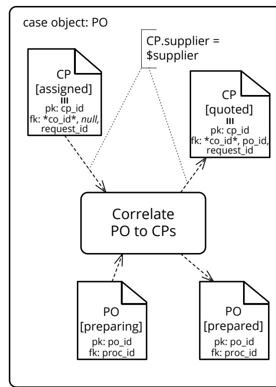


Figure 97: Activity *Correlate PO to CPs* reads and writes data nodes of classes *CP* and *PO* with *PO* being the case object of the corresponding scope instance and data node *CP* being of type multi-instance.

attributes of a data node are distinguished into key attributes (primary and foreign keys), the state attribute as well as further mandatory and optional attributes. While the key attributes and the state attribute are covered by the already introduced SQL queries, the remaining attributes holding actual content for the process execution are not considered in these queries. Therefore, the SQL queries need to be adapted in this direction.

Since mandatory as well as optional attributes must be stored in the corresponding database table, we handle both equally and refer to the union of these attribute sets as *data node attributes*. In Figure 97, each object of class *CP* that is read by activity *Correlate PO to CPs* requires information about the supplier. Thus, the data node attribute is marked mandatory for the corresponding data node in the process model. Visualization of this fact (see PDR-4) is omitted in the scope of this chapter since it reduces model readability.

However, one option for visualization would be to annotate each data node with the corresponding data node attributes as shown exemplarily in Figure 98; a "+" symbolizes a mandatory attribute and a "-" symbolizes an optional attribute. Both examples are an extended version from the examples in Figure 91. In Figure 98a, the input data node does not require any data node attribute while the output data node requires attribute *supplier* to be set (mandatorily). In Figure 98b, both, the input and the output data node of type *CP* requires the attribute *supplier* to be set. For this low number of data nodes and attributes, this option might be feasible; however, considering the build-to-order and delivery process in Section 2.4 shows this option's weakness, i. e., visual complexity. Alternatively, the attributes can be managed in a property editor that is not visualized in the process model. Summarized, visualization remains a tooling challenge and is not part of the discussed concept. A formal representation of attributes is exemplified in Listings 2 and 4 on page 169 and page 170 using XML.

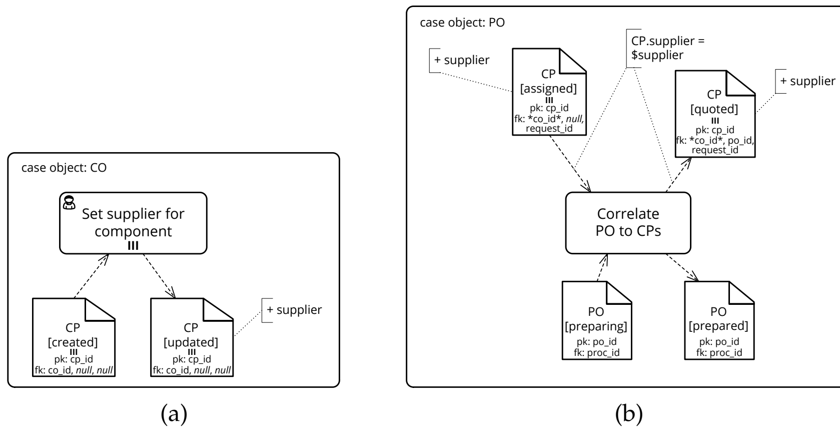


Figure 98: Examples from Figure 91 extended with explicit attribute annotation to data nodes. A "+" symbolizes a mandatory attribute and a "-" symbolizes an optional attribute. Possible attributes are determined by the data model given in Figure 87 on page 206.

Reading the data node attribute values is covered by above adaptation trough `SELECT *` by reading all attributes the corresponding data object might utilize independently from data node specification. This can be optimized by considering the data node attribute information given for each data node. Then, in the SQL query, the $*$ is replaced by a comma separated list of key attributes, the data state attribute, and specified data node attributes. For case objects, only primary key attributes are specified. The SQL query to read the object represented by data node *PO* in Figure 98b is `SELECT (po_id, state) FROM PO WHERE po_id = $ID AND state = 'preparing'`. The SQL query to read the object represented by data node *CP* is `SELECT (cp_id, co_id, po_id, request_id, state, supplier) FROM JOINALL (CO, PO, ProC) WHERE ProC.proc_id = $PID AND CP.state = 'assigned' AND CP.po_id IS NULL AND CP.supplier = $supplier`[7]. For reading non m:n data nodes, the last condition *CP.supplier = $supplier* would be omitted. Both queries base on patterns CR1 and $D^{m:n}$R4 respectively.

Writing the data node attributes requires an adaptation of the insert and update statements (see PDR-2). In case of data object creation (*[new]* annotation in upper left corner of data node), the SQL queries in Section 8.2 contain already comma separated lists of key attributes and the state attributes. This list gets extended by the specified data node attributes. Assuming, in Figure 98b, *CP* would be a single-instance data node being created by activity *Correlate PO to CP*, the SQL query basing on pattern $D^{1:1}$C1 would be `INSERT INTO CP (cp_id, co_id, po_id, request_id, state, supplier) VALUES (DEFAULT, fk1, fk2, fk3, 'quoted', $supplier)` with *fk1*, *fk2*, and *fk3* resolving to SQL queries as given in the pattern description and *$supplier* resolving to the value pro-

7 *JOINALL(D2, D3, D4, D1) = (((D2 INNER JOIN D3 USING d3_id) INNER JOIN D4 USING d4_id) INNER JOIN D1 USING d1_id)*

vided for data attribute *supplier*. For instance, the query for *fk1* is `SELECT CO.co_id FROM JOINALL(CP, CO, ProC) WHERE ProC.proc_id = $ID`.

Secondly, consider activity *Receive quote* in Figure 99 from the build-to-order and delivery process (cf. Figure 9 in Section 2.4; again discussed in Section 8.4. All objects being output to the given receive task shall be created represented through the *[new]* annotation in the upper left corner of all data nodes. The corresponding, complete query for the object of class *Quote* is `INSERT INTO Quote (quote_id, request_id, state) VALUES (DEFAULT, $ID, received, 21, 30)`, since no additional attributes are specified in the data model in Figure 87 on page 206.

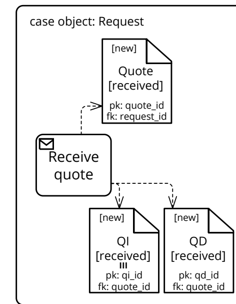


Figure 99: Task *Receive quote* from the *Request quotes* subprocess in the build-to-order process.

Please note, the order of storing the data objects into the local database is important since, for instance, one object may relate to another object via foreign key relationship. In this case, the second object must have been stored first to ensure that the key value is known to be added for the first object. In our second example, an object of class quote details *QD* has a foreign key relationship to an object of class *Quote* such that it must be inserted after the object of class *Quote*.

We assume that the foreign key relationships between the output data nodes (objects) of a receive task form a directed acyclic graph over the respective data classes. It implies that these relations have a partial order and that it is possible to insert referenced data objects before the ones that reference them. Then, this directed acyclic graph describes the insertion order from leaf to root node so that first the *Quote* object and then the quote details *QD* and the quote item *QI* objects are inserted. When the graph is completely traversed, the receive task has finally reached the *terminated* state (cf. activity life cycle in Figure 30 on page 74).

Updating an object as given in the examples in Figure 98 requires – analogously to the create statements – an extension of the attribute list for the *SET* part of the query by the specified data node attributes. Considering Figure 98a, after resolving the multi-instance property by following pattern MI2: update, the update statement bases on pattern $D^{1:1}$U2 because of the given input data node. The adapted SQL query is `UPDATE CP SET state = 'updated', supplier = $supplier WHERE CP.co_id = (SELECT CO.co_id FROM JOINALL(CP, CO, ProC) WHERE ProC.proc_id = $ID) AND state = 'created'` under the assumption that *$supplier* resolves to the value provided for data attribute *supplier*.

Finally, delete statements are not adapted, since the given SQL queries already delete the complete object including all data node attributes.

During process execution, upon start of activity execution, the data retrieved from the database tables can be used to generate the forms to provide this information to the user and to gather information required for activity termination and thus, data storage (see PDR-3). Each element of the form is linked to a specific attribute in the data model such that form generation and data storage is clear without ambiguity. Figure 100 shows an example how such generated form could look like for activity *Set supplier for component* after application of pattern MI2: update to resolve the multi-instance property.

(a) Upon activity start.

(b) Before activity termination.

Figure 100: Form generation for task *Set supplier for component* based on actual process data.

Figure 100a shows the form directly after start of the activity with all retrieved information; key attributes and the current state are read-only, data node attributes referred to as *content attributes* are changeable. Information collected by the user is required to be entered in the form. At termination, all entered information is stored in the corresponding database tables. In the shown example, the supplier is the only information to be entered during process execution; here, the value is set to *B*. Additionally, the output data state must be determined. We assume that the state is set manually by the process participant by selecting the corresponding radio button. In the example, only one option exists: data state *updated*. Otherwise, multiple options would have been shown in the bottom of the state attributes field. In contrast to this manual data state handling, functions could be used to automatically determine the business state based on the input information given by the user. The corresponding algorithms are out of scope for this thesis.

8.4 AUTOMATING DATA EXCHANGE IN CHOREOGRAPHIES

In daily business, organizations generally do not act purely individually as assumed in above sections. Instead, organizations interact with each other, e. g., concluding contracts or exchanging information. Aligned with the build-to-order and delivery process from Section 2.4, Figure 101 describes the interaction between the *Computer retailer* and a *Supplier* with respect to a request for a quote (first part of Figure 35). The computer retailer sends the *Request* to a chosen supplier which internally processes it and sends the resulting *Quote* as response which then is handled internally by the computer retailer. An interaction between business processes of multiple organizations via message exchange is called *process choreography* [370]. Choreography modeling usually utilizes concepts to specify the message exchange (order, participants, type (send, receive)) and to correlate this to the process' control flow.

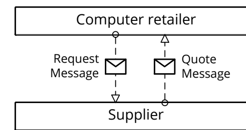


Figure 101: Request for quote choreography.

For instance, the industry standard BPMN [243], one of the few process description languages explicitly supporting process choreography modeling, provides the following concepts to model process choreographies. A *choreography diagram* describes the order of *message exchanges* between multiple participants from a global view, called *global choreography model* (cf. Figure 102). In the given example, only one message exchange takes place such that only one choreography task is shown. The message exchanges are then refined into *send* and *receive activities* distributed over the different participants. This can be captured in *collaboration diagrams* connected by message flows describing how each participant's *public* process interacts with other participants [342], also called *global collaboration diagram*. Figure 104 (left) on page 254 shows an example for global collaboration diagram. Finally, each participant refines the own local view on the choreography through private process models that include representation of communication to other participants, also called *local choreography model*. Figure 104 (right) shows a corresponding example fitting to the given global collaboration diagram.

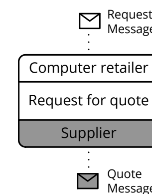


Figure 102: Global choreography model for above request for quote choreography.

Problem context

The problem of implementing local choreography models that adhere to a global agreement can be approached in two ways: top-down or bottom-up. Following the top-down methodology, all participants jointly agree on a global data exchange and collaboration model to

which each participant's local process and data models either must adhere or are required to be changed accordingly [350]. Conversely, if local process or data models are the starting point and *not* to be changed, often, a mediator is required to realize the collaboration. Such mediator can be implemented through the Enterprise Integration Patterns [140] or orchestration services [251]. In this section, we follow the top-down methodology. As valid in Sections 8.1, 8.2, and 8.3, we utilize the activity-centric process modeling paradigm and utilize BPMN as representative during concept discussion.

Deriving a local choreography from a global one is a non-trivial step; various techniques are required [69] including *locally enforcing* the order of globally specified message exchanges. In general, both control flow (order of message exchange) and data flow (actual message contents) need to be addressed when transitioning from global to local models.

Typically, choreography models are used to globally agree on a contract about the messages exchanged and their order. In above example, both participants agreed that first the customer may send a request to the supplier which is then answered with a quote by the supplier. Based on the agreement, each participant has to implement its public process as a *private* process describing the executable part of this participant including the interactions with other participants as described in the choreography; this private process is called a *process orchestration* [201]. Thus, the concepts introduced in Sections 8.1 to 8.3 can be used for local data execution. Existing approaches for deriving a process orchestration for each participant from a choreography, such as the *Public-to-Private* approach [342], only cover the control flow perspective of the contract: ensuring the correct *order* of messages. In the following, we address the correct *contents* of messages to achieve a correct *data exchange* that realizes the choreography.

Challenges

Challenges

Generally, organizations store their data in local databases that other choreography participants cannot access. These databases follow local data schemes which differ among the organizations. However, the interacting organizations want to exchange data and therefore have to provide the information to be sent in a format which is understood at the receiving side. Thus, an agreed exchange message format has to be part of the global contract mentioned above. For a successful process choreography, it has to be ensured that messages to be sent are provided correctly and that received messages are processed correctly *based on the global contract*. In more detail, three challenges for collaboration (CC) with respect to data flow arise.

(CC-1—Data heterogeneity) Interacting participants, such as our computer retailer and supplier, each implement their own data schema for handling their private data. For sending a message to another participant, this local data has to be transformed into a message the recipient

can understand. In turn, the received message has to be transformed into the local data schema to allow storing and processing by the recipient; i. e., CC-1 is about mapping between global and local data models.

(CC-2a—Correlation) A participant may run *multiple instances* simultaneously. A message sent to a participant is typically intended for a particular process instance and must only be received by that instance. Assigning a message to the intended process instance is called correlation and may happen through dedicated *correlation identifiers* stored in the message. The challenge here is to populate correlation identifiers correctly and to correctly match a message to the right process instance.

(CC-2b—1:n communication) In addition to one participant running multiple instances of its process, a single instance of that process may need to interact with *multiple* process instances of another participant at the same time. For example, a computer retailer may send multiple requests for a quote to multiple suppliers and receives multiple corresponding answers (see Figure 9). The challenge here is to produce multiple messages for different participants and to process multiple incoming messages from different participants.

Although CC-2a and CC-2b are closely related, they require distinct solutions. A correlation mechanism ensures that one message arrives at its intended receiver. Here, 1:n communication adds another dimension to the problem as it requires to consistently handle a *set* of correlation identifiers and to process *sets* of messages and *all* their contents.

Current choreography modeling languages such as BPMN do not provide concepts to solve CC-1, CC-2a, and CC-2b. Instead, each participant manually implements message creation and processing for their private process, which is error-prone, hard to maintain, and easily results in incompatibilities to other participants in the choreography.

Solution

Proposed solution

We combine several existing approaches and previously described concepts to automate data exchange in process choreographies entirely model-driven as follows:

1. All participants agree on a global collaboration diagram as defined in Definition 4.19 on page 82. In this chapter, we will express a global collaboration diagram in BPMN as representative for the activity-centric process description languages; BPMN will also be used for the private process orchestration models.
2. In addition, we introduce that all participants globally agree to specific data exchange formats used in the collaboration modeled in UML [244] and referred to as global data model.
3. To map the control flow of a global collaboration diagram into local ones, we utilize the Public-to-Private approach [342] unchanged.
4. We use a straight-forward attribute-level data mapping between global and local data models to address challenge CC-1.

5. We utilize correlation identifiers that are specified as part of the data exchange format and naturally translate to locally usable correlation keys by the above mentioned data mapping to address challenge CC-2a. The concept of correlation identifiers is borrowed from the BPMN standard.
6. To process (create and store) messages in a model-driven fashion, we apply concepts from Sections 8.1, 8.2, and 8.3 for automatically deriving SQL queries from data nodes to enact complex data-dependencies and to handle process data.
7. We utilize the notion of dedicated (multi-instance) case objects for subprocesses (cf. Section 8.1) to realize 1:n communication with a set of participants to address challenge CC-2b.

Summarized, our combination of model-driven data flow, model-driven data transformation (from global to local respectively from data to message and vice versa), choreography-orchestration mapping, and key-based correlation principles results in two actual contributions: (i) few additions to choreography modeling in terms of model-driven correlation identifiers and – as main contribution – (ii) a methodology and operational semantics that connects existing research works and makes them executable following the top-down approach finally resulting in automatically executable message handling based on model-information only. Thereby, the operational semantics translates model-features into executable code for data retrieval, storage, and transformation following the goal of platform-independence. Specifically, we utilize SQL and XQuery [380] as thoroughly discussed in Section 8.6.

Choreography Execution Requirements

Requirements

The challenges CC-1, CC-2a, and CC-2b give rise to specific requirements for automating data exchange in process choreography modeling and execution. We discuss these choreography execution requirements (CER) and their possible realization in the following.

(CER-1—Content of message) Messages contain data of different types to be exchanged. The involved participants have to commonly agree on the types of data and their format they want to exchange.

(CER-2—Local storage) The participants create and process data used for communication with other participants in their private processes. This needs to be stored and made available in their local databases.

(CER-3—Message provision) As the data provided in a message is local to the sender, the data must be adapted to the agreed format such that the recipient can interpret the message content.

(CER-4—Message routing) Multiple parties may wait for a message at a certain point in time. This requires to route the message to the correct recipient.

(CER-5—Message correlation) After being received by a participant, the message needs to be correlated to the activity instance which is capable to process the message content.

(CER-6—Message processing) Activities receiving messages have to extract data from the message and to transform it into the local data format usable within their processes.

Requirements CER-1, CER-2, CER-3, and CER-6 are basic features to realize CC-1; CER-4 and CER-5 originate in CC-2b; and CER-5 also addresses CC-2a.

Languages such as Web Services Description Language (WSDL) [377] use data modeling to specify message formats; we adopt these ideas to address CER-1. Requirements CER-2, CER-3, and CER-6 concern the processing of data in an orchestration. The concepts in Sections 8.1 to 8.3 allow to model and enact data dependencies as well as to handle process data through create, read, update, and delete operations on multiple data objects – even in case of complex object relationships. For this, annotations on data nodes (objects) are automatically transformed into SQL queries (CER-2). Further, data querying languages such as XQuery [380] allow to implement data transformations between a message and a local data model. In the following, we combine these approaches to specify message extraction (CER-3) and message storage (CER-6) in a purely model-based fashion. Process description languages such as business process execution language (BPEL) [240] and BPMN correlate a message to a process instance based on key attributes in the message; we adopt this idea to address CER-5. Below, we describe how to tackle these requirements to automate data exchange in process choreographies.

Requirement CER-4, the actual transmission of messages from sender to receiver, is abstracted from in choreography and process models and also not discussed in this thesis. One can use standard technologies such as middleware or web services to realize the communication between the process engines of participants.

Modeling Guideline

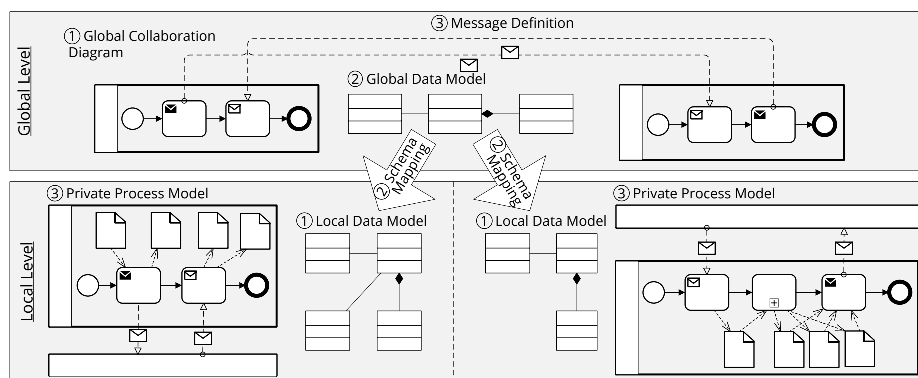


Figure 103: Modeling guideline.

Figure 103 illustrates our modeling guideline as first step towards the implementation of automatic data exchange of a process choreog-

raphy in an entirely model-based approach. The operational semantics follows on page 258. The modeling guideline consists of two levels: the *global level* (top), where the public contract is defined, and the *local level* (bottom), where the local process implementations can be found. Next, we describe the details of the global contract followed by the local level both along our modeling guideline.

Global level

We assume that the choreography partners have already specified a global collaboration diagram that shows how each participant's public process interacts with the other participants and ensures local enforceability of control flow [342]; see Figure 103 (top). To support data exchange between participants, we propose that this public contract is supplemented with a global data model in which the partners specify the business objects to be exchanged in terms of data classes; see Figure 103 (top middle). On the global level, all choreography parties together define the following artifacts:

Global collaboration diagram. The global collaboration diagram describes the control flow layer of the choreography, i. e., it describes which messages are exchanged in which order on a conceptual level. Exemplary, the left part of Figure 104 shows the global collaboration diagram of the *Request for quote* choreography sketched above[8]. It includes public processes with all necessary send and receive tasks for each participant, the computer retailer and the supplier. Note that we – following the process model definition in Definition 4.10 – restrict the modeler to use the send and receive tasks for message exchange modeling although BPMN also allows the utilization of events to equivalently model message exchange. Such events can only receive and send data but are not supposed to process the data, for instance, in terms of transformation or correlation. Since we do process data before message sending or upon/after message retrieval, utilization of send and receive tasks ensures compliance to the BPMN standard after application of the new concepts.

Global data model. Messages are used to exchange data. In choreography modeling languages such as WS-CDL [379] or BPEL4Chor [70], the data carried by a message is described technically by attribute names and data types for each message individually [377]. Instead, we propose that the interacting parties first agree on data classes whose objects they want to share and document this in a message data model, for instance, using UML class diagrams [244] or XSD [380], globally agreed on from all participants. In the following, we refer to this as *global data model*. In our example, computer retailer and supplier have agreed on three data classes, *Global_Request*, *Global_Quote*, and *Global_Articles*, for their collaboration as shown in the upper part of Figure 106. Each object of these classes has a unique identifier attribute (e. g., *r_id* for

8 Note that we changed the key identifiers for *Request* and *Quote* data classes from *request_id* and *quote_id* to *r_id* and *q_id* respectively with respect to the data model given previously in this chapter to increase readability of the figures in this section

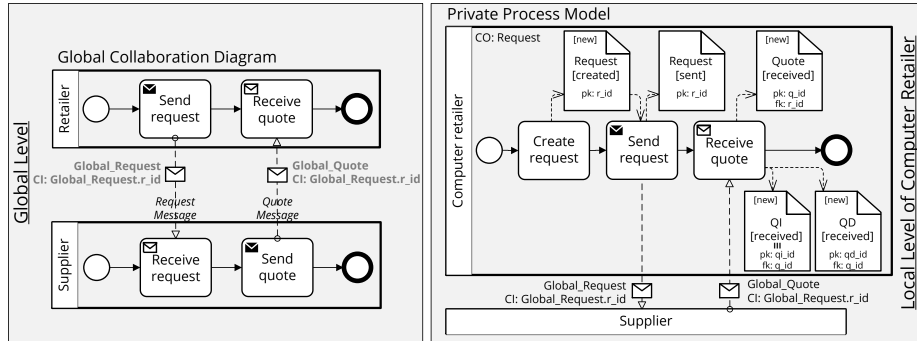


Figure 104: Global collaboration diagram (left) and local process model of the *Computer retailer* (right).

Global_Request) and some have a foreign key attribute (e. g., *r_id* for *Global_Quote* referencing a *Global_Request*) to express relationships.

Message definition. Then, message types are specified by *referring* to business objects defined in the global data model. As a technical assumption, each message carries exactly one global data object; this object can be hierarchical allowing sending multiple objects as data object collection within one message. Further, we adopt key-based correlation [240, 243] for messages: each message contains a set of key/value pairs that allow identifying the correct process instance on the receiver side; each key is an attribute of some data class in the global data model. For example, *Request Message* in Figure 104 (left) refers to an object of class *Global_Request* and *Quote Message* refers to an object of class *Global_Quote* which in turn may have multiple objects of class *Global_Article* (data object collection); a *Quote Message* contains a *Global_Quote* object and all its *Global_Article* objects. Both messages use attribute *r_id* of class *Global_Request* as correlation key. Altogether, a message is declared as tuple $msg = (name, CI, c)$, where $name$ is the message type, the correlation information CI is a set of fully qualified attributes of the scheme $class.attribute$ represented as string, and c is the class of the actual data object in the message. Figure 105 shows the graphical representation of the message class in Unified Modeling Language (UML). Considering BPMN, the standard must be extended. Originally, a BPMN message contains a string identifying its name, i. e., message type. We add correlation information as a list of strings, each denoting one fully qualified attribute of the global data model, and the payload as class of the global data model referring to the exchanged data object.

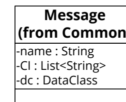


Figure 105: Message class.

Local level

Having specified the global collaboration diagram, the global data model, and the message definitions, each participant has to locally realize the collaboration by specifying the local data model, the schema

mapping, and the executable private (local) process model. For this, we extend the existing P2P approach [342] which already describes how to realize the control flow of the global collaboration diagram in a local process model for each participant. Our contribution is to include the data perspective in this step: each participant separately defines a local data model and a schema mapping between their local and the global data model and implements the private process conforming to their public process in the global collaboration diagram.

In the following, we first describe the ideal situation where the local process model does not exist yet and can be created from the global model [342] or is easily adapted to realize the global model. Moreover, we assume that the local data model can be created or can be mapped to the global data model. Other cases which limit our approach are discussed starting on page 257.

Next, we introduce the local artifacts one by one:

Local data model. Each participant defines a local data model which describes the classes of data objects handled by the private process. For example, the local data model of the *Computer retailer* has four classes *Request*, *Quote*, *Quote Details DQ*, and *Quote Item QI*; see Figure 106 (bottom). We propose to also use the local data model to design the schema for the database where the objects are stored and accessed during the process execution. There are some requirements to the local data model with regards to the global data model as described next.

Schema mapping. A schema mapping defines how attributes of local classes map to attributes of global classes and allows automatation of a data transformation between global objects contained in messages and local objects. We consider an attribute-to-attribute schema mapping which injectively maps each attribute of a global data class to an attribute of a local class as shown in Figure 106. Note that the attributes of class *Global_Quote* are distributed over classes *Quote* and *Quote Details QD*. The local implementation can hide private data in a local attribute by not mapping it to a global attribute (the mapping is not bijective), e. g., the *state* attributes of each local class and the suuplier attribute of the local *Request* class.

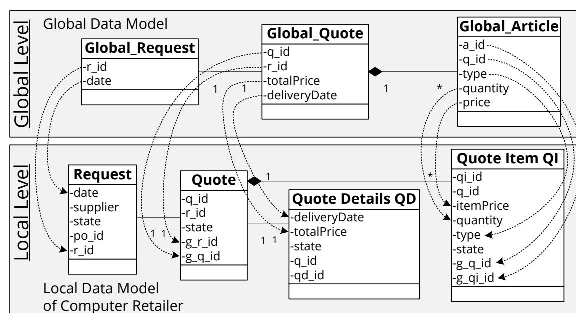


Figure 106: Schema mapping for *Computer retailer*.

By restricting ourselves to injective mappings, we exclude aggregating several local attributes into one global attribute, e. g., aggregating a list of review scores into a single aggregated score. While aggregation is simple for producing a single attribute from multiple ones, it is impossible to split one attribute into multiple individual attributes without domain knowledge; such splitting requires a domain-specific function increasing platform-dependency. As our goal is to keep code generation (SQL queries and data transformation through XQuery) platform-independent and as such encapsulated, we decided to move the aggregation outside the data mapping by pushing domain-specifics to the process logic controlled by the process owner. By enforcing an injective mapping, the process model must handle aggregation locally such that it can be used in global communication. In the above example, the review scores are first aggregated locally into an attribute of the local data model which can then be sent in a message.

Local data model and schema mapping must ensure that primary and foreign keys are managed locally to avoid data inconsistency: When a local object can be created from a received global object, key attributes of the global object must map to non-key attributes of the local objects. For example, the local *Quote* shall be created from a *Global_Quote* object, thus the class *Quote* gets the attributes *g_q_id* and *g_r_id* to allow storage of the primary key *q_id* and the foreign key *r_id* of a *Global_Quote* for local use. Typically, these keys are used for correlation.

Executable private process. Based on the global collaboration diagram, each participant designs their private process by enriching their public process with activities that are not publicly visible. In addition, each local process (and each subprocess) gets assigned a *case object*; a concept that we borrow from object-centric processes [237] that was already described in Sections 4.2 and 8.1. A case object represents the driving object – the actual processed main information object – for the (sub-) process and the other related information objects are associated directly or indirectly to the case object. Typically, it is easy to identify, if the data view is considered for process modeling, since it is usually closely related to the goal of the business process. Instantiating the process also creates a new instance of this case object that uses as primary key value the process instance id (see Section 8.1).

Figure 104 (right) shows the private process model of the customer. First, activity *Create request* creates and prepares a new instance of case object *Request* (see "CO" in the top left corner of the process). The schema mapping defines which local data objects are required to derive the payload and the correlation information for a message to be sent; this is included in the process model by associating the required data objects as input data nodes to the send task. In our example in Figure 104, activity *Send request* creates a *Request Message* containing a *Global_Request*. The respecting local *Request* is associated to *Send request* as input data node. Correspondingly, we associate the local data objects

Figure 107: Private process model and local data model of the *Supplier*.

into which the payload of a received message is transformed as output data nodes of a receive task. The last activity modeled in the computer manufacturer process receives the *Quote Message*. The payload of this message is transformed into data objects of classes *Quote*, *Quote Details QD*, and *Quote Item QI* (multi-instance) all being associated as output to the receive task. The process designer has to utilize the data annotations from Section 8.1 to specify whether the receive task creates new or updates existing data objects. In the given example, the message payload is used to create new objects only as indicated through identifier *[new]* in each data node. Local data schema, schema mapping, and private process *together* define the local choreography of the participant.

Figure 107 shows the private process model and the local data model of the second participant – the *Supplier*. Here, each attribute of a local class directly maps to a corresponding attribute with an equivalent name in the corresponding global class. For instance, attribute *price* of class *Global_Article* maps to attribute *articlePrice* of class *Article*, attribute *r_id* of class *Global_Request* maps to attribute *g_r_id* of class *Request*, and so on. The private process has three activities: After receiving the *Global_Request*, which is stored as *Request* object in state *received*, the supplier processes the request and creates the *Quote*. Sending the *Global_Quote* message requires data objects *Quote* and *Article* to set the payload and *Request* to set the correlation identifier *Global_Request.r_id*.

This modeling guideline proposes a logical order in which the artifacts should be created based on dependencies between them. However, situations may arise where a different order (or iterations) are required. In any case, by refining the public process into a private one and by defining local data model and schema mapping as described, a process modeler always obtains a local choreography that is consistent with the global contract (global collaboration diagram).

Limitations

Our approach assumes that the local process model and data model can be created as desired (e.g., they do not exist yet locally). This partly

relies on the assumption that by choosing a top-down approach, each participant has the freedom to realize its share of the globally agreed collaboration. In other cases, where a local process model already exists and is difficult to adapt or change, one either has to negotiate a more suitable global choreography or pursue a bottom-up approach.

However, even a newly created local process model is typically embedded in a larger organizational context that already defines a local data model for the entire organization and related processes. In the extreme case, the to-be-created local process model may not be compatible with the local data model requiring action in data model adaptation – on the local or global side –, application of a mediator, or a change of the overall approach from top-down to bottom-up. In the latter two cases, the results of this section cannot be utilized.

If the to-be-created local process model is not compatible to the other local process models with respect to, for instance, internal control flow or data flow, or the already existing one cannot be adapted accordingly, analogous actions can be taken. If adaptations on the local or global level can be performed, the collaboration may take place. Otherwise, either a mediator needs to be applied or the overall approach needs to be changed. If none of the discussed action alternatives is applicable with respect to local model specification, the collaboration simply cannot take part.

Executing Data-annotated Process Choreographies

Next, we show how to make the local choreography executable, thus achieving a correct implementation by design. Thus, we introduce the execution semantics to automatically generate as well as correlate messages and to persist them using the introduced modeling concepts. First, we start with an overview based on our example before we dive into details.

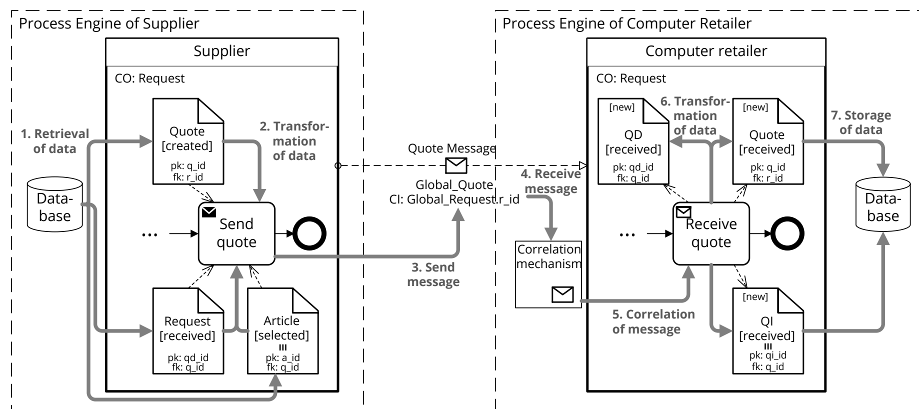


Figure 108: Approach overview.

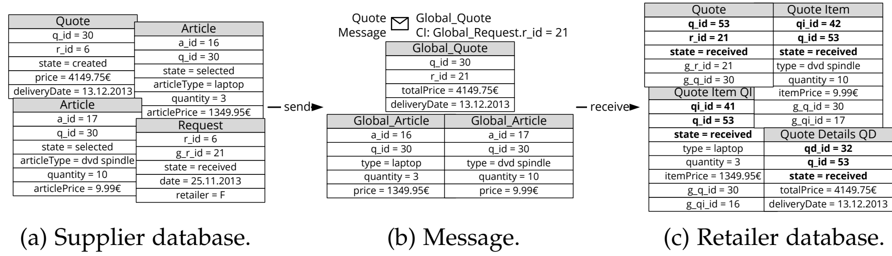
Execution overview As discussed by requirements CER-1 to CER-6, exchanging a message requires the following 7 steps that we also illustrate in Figure 108 for the

Supplier sending a quote to the *Computer retailer*: (1) The required data objects are retrieved from the supplier's database (satisfying CER-2) and (2) transformed to the message (satisfying CER-1 & CER-3), which is (3) sent from the supplier and (4) received at the computer retailer's side (satisfying CER-4). The received message is then (5) correlated to the corresponding activity instance (satisfying CER-5), where the message (6) gets transformed into data objects (satisfying CER-1 & CER-6) which are then (7) stored in the computer retailer's database (satisfying CER-2 again).

The send task labeled *Send quote* creates and sends the message. Generally, the input data nodes specify the data objects and their states required to start activity execution. In the context of sending a message, the input data nodes to send tasks additionally describe the local data required to create the message to be sent. Therefore, in step 1, they are retrieved from the local database before step 2 transforms this information into the corresponding message based on the given schema mapping. Here, objects of classes *Quote* and *Article* (as data collection) are utilized to create the message's payload *Global_Quote*. *Global_Quote* represents a hierarchical object consisting of a number of *Global_Article*s (see global data model in Figure 106). Further, the specified correlation identifier *Global_Request.r_id* is added to the message based on the input data object *Request*. This is needed by the computer retailer to correlate the message to its correct scope instance (process or sub-process instance). After preparing the message, the actual sending to the recipient is executed by the send task (step 3). The execution of the send and the message reception by the correct recipient is done by inter-engine communication in lower layers, e. g., by web services or middleware, which is not discussed in this chapter.

Analogously, the retrieval of the message at the recipients side, here the computer retailer, is managed by the same underlying layer (step 4). Next, the message needs to be correlated to the corresponding scope instance in the correct process model (step 5), where it is assigned to the correct instance of the respective receive task. The correlation is done with the correlation key of the message which is the *Global_Request.r_id* in the example. In our example, the response of the supplier is handled in the *Receive quote* activity. Analogously to the input data nodes, the output data ndoes of an activity specify the data objects and their states expected to be created and stored after activity execution. Therefore, step 6 transforms the received message's payload into the specified data objects following the given schema mapping. Here, data objects of classes *Quote*, *Quote Details QD* and *Quote Item QI* (as data collection) are derived from the message's payload *Global_Quote*. Finally, execution of the receive task stores the derived data objects in the customer's local database (step 7).

Next, we present in more detail the operational semantics for the modeling concepts that automatically enact these steps from the local



(a) Supplier database. (b) Message. (c) Retailer database.

Figure 109: Representation of one instance from the message flow shown in Figure 108 where each presented object refers to one column in the corresponding database table named as the respecting class of the utilized data nodes (objects).

process model only. We first consider sending and receiving messages respectively followed by correlation handling.

Send message

The preparation of a message to be sent consists of the retrieval of required data objects from the database (step 1) and their transformation accordingly to a given schema mapping (step 2). During execution, each activity appears in various activity states depending on the current status of execution (see Definition 4.12). According to [370] and [243], an activity may be, among others, in states *initialized*, *enabled*, *running*, *completing*[9], and *terminated*, in this order. Initially, since initialization of the respective process instance, the send task *Send quote* is in state *initialized* and waits for its enablement. With arrival of the control flow, the send task changes into the *enabled* state. There, data dependencies, i. e., the availability of the specified input data objects, are checked.

In our example, the data objects *Request* and *Quote* as well as all *Article*s have to be available in the states annotated to the corresponding data nodes to activate the *Send quote* activity. Those data objects are necessary to provide the payload and correlation identifier of the message to be sent. The availability can be checked through an automatically generated SQL query (see Section 8.1). For example, activity *Send quote* requires data object *Quote* in state *created* with primary key *q_id* and foreign key *r_id* pointing to case object *Request*; the corresponding guard is `(SELECT COUNT (q_id) FROM Quote WHERE r_id = $ID AND state = created)` $\geqslant$ `1`. The SQL query returns the number of *Quote* entries in the local database that are in state 'created' and related via foreign key *r_id* to the case object instance *Request* of the current process instance (identified by `$ID`); there has to be at least one *Quote* entry for start of execution.

In Figure 109a, an extract of the supplier database[10] is illustrated with each table representing one entry in the *Request*, *Quote*, and *Article*

9 Compared to the activity life cycle introduced in Chapter 4 (see Figure 30 on page 74), we added state *completing* to explicitly show when data objects are persisted in the course of activity execution.

10 for each data class of the data model, there exists one corresponding table in the database

database tables respectively. Assuming that the currently running process instance has the identifier `$ID = 6`, then executing the above SQL query returns 1, i. e., the required *Quote* object is available. Availability is checked in this way for all input data nodes (objects).

If all data dependencies are fulfilled, the send task retrieves the data objects from the local database (step 1) and transforms them into the actual message (step 2). For retrieval, we utilize the SQL queries from Section 8.3. For example, object *Quote* is retrieved by statement `SELECT * FROM Quote WHERE r_id = $ID AND state = created`. All specified data nodes (objects) are retrieved analogously and then transformed into the global representation following the given schema mapping. For instance, object *Quote* is transformed into object *Global_Quote*. In our example, we utilize the schema mapping shown in Figure 106. Since object *Global_Quote* is a hierarchical data object, also all related *Article* objects are transformed into corresponding *Global_Article*s. As shown in Figure 109a, there are two *Article* objects indicated by the foreign key `q_id = 30` matching the corresponding primary keys of the *Quote* object referring to the process instance with id 6. After transformation, all three global objects are added to the payload of the message to be sent by the corresponding send task. The correlation information *Global_Request.r_id* = 21 is taken from attribute *g_r_id* of the local object *Request* as specified in the schema mapping as well. After completing the message creation and adding the correlation identifier, the state of the send task changes from *enabled* to *running* indicating processing of the activity. The work now performed by a send task is to initiate the actual sending of the prepared message shown in Figure 109b.

Receive message

After a received message has been correlated to the corresponding instance (see below) it can be processed by basically reversing the two steps for sending a message. First, the objects in the message are transformed into the local data model (step 6 in Figure 108) followed by storing them in the local database (step 7). A receive task can only receive a message when it is in state *running*; when it received the message it changes to state *completing*. In this activity state, the transformation and storage steps take place.

The transformation, again, follows the given schema mapping. In our example, the received message of Figure 109b is transformed through the schema mapping shown in Figure 106: *Global_Quote* and its contained *Global_Article*s are mapped to a *Quote*, *Quote Details QD*, and multiple *Quote Items QI* (as data collection) as shown in Figure 109c. Note that attributes in bold are private attributes that are not defined by the schema mapping (consider them empty for now). For instance, the local object *Quote* gets attributes *g_r_id* = 30 and *g_q_id* = 21 while attributes *state*, *r_id*, and *q_id* (in bold) are set locally in the last step after *all* objects have been transformed.

The last step, i. e., step 7, persists the transformed data objects in the local database based on the annotated output data nodes of the receive

task. New information received by the message overwrites probably existing one. Setting the missing information (primary and foreign keys as well as states referred to as private attributes above) and storing the data objects in the local database utilizes the concepts from Section 8.1. We differentiate between creating a new table entry (*INSERT*) and updating an existing one (*UPDATE*) using the SQL queries introduced in Section 8.3 for process data handling. We also allow combinations of inserts and updates for one receive task, if the limitation for created data objects, i. e., insertion order[11], is considered.

Updates of data collections need to be handled specially. The patterns from Section 8.2 for updating a collection assume that all collection items are updated with the same values except for the primary key. However, when receiving a message, each collection item of a collection should be updated individually. Thus, we apply the 1:1 update pattern for single objects being dependent from the case object for each object of the collection (see pattern $D^{1:1}U1$). To select the correct object for the update from the local database, the statement needs to be extended by checking the global identifier in the WHERE-clause. It is assumed that the identifier for the data collection exists in the global data model enabling a distinct identification of each object. Assuming the quote items *QI* are updated and not created in the given example (see Figure 109), the quote item with the global identifier *g_qi_id* = 16 is updated by query `UPDATE QI SET state = received, type = laptop, quantity = 3 ,... WHERE r_id = $ID AND g_qi_id = 16`.

Message correlation

Before a message can be handled, it has to be assigned to its *receiving instance* which is also known as *correlation handling*. Note that correlation assumes that the the message is already routed to the correct receive task (see requirements CER-4 regarding message routing which is out of scope for this section). The standard approach for correlation handling is *key-based* correlation [240, 243], where some attributes of the message are designated as *correlation identifiers* (CIs). We first informally describe how key-based correlation works out in our approach and then present the formal definitions.

Basic idea. A process *instance* sending or receiving a message with CIs handles *local correlation keys* (CKs) which are basically local copies of the CIs. Four principles govern the handling of CKs: (1) all CKs of a new process instance are *uninitialized*; (2) the values of all CIs in a message (sent or received) must *match* the values of the corresponding CKs of the sending/receiving process instance; (3) if a CK is still uninitialized when sending/receiving a message, then the CK is *initialized* to the value of the corresponding CI; and (4) a CK must not be changed after initialization. Thus, when two messages carry the same CI, then the first message sent/received initializes the CK, and the second message

11 The order of storing the data objects into the local database is important since, for instance, one object may relate to another object via foreign key relationship (see Section 8.3).

can only be sent/received if its CI matches that CK. The underlying assumption is that any two process instances are distinct on their correlation values; if there is no process instance with matching correlation keys, then a new process instance is created.

In our approach, each CI is an attribute of the global data model and the global-local data mapping relates each CI to an attribute of the local data model that becomes the CK. In this sense, our approach refines key-based correlation where CKs are not defined as dedicated attributes, but are part of the local data model. For example, message *Global_Request* of Figure 107 has the CI *Global_Request.r_id*. By the schema mapping of Figure 104, the sending instance of process *Computer retailer* handles the CK *Request.r_id* in its local data model; the receiving instance of process *Supplier* also handles a CK *Request.r_id* in its own local data model (see Figure 107).

When receiving a message, the global-local data mapping is used to store the message contents locally – the values of each CI in the message is written (or compared) to a local attribute which is the CK. This allows to automatically initialize and match CKs. In the example of Figure 107, assume that the *Global_Request* message sent by *Computer retailer* has the CI *Global_Request.r_id* $= 21$. Receiving this message creates a new *Supplier* instance which writes the local *Request* object with attribute *r_id* and the same value 21. This way, the CK *Request.r_id* $= 21$ is automatically initialized. When the *Supplier* sends a *Global_Quote* message, it sets the CI *Global_Request.r_id* of this message to the value that matches the corresponding CK *Request.r_id* $= 21$ (according to the data mapping). Finally, the *Global_Quote* message can only be received by the *Computer retailer* process instance where the CK has been initialized to *Request.r_id* $= 21$. By the data mapping, this means the receiving *Computer retailer* instance has to be associated to a local *Request* object with attribute *r_id* $= 21$.

Next, we show how the information in the process model suffices to generate a query that extracts the value of a local CK for a given CI from the local data model. This query is used both for (1) *matching* an incoming message to the correct process instance and (2) *setting* CIs of message and additionally (3) for *initializing* CKs of process instances.

Matching initialized keys. Formally, the correlation information of a message $msg = (name, CI, c)$ is a set $CI = \{(k_1, v_1), \ldots, (k_n, v_n)\}$ of key/value pairs, where each key $k_i = c_i.j_i$ is an attribute $j \in \mathcal{J}_i$ of a global data class c_i. The correlation identifiers may be defined by global data classes which are not in the payload of msg, i. e., which are neither the specified data class nor a subclass of it. For example, the message of Figure 109b has the CI *Global_Request.r_id* while its payload is of class *Global_Quote* (as specified in Figure 104).

A participant's schema mapping θ maps each key to a local attribute. Each process instance *$ID* has its own case object instance and related object instances; message msg correlates to *$ID* when the value of each $c_i.j_i \in CI$ matches the value of the corresponding $\theta(c_i.j_i) = c_i'.j_i'$ of

some data object of class c'_i related to instance *$ID*. For example, the *Computer retailer* maps *Global_Request.r_id* to *Request.r_id* (see Figure 106). Thus, the message of Figure 109b can be correlated to a process instance where the case object has *Request.r_id* $= 21$.

The *value* of the correlation attribute $c'_i.j'_i$ can be extracted with respect to the case object $case$ of the receiving instance *$ID* as follows. As already stated, the case object $case$ is the driving object of a process; $case$ relates via its primary key to its process instance and all other data objects of the instance are dependent from it. An object of class c'_i relates to $case$ via foreign key relations. Thus, we can build an SQL query joining the tables that store c'_i and $case$, select only the entries where the primary key of $case$ equals *$ID*, and finally extract the value of attribute $c'_i.j'_i$; see Section 8.2 for details. Let $\alpha(c'_i.j'_i, case, \$ID)$ denote the results of this query. By ensuring that in the local data model the relations from $case$ to c'_i are only 1:1, the extracted $\alpha(c'_i.j'_i, c, \$ID) = v$ is uniquely defined.

With the above definitions, msg *correlates* to an instance *$ID* of a process with case object $case$ if and only if for each $(c_i.j_i, v_i) \in CI$ holds $\alpha(\theta(c_i.j_i), c, \$ID) = v_i$. This definition can be refined to not only consider the case object of the entire process, but also the case object and instance id of the scope that encloses the *running* task that can receive msg.

Setting correlation identifiers in messages. When sending a message msg, then each local CK that is mapped to a CI of the message has to have a valid value – otherwise the CIs in the message could not be set from the local process data. The CIs in the message can then be set automatically to *match* the local CKs by extracting for each CI $c_i.j_i$ of the message the corresponding value $\alpha(\theta(c_i.j_i), case, \$ID) = v_i$ from the sender's local data. Technically, this can be done in the same way as extracting the payload of msg as described above for sending a message.

Initializing correlation keys. From this point on, all process instances receiving a message with correlation key $c_i.j_i$ have to agree on the value v_i. The only exception is when v_i is still *undefined* at the receiving instance, i.e., $\alpha(\theta(c_i.j_i), case, \$ID) = \perp$. By initializing the local attribute $\theta(c_i.j_i)$ to value v_i, we can make *$ID* a matching instance for msg. Thus, we generalize the above condition: msg *correlates* to an instance *$ID* of a process with case object $case$ if and only if for each $(c_i.j_i, v_i) \in CI$ holds that if $\alpha(\theta(c_i.j_i), case, \$ID) \neq \perp$ then $\alpha(\theta(c_i.j_i), case, \$ID) = v_i$. When receiving msg, the local key attribute $\theta(c_i.j_i)$ can be initialized for *$ID* to value v_i. By ensuring that any correlation key $c_i.j_i$ that needs to be initialized is an attribute of the payload data class c in msg, receiving msg and transforming the object referring to c to the local data model will automatically write $\theta(c_i.j_i)$ to the local data store as discussed above for receiving a message (in particular, step 7 in the execution overview in Figure 108).

By persisting CKs as data attributes of the local data model, we rely on two assumptions: local CKs that are used when sending a message must have a valid value when sending this message and local CKs that are mapped to global CIs must not be overwritten. The next section shows how to automatically verify that these properties hold by analyzing the given process models as some aspect of process model and process choreography correctness.

8.5 correctness and consistency discussions

Automating the execution of data dependencies, process data, and data exchange from models only relies on the correctness of the involved models being for (1) process orchestrations a private process model and being for (2) process choreographies a global collaboration diagram and a global data model as well as a local data model and a private process model for each participant. Thus, in addition to the already introduced modeling techniques and the corresponding code generation (SQL and transformation), we introduce a set of approaches enabling to check and ensure correctness. We utilize multiple approaches from literature, especially for the control flow parts of the process orchestration or choreography as well as the ordering of messages (e. g., soundness checking [331] and realizability [69]), and an approach introduced in an earlier chapter of this thesis for data flow correctness (i. e., weak conformance). Additionally, we describe some new means imposed by the introduced concepts regarding proper message definition and sufficient data modeling. Altogether, the following eight correctness properties (CP) have to hold in order to correctly execute the process orchestration or the process choreography.

- **(CP-1)** Correct process orchestration
- **(CP-2)** Sufficient data information specification in orchestrations
- **(CP-3)** Structural compatibility of process choreography
- **(CP-4)** Behavioral compatibility of process choreography
- **(CP-5)** Local enforceability of process choreography
- **(CP-6)** Consistency between global and local process models
- **(CP-7)** Consistency between global and local data models
- **(CP-8)** Correct message definition

Next, we introduce each property and apply it to our build-to-order and delivery example. Properties 1 and 2 ensure the correctness of the private process models, i. e., process orchestrations, such that they can be executed properly after code generation from the models. Both properties apply to concepts introduced in Sections 8.1, 8.2, 8.3, and 8.4. Properties 3 to 6 jointly ensure realizability of the process choreography discussed manifold in literature, e. g., [111] for finite state machines, [37] for UML collaboration diagrams, or [68, 69] in the context of interaction-centric choreography modeling instead of interconnected-centric modeling. Here, we follow the concept of *projection realizabil-*

ity [4, 156, 264, 296] that indicates realizability of the process choreography if and only if the interactions specified in the global collaboration diagram and the interactions specified in the local process models are the same. Properties 7 and 8 specifically arise in the context of our proposal to automate the data exchange. Properties 3 to 8 target the concepts introduced in Section 8.4.

Correct Process Orchestration

Correctness of process orchestrations is generally checked in terms of behavioral correctness, usually meaning the private process model has to be *sound* regarding its control flow; various soundness notions are available for different contexts [72, 195, 262, 331]; also see Section 3.3 for a brief discussion. This is usually done by first mapping the process model to a Petri net, e. g., through the mapping specified in [80], describing the control flow which is then checked for correctness.

As our approach makes occurrences of activities also depend on the presence of data (in a particular state), we have to verify that the private process can still terminate under the given data dependencies and each data-dependent activity can be executed. Checking the correctness of data flow is two-fold; thus, we distinguish two types of data flow correctness: *process model internal data correctness* and *object life cycle conformance*. For process model internal data correctness, the basic problem that can arise is that the data flow precondition of one task will only be satisfied by the execution of a subsequent task or gets invalidated by a concurrent task leading to a deadlock. Thereby, all data classes occurring in the process model must be considered collectively. For object life cycle conformance, one considers object life cycles [154, 288] (also see Definition 3.8 on page 47) as references specifying the data manipulations allowed to be performed by some activity in the process model. The main challenge is to ensure that all reads and writes of data nodes in a process model are covered by some specified data state transitions in the OLCs. We apply the weak conformance checking introduced in Chapter 6 to check both properties. Summarized, we map the private process model to a Petri net containing detailed information about data access from activities and check for soundness following the standard approach of checking behaviorally correctness. A sound net indicates that all utilized data nodes do not block the process model through, for instance, interrelated data access and the specified manipulations adhere to the OLC specifications.

Following Chapter 6, the private process model and the OLCs are separately mapped to Petri nets. We utilize the mappings in [80] and Section 4.7 for the process model; focusing on correctness of the private process, send and receive activities are considered as local activities. We utilize the fact that Petri nets are state machines if and only if each transition has exactly one preceding and one succeeding place [334] for the OLCs. Both Petri nets get integrated through the data states

before the resulting integrated Petri net is transformed to a workflow net that can be checked for soundness (cf. Section 6.2). If the workflow net satisfies soundness or weak soundness, no contradictions exist – neither in the control flow nor the data flow. Thereby, weak soundness indicates that some activities do not participate in process execution or that some states of some data node (object) are never reached in the process model or both.

Next, we illustrate the central features on an extract of our running example, the build-to-order and delivery process. As process model, we consider the private process model from the *Computer retailer* being part of the *request for quotes* choreography from Section 8.4; Figure 110 recalls this process model. Translating the first two activities *Create request* and *Send request* and their access to data class *Request* yields the Petri net shown in Figure 111. The control flow (translated according to [80]) is shown in gray while the data access is shown in white. Each activity is represented by a transition (in grey) that describes the actual task execution and two transitions (in white) describing the check for data pre-conditions (*Initiate data object reads*) and the data writes (*Confirm data object writes*) of the task. These are connected to the transitions covering the actual data operation, e. g., *Write request in data state created*.

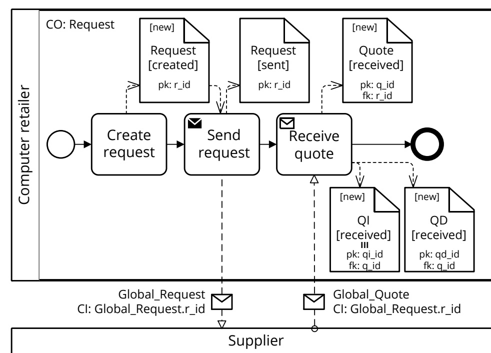


Figure 110: Private process model of the *Computer retailer* recalled from Figure 104 (right).

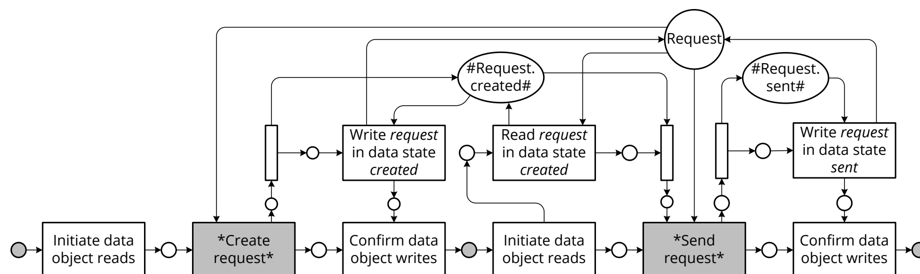


Figure 111: Worklflow net representation of the private *Computer retailer* process model showing the first two activities *Create request* and *Send request*. The gray-colored modeling constructs originate from the control flow mapping [80] while the white-colored modeling constructs originate from the data mapping introduced in Section 4.7.

The places being labeled *class.state* surrounded by dashes, e. g., *#Request.created#* represent the data state of the corresponding data node (object). The place labeled *Request* ensures that no parallel modification operations are performed on objects of the same data class independent of their data state.

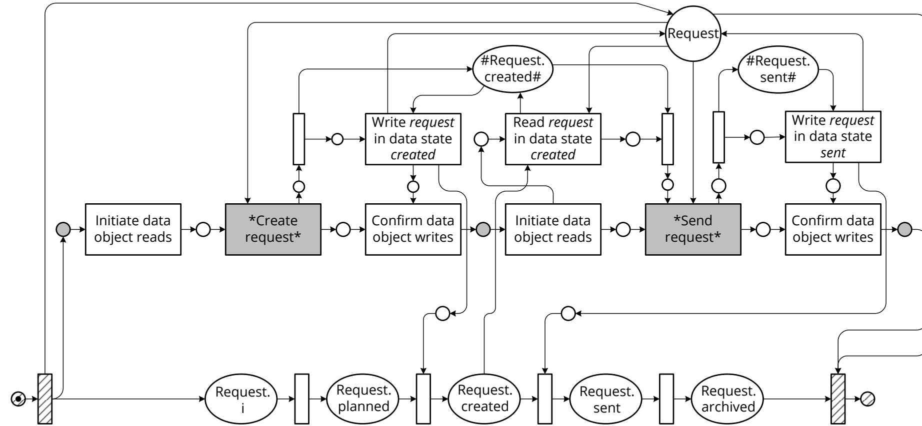


Figure 112: Integrated workflow net of the private *Computer retailer* process model showing the first two activities *Create request* and *Send request* and the object life cycle of the utilized *Request* data class. Application of soundness checking to this workflow net determines correctness of control flow and data flow of the corresponding process model. Here, satisfaction of classical soundness indicates a correct private *Computer retailer* process model.

The shown part of the build-to-order and delivery process utilizes one data class: *Request*. An object of class *Request* may be in states *i* (for initial), *planned*, *created*, *sent*, or *archived* in that order. Indeed, not all these states are utilized in the given process model (see Figure 110) but may be utilized in the computer retailer's process environment. Figure 112 shows Petri net of the process model at the top and the Petri net of the object life cycle at the bottom. The unlabeled places between the process model and the OLC connect both nets resulting in the integrated Petri net. By adding the enabler and collector fragments (shaded modeling constructs; see Section 6.2), the integrated Petri net is extended into a workflow net ready for soundness checking as standard procedure for behaviorally correctness checking. This workflow net (and also the one containing all three activities and both data classes) satisfies the notion of classical soundness and hence the notion of weak conformance such that the private process model of the *Customer retailer* is correct from control and data flow point of view.

Sufficient Data Information Specification in Process Orchestrations

Proper execution of the modeled process orchestrations and process choreographies requires a sufficient data information specification. In process orchestrations, sufficient data information refers to (1) the exis-

tence of data classes and data states, used in the process model, in the data model and (2) the utilization of the correct data attributes as primary key and foreign keys respectively. Ensuring (1), the distinct data classes of all data nodes of the process model are determined followed by determining the states used by nodes referring to each such data class. Considering Figure 110, data classes *Request*, *Quote*, quote details *QD*, and quote items *QI* are identified. For class *Request*, the data states *created* and *sent* are determined while for each of the remaining data classes the state *received* is determined. Both data states for class *Request* are valid (see object life cycle in Figure 112). Assuming, the date states for the other classes are also valid, property (1) is validated. Ensuring (2) requires a check against the data model. Considering the bottom model in Figure 106 on page 255, the correct keys have been used. For data nodes of class *Request*, *r_id* is used as primary key and no foreign key exists; for data nodes of class *Quote*, *q_id* is used as primary key and *r_id* is used as foreign key pointing to class *Request*. Analogously, the keys for data classes *QD* and *QI* are used correctly.

Additionally, in process choregraphies, send tasks also require data nodes (objects) as input to provide a message while receive tasks require data nodes (objects) as output to handle and store a message. For both tasks, these are the data nodes of the class specified in the message definition and all its children. For send tasks, this also comprises the data nodes (objects) that hold the correlation information in terms of correlation identifiers. Therefore, we statically scan the private process model of a participant for send and receive tasks. For each such task, we first retrieve the data class from the corresponding message definition and identify its child classes in the data model. Second, we determine the data classes that contain the specified correlation identifiers. For these global data classes, we determine the corresponding local ones. Then, we check whether all these local classes are read (send task) or written (receive task) in some data state indicated by data nodes. If not, a violation occurred.

Applying this property to the private computer retailer process model of the build-to-order and delivery process (see Figure 110) shows that the data information is sufficiently specified. Activity *Send request* refers to message *Request Message* that transmits an object of class *Global_Request* that does not have children. Additionally, the correlation identifier *Global_Request.r_id* is stored in the same *Global_Request* data object. The corresponding local object is of data class *Request* on the computer retailer side because each attribute of the *Global_Request* object is mapped to the local *Request* object (see the schema mapping Figure 106 on page 255). Thus, reading data node (object) *Request* in state *created* satisfies this property. Activity *Receive quote* refers to message *Quote Message* that transmits a data object of class *Global_Quote* that contains *Global_Article*s. As shown in the schema mapping in Figure 106, an object of class *Global_Quote* is represented by local objects of classes *Quote*

and *Quote Details QD* while an object of class *Global_Article* maps to local object of class *Quote Item*. Since activity *Receive quote* writes these three data nodes (objects) in data state *received*, the property is satisfied. Finally, considering activity *Send quote* from the supplier, data node of class *Request* as input to this activity satisfies the correlation identifier requirement.

Structural Compatibility of Process Choreography

This property exists in two forms: strong and weak structural compatibility [67]. Strong structural compatibility requires that each message flow edge in the global collaboration diagram has a source and a target activity in the global collaboration diagram, i. e., each message flow connects a send task with a receive task of different participants such that each sent message gets received within the process choreography. This is important to ensure that all information required to automate the message exchange is available. Weak structural compatibility allows participants to also receive messages that where sent by someone not participating in the choreography under the assumption that these messages received from an external source do not interfere with the choreography; e. g., they are received in an independent branch of the respective local process model and the corresponding receive task is not executed in the context of the process choreography. Since we focus on the automated execution of data exchange in process choreographies, we have to assume strong structural compatibility so that each received message is also sent by some other participant known in the process choreography.

Assume, a message flow only has a target activity but no source activity. Then, it cannot be ensured that the message follows the negotiated global contract, the global data model, and as such no reliable information is given about the message's content. Assume, a message flow only has a source activity but no target one, i. e., it is sent to an external recipient. Then, the sent message conforms to the global contract agreed on for this process choreography, but it may not conform to some other choreography for which the receive task is part of. Therefore, only strong structural compatibility guarantees correct process execution and message handling. Indeed, if messages received from some external source or if messages sent to some external target conform to the respective global data models, we can process them correctly. Since we cannot guarantee that, this is out of scope.

Structural compatibility checking is a syntactical check on the global collaboration diagram. In the request for quote choreography from Section 8.4 presented again in Figure 113, two message flows are modeled. The *Request Message* connects tasks *Send request* of the computer retailer and *Receive request* of the supplier while the *Quote Message* connects tasks *Send quote* of the supplier and *Receive quote* of the computer retailer. Since both message flows have a source and a target activity

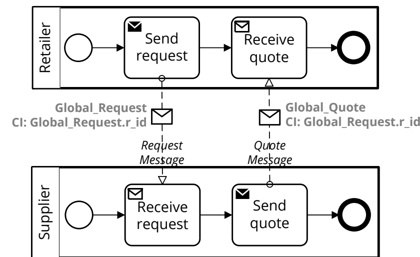


Figure 113: Global collaboration diagram of request for quote choreography recalled from Figure 104 (left).

that are part of the global collaboration diagram, the property of strong structural compatibility is satisfied.

Behavioral Compatibility of Process Choreography

Behavioral compatibility checks whether behavioral dependencies are met, i. e., the message flow does not block execution of local process orchestrations. This property can be checked by using the approach from Martens [195], where semantic compatibility of workflow modules is introduced. Workflow modules are workflow nets with additional communication places. To apply this approach to a process choreography modeled as global collaboration diagram, one workflow module needs to be created for each participant.

A workflow module contains representations of the participant's send and receive tasks as transitions that are labeled with the corresponding message name preceded by an exclamation (send) or question (receive) mark. Further, it contains representations of the message flows as places that get labeled with the corresponding message name and that have either an incoming edge (send) or and outgoing edge (receive) but not both – referred to as communication places. The resulting workflow modules are combined by using the communication places through label matching. Additionally, an initial and a final place are added to the combined workflow module to fulfill structural soundness [332]. The initial (final) place is connected via transitions to the initial (final) places of the single workflow modules resulting in a workflow net if strong structural compatibility is satisfied – a prerequisite to apply behavioral compatibility. Finally, the combined workflow module is checked for weak soundness indicating whether the global collaboration diagram satisfies the property of behavioral compatibility.

The workflow modules for the computer retailer and the supplier of the request for quote process choreography (see Figure 113) are shown in Figure 114, where the transition labeled *!Request Message* refers to activity *Send request* of the computer retailer while transition *?Request Message* refers to activity *Receive request* of the supplier. The communication place labeled *Request Message* in both workflow modules refers to the corresponding message between those activities in the global collab-

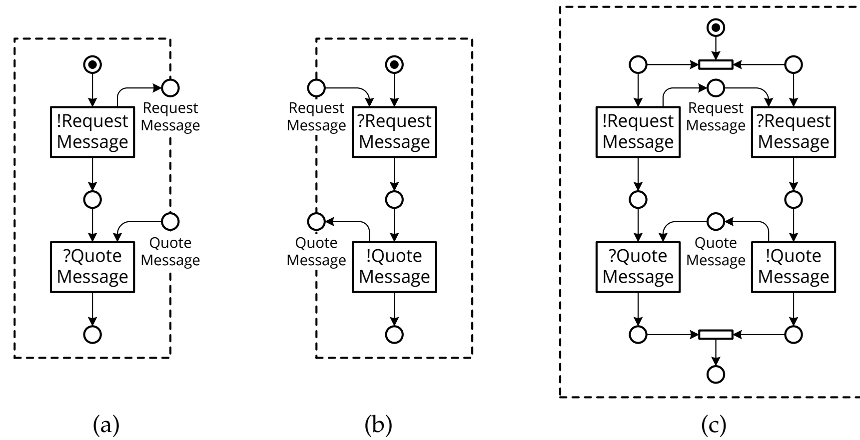


Figure 114: Workflow modules for (a) the computer retailer and (b) the supplier of the request for quote choreography presented in the global collaboration diagram in Figure 113. (c) shows the combined workflow module of (a) and (b) fulfilling the soundness property.

oration diagram. Analogously, the *Quote Message* is represented in the given workflow modules. Combining both workflow modules reveals that the notion of soundness [331] and therefore also weak soundness is fulfilled and thus, the request for quote choreography satisfies the property of behavioral compatibility.

Local Enforceability of Process Choreography

Local enforceability [68, 387] determines whether the ordering of messages is strictly ensured, i. e., whether for each message flow, it is unambiguously specified when the corresponding message is sent [69]. For instance, in case that A shall send a message to B before C sends one to D, local enforceability is not ensured. As C is not involved in the previous message between A and B, C does not know whether the message is already sent or not. Thus, C may send the second message before the first one of A. Different rules and an algorithm to ensure local enforceability are introduced in [387] and formally grounded in [68]. These rules also include the one that subsequent interactions must unambiguously know about the directly preceding one as sketched above.

In the request for quote example, the supplier knows when to send the *Quote Message* because she participates in the *Request Message* message exchange. Thus, the order of those two messages is strictly ensured. Since there do not exist more message flows in the given collaboration diagram, this process choreography is locally enforceable.

Consistency between Global and Local Process Models

The public-to-private approach [342] defines means to ensure the consistency between global and local process models in terms of control flow. Originally, it was defined for workflow nets but since the concepts were

introduced independently from the process description language, they can be utilized generically; e. g., applied to BPMN process models. The public-to-private approach bases on refinement of the global process model and allows three types of operations: (i) addition of a loop that starts and ends in the same place, (ii) addition of additional process fragments by substituting some control flow in the process orchestration with this fragment, and (iii) addition of a concurrent path that is synchronized again with the main path from the global model. A private process model being refined by utilizing only operations (i), (ii), and (iii) is consistent to the global process model by structure.

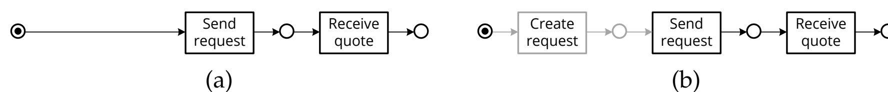


Figure 115: Workflow nets for the (a) global and (b) private process models of the computer retailer from the request for quote example presented in Figure 113.

Formally, the link between global models and local models in terms of a subclass-superclass relationship is established by utilizing [20, 338] that in turn bases on a strict notion of equivalence: *Branching bisimulation* [353]. Two process models satisfy the notion of branching bisimulation if and only if they mutually simulate each other's behavior disregarding local behavior; i. e., considering only control flow nodes existing in both process models, they must provide identical behavior. In the context of this chapter, the global process model of a participant is taken from the global collaboration diagram; e. g., for the computer retailer from the request for quote example, the global model is the upper pool in Figure 113. Figure 110 shows the private process model of the computer retailer. Checking both process models for branching bisimulation reveals that the notion is satisfied. Since branching bisimulation is checked on workflow net level, the required representations for the global and local level are given in Figure 115. Abstracting from the highlighted transitions in the local model (Figure 115b) that have been added through operation (ii), both workflow nets are identical and thus allow simulation of each other.

Consistency between Global and Local Data Models

Consistency between a global and a local data model is achieved if and only if the local data model of a participant comprises all data classes of the global data model that are relevant for that participant. A global data class is *relevant* if it occurs in a message sent/received by the participant (either as payload data class or as child class). Moreover, any class that is related (via containment or association) to a class in a message sent/received by the participant is also relevant (cf. class dependencies in the data model). Two properties have to hold: (1) each relevant global

data class needs a representation in the local data model and (2) the dependencies between global data classes (containment, association) must also be reflected in the local data model. Both properties are properties of the local data model and the global-local schema mapping.

Regarding (1), we call the local data model and schema mapping *complete* (wrt. a set of relevant data classes) if and only if for each attribute j of each relevant global data class there exists in the local data model some data class and attribute j' that is mapped to a.

Regarding (2), we say that local data model and schema mapping *satisfy dependency congruence* (wrt. a set of relevant data classes) if and only if the following three properties hold: If a global data class *a* contains another global class *b*, then one of the local representatives of *a* must contain one of the local representatives of *b* with the same cardinality. If a global data class *A* is connected to another global class *B* via an association, then one of the local representatives of *A* must be connected to one of the local representatives of *B* via an association of the same cardinality. If two attributes j_1, j_2 of a global data class c are mapped to attributes in different local classes c'_1, c'_2, then c'_1 and c'_2 are related by 1:1-relations only.

Each participant can locally check completeness and dependency congruence on the data models and the chosen mapping by graph traversal. For example, consider the schema mapping for the computer retailer in the request for quote example given in Figure 106: All attributes of class *Global_Request* map to some attribute of the local class *Request*, all attributes of the class *Global_Quote* map to some attribute of the local classes *Quote* and *Quote Details QD* respectively, and all attributes of the class *Global_Article* map to some attribute of the local class *Quote Item QI*. Thus, the schema mapping is complete. Regarding dependency congruence, *Global_Request* is associated to *Global_Quote* and so are *Request* and *Quote*, respectively; the 1:n-containment relation between *Global_Quote* and *Global_Article* is represented by the 1:n-containment relation between *Quote* and *Quote Item QI*; finally, the attributes of *Global_Request* are mapped to two local classes *Quote* and *Quote Details QD* which are related by a 1:1-association. Thus, the schema mapping is complete and dependency congruence is satisfied showing that the local data model of the computer retailer is consistent to the global data model.

Correct Message Definition

A message definition consists of a message name indicating its type, a list of correlation identifiers, and a data class as payload. A message definition may be inconsistent with the choreography both syntactically and behaviorally.

Syntactically, message names must be unique. While a process choreography may specify multiple message flows, each transferred message has to have a unique name to ensure behavioral compatibility (see

above). This can trivially be checked by searching all defined message names for duplicates. Moreover, the payload data class of a message must exist and each of its correlation identifiers must be valid attributes of the global data model. For the payload data class, we check whether the referenced class is defined in the global data model. Each correlation identifier is a fully qualified attribute; thus, we check for the existence of the referenced class in the global data model and check whether it contains the corresponding attribute. The message definitions in the request for quote example in Section 8.4 satisfy all these properties.

Besides basic type correctness, correlation identifiers also have to be consistent with choreography behavior. First, in request–response scenarios, e. g., a request for quote as presented above, the sender of the request must be able to correlate the response message to the request. Thus, request and response must share at least one correlation identifier; i. e., the intersection of their correlation identifiers is non-empty. However, as a process choreography may send two independent requests in opposite directions, the case of non-overlapping correlation identifiers should only be considered as a warning to investigate the situation, not as a violation.

Second, it must be ensured that each correlation identifier is initialized by sender and receiver upon first usage. A correlation identifier may be initialized only once. Any subsequent use of that identifier may only be used for matching. In our model-driven approach, a correlation identifier is always initialized when it is also in the payload of the message as the receiver will store the message contents and thus set the correlation information; matching occurs when the correlation identifier is *not* in the payload, so already existing correlation values are only compared but not overwritten; see correlation part at the end of Section 8.4. For example, the *Request* message with data class *Global_Request* as payload of Figure 116 (left) initializes the correlation identifier *Global_Request.r_id*; the *Quote* message with data class *Global_Quote* as payload uses correlation identifier *Global_Request.r_id* for matching only. Properly distinguishing the first use of an identifier from any subsequent use requires a behavioral analysis. We show that checking for correct utilization of correlation identifiers can be reduced to checking soundness of a Petri net [334].

We translate the global collaboration diagram to a Petri net such that the Petri net can always terminate in a final state if and only if all correlation identifiers are initialized and matched correctly. We illustrate our generic translation on the request for quote example from Section 8.4. In principle, we map the control flow and message exchange of the global collaboration diagram to a Petri net – in the context of BPMN as proposed by Dijkman et al. [80]: Each activity a is mapped to a transition t_a; each sequence flow between two activities a and b is mapped to a place between t_a and t_b; each message flow between activities a

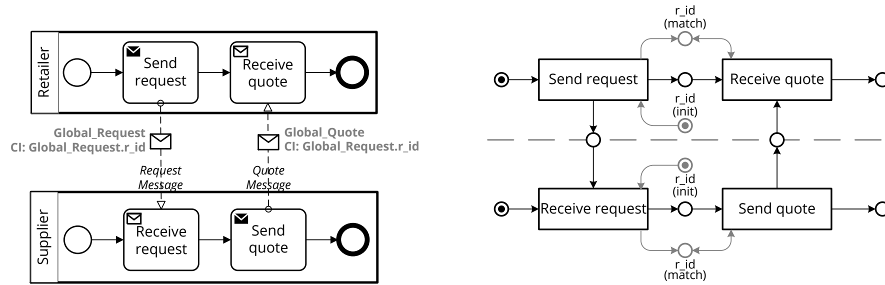


Figure 116: Translation of Figure 104 (left) to a Petri net with correlation handling to check correct initialization and matching of correlation identifiers.

and c (of different participants) is mapped to a place between t_a and t_c. More advanced constructs such as gateways or events are translated accordingly, see [80] for details for BPMN. [190] provides a survey on Petri net mappings for various process description languages.

We extend such standard translation to distinguish initialization of a correlation identifier from subsequent matching. For each correlation identifier *id* specified in some message, we introduce two places for each participant; place p_{id}^{init} describes the situation where *id* has not been initialized yet and is initially marked; place p_{id}^{match} represents that *id* has been initialized already. For each activity A sending/receiving a message where the correlation identifier *id* is also contained in the message payload, A initializes *id*. In this case, we add p_{id}^{init} to the preset of t_A and p_{id}^{match} to the postset of t_A expressing that A initializes *id*. If *id* is *not* in the payload of the message sent/received by A, then A matches *id* (assuming it has been initialized). In this case, we add p_{id}^{match} to both the preset and the postset of t_A. By construction, the choreography correctly uses its correlation identifiers if and only if the Petri net can always reach its final state (end places of each participant are marked and there are only tokens on the correlation identifier places). This can be checked using existing verification techniques such as model checkers [103] or by transforming the Petri net into a workflow net and checking for soundness [331].

Thereby, the presented check ensures that all violations are determined but may also provide false positives; e. g., overwriting the value of a correlation identifier with the same value in case the same global object is sent there and back. Thus, during violation handling, the false positives need to be identified and marked as correct. A completely automated checking of correlation identifier initialization and matching is subject to future work.

Figure 116 (right) shows the result of translating the global collaboration diagram to the left in the described way. The gray dashed line visually separates the Petri nets for both participants and the correlation identifier *Global_Request.r_id* is abbreviated as *r_id*. The first message ex-

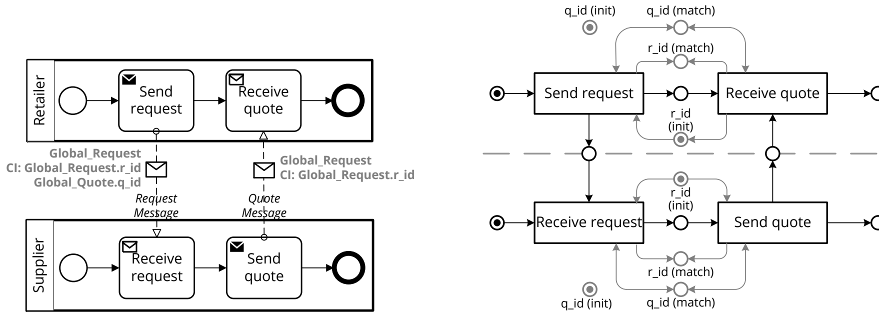


Figure 117: Example of incorrect correlation handling; the resulting Petri net (right) cannot reach the final state.

change initializes *r_id* and the second message exchange matches the initialized *r_id*; this net can always reach its final state.

Figure 117 shows a choreography with two correlation handling errors: the first message initializes *Global_Request.r_id* and only matches *Global_Quote.q_id* (not contained in *Global_Request*), the second message re-initializes *Global_Request.r_id*. The resulting Petri net in Figure 117 (right) is not sound: transition *Send request* cannot occur as *q_id (matched)* is (and remains) unmarked; no transition can move the token from *q_id (init)* to *q_id (matched)* as no message carries the *Global_Quote* object. This first problem could be resolved by removing *Global_Quote.q_id* from the correlation identifier of the first message; then the first message exchange would succeed. However, the second message exchange would still fail, since *Send Quote* needs a token on *r_id (init)* which has already been consumed by *Receive Request*; *Send Quote* cannot occur and the Petri net cannot reach its final state.

The above checks only cover syntactic and behavioral correctness. Checking whether the correlation identifiers are semantically correct is out of scope of this section. This needs to be validated by the process expert manually.

8.6 EVALUATION

We evaluated the approaches introduced in Sections 8.1 to 8.4 for automatically executing process models with complex data dependencies as well as automating the message exchange in process choreographies by implementation. The operational semantics translates model-features into executable code. This code is quasi-platform-independent for two reasons: (1) we only generate platform-independent code in standard technologies (SQL, XML Query Language (XQuery)) and (2) this code is well-encapsulated having no external dependencies except for invocation based on the process model and its input/output being in Extensible Markup Language (XML) format. More specifically, we generate XQuery code [380] to transform between local and global data in XML format. The XQuery code is generated by a small extension

module of the process modeling tool that is aware of the global and local UML data model. The generated XQuery code is stored in the process model in a predefined extension point and reads and writes data from dedicated process variables. For data access to the local data storage and for handling complex data dependencies, we generate SQL queries [46, 53, 147]. The SQL queries are generated at run-time by a separate, encapsulated module that takes as input the current model-based task definition and reads from/writes to those dedicated process variables. The code for query generation is invoked according to the utilized process description language (here: the BPMN standard) along the life cycle of an activity [370]: checking for data existence upon activity enablement, reading data upon activity start, and writing data upon activity termination. The data transformation code in the process model is invoked upon reading data input for a send task and upon writing data output for a receive task.

Focusing on the data exchange and demonstrating the feasibility, we additionally implemented the *service interaction patterns* [19] which capture basic forms of message-based interaction. We first introduce the implementation details again using BPMN as representative before we discuss the application to the service interaction patterns.

Implementation

In the spirit of building on existing standards and techniques, we implemented our approaches by extending the *camunda Modeler*, a modeling tool supporting BPMN, and the *camunda BPM platform*, a Java-based, lightweight, and open source process engine specifically tailored for a subset of BPMN process models. Both are forked from the Activity [2] project. The engine enacts process models given in the BPMN XML format and supports standard BPMN control flow constructs. Data dependencies are not enacted from the process model, but are specified separately. Introducing data-awareness as explained in the previous sections of this chapter into the modeler and the engine, we added only few extensions at well defined points.

Process orchestration

For supporting process orchestrations, we first extended the XML specification by utilizing *extension elements*, which the BPMN specification explicitly supports to add new attributes and properties to existing constructs. We added the *case object* as additional property to the (sub-)process construct. The data node was extended with additional properties for *primary key* (exactly one), *foreign keys* (arbitrary number), and the *data access type* as attribute. The BPMN parser was extended to read data nodes – including the new attributes and properties – and data associations.

The actual execution engine was extended at three points: before invoking the execution of an activity to check the preconditions of an activity, upon activity execution to retrieve the actual process data, and during completion of an activity to realize the postconditions, all three

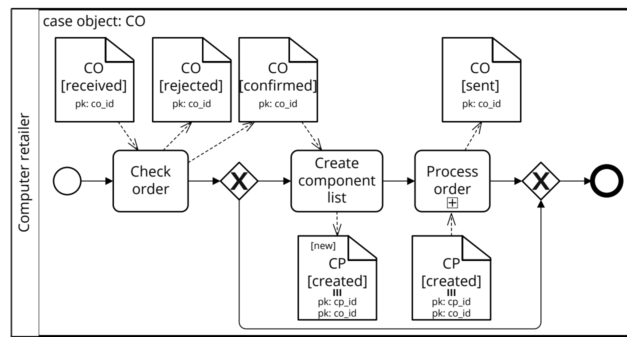


Figure 118: Extract of build-to-order and delivery process from Section 2.4 tailored to show the implemented procedure for SQL derivation.

with respect to the modeled data nodes. At either point, the engine checks for patterns of data input and output objects and categorizes them. For instance, in Figure 118, *Customer order CO* is input and output to activity *Check order* in different states. The engine classifies this as a "*conditional update* of case object customer order *CO*". The data operations at task *Create component list* would be classified as "*conditional creation* of multiple data objects that depend on the case object (1:n relationship)". Classification proceeds from most specific to most general patterns.

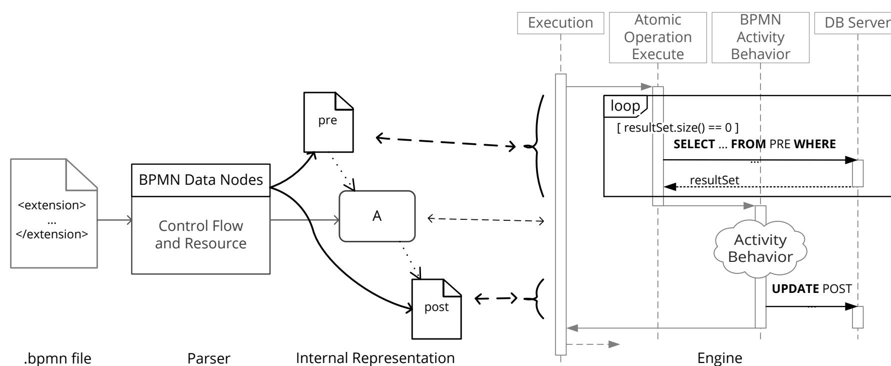


Figure 119: Data handling is a conservative extension to data format, parser, internal representation, and execution engine.

When invoking an activity, for each matching precondition pattern a corresponding SQL select query is generated to read whether the required data objects are available (cf. Sections 8.1 and 8.2). The query assumes that for each data class of the process model there exists a table holding all objects of this class and their attributes. If there is at least one data object in the right state, the SQL query returns a value greater zero. The engine repeatedly queries the database until a number greater zero is returned; i. e., the task is enabled, as shown in Figure 119. Then the activity is executed. Upon activity start, the actual process data is retrieved by generating corresponding select query as discussed in Section 8.3. Upon activity termination, an SQL insert, update, or

delete query is generated for each matching postcondition pattern and executed on the database.

Process choreography

For supporting process choreographies, the modeling tool was additionally extended with the annotations for messages and correlation identifiers. Message types of the global data model are specified in XSD [381] and a simple editor allows to create an attribute-wise schema mapping from the global to the local data model. Once a private choreography model has been completed, the user can automatically generate XQuery expressions at the send and receive tasks to transform between the local and the global data model. The engine was extended with a messaging endpoint for sending and receiving messages in XML format to correlate messages, to read and write local data objects by generating SQL queries from process models, and to process messages as described in Section 8.4. Analogously to the concepts also our implementation does not address requirement CER-4 (message routing); in particular if the receiving task is not in state *running* to receive the incoming message, the message will be discarded. Making the process layer compatible with error handling of the message transport layer is beyond the scope of this chapter.

Summary

Altogether, we had to extend the camunda BPM platform at merely 6 points to realize our concepts: (1) at the XML, (2) at the parser and internal representation, (3) when checking for enabling of activities, (4) when starting an activity, (5) when completing an activity, and (6) during execution of send and receive tasks to send/receive messages. The extended engine, a graphical modeling tool, example process models and choreographies, screencasts, and a complete setup in a virtual machine are available together with the source code at `http://bpt.hpi.uni-potsdam.de/Public/BPMNData/`. With the given implementation, a user can model data annotated processes in BPMN and directly deploy the model to the extended engine which then executes the process *including* all data dependencies. No further hard-coding is required since all information is derived from the process model.

Platform independence

Although implemented for the camunda BPM platform, our solution is generally applicable to activity-driven process technology quasi-platform-independent as follows. The solution's general idea is to present small, isolated modules of functionality that is orthogonal to existing activity-driven process technology. Code generation within these modules is platform-independent at run-time by using process models with our generic annotations (explained for extending BPMN) as input and standardized code as output. The modules are encapsulated with well defined interfaces resulting in purely local transformations based on the input models only. Furthermore, the modules are invoked locally at well defined points in the life cycle of an activity – the four points discussed above: checking for data existence upon enablement, retrieving the actual process data at activity start, storing data during activity

completion, as well as sending a message and receiving a message during the running state. For invoking the modules, they are inserted into an activity-centric process engine of choice – in our case, this is the camunda BPM platform. Checking for data existence, retrieving data upon activity start, and storing data upon activity termination map the input models to standard SQL queries that are run completely platform independent. Data transformations between global and local data required by sending and receiving a message respectively depends on the used platform at design-time, since the corresponding query language must be chosen. The camunda BPM platform utilizes an XML-based message exchange and thus, we use XQuery [380]. In case, a platform uses a different communication interface, e. g., tree-structured message exchange as JSON [146], another query language must be used; in the case of JSON, JQuery [152] may be used. However, the concepts stay the same; only the messaging API changes resulting in possibly few adaptation changes. Since data transformation is specified statically (cf. the mapping between global and local data models in Section 8.4), one has to define this interface once and the overall approach dynamically generates the required messages and data objects.

Service Interaction Patterns

To demonstrate the feasibility of the data exchange automation, we implemented the *service interaction patterns* [19] which capture basic forms of message-based interaction. The patterns are available along our implementation at http://bpt.hpi.uni-potsdam.de/Public/BPMNData/.

Next, we briefly describe each pattern and how it can be realized using our approach from Section 8.4. Thereby, we discuss the actual execution of the corresponding communication such that we act on the data object instead of the data class level. We reuse the pattern classification into *single-transmission bilateral*, *single-transmission multilateral*, *multi-transmission*, and *routing* interaction patterns for structuring. For each pattern, we make some observations regarding the *method* by which each pattern can be realized with our approach.

SINGLE-TRANSMISSION BILATERAL INTERACTION PATTERNS. Patterns in this category describe the interaction of two participants A and B that each send/receive one message.

P1 Send and P2 Receive. Participant A sends a message which has to be received by participant B. The challenges are to generate and send a message, to correlate a message based on an initialized or uninitialized key, and to process a received message.

Figure 120 shows the data model for this choreography (P1 and P2 only utilize Request messages and objects) with the global part in gray. These global data classes and their attributes have local counterparts (white classes) as indicated by the dashed arrows.

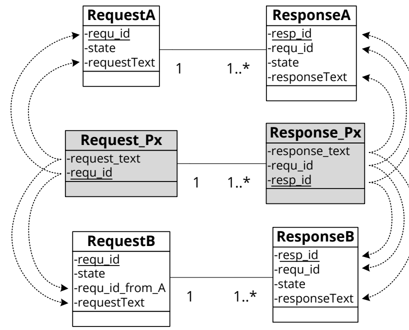


Figure 120: Data model for patterns P1 to P4.

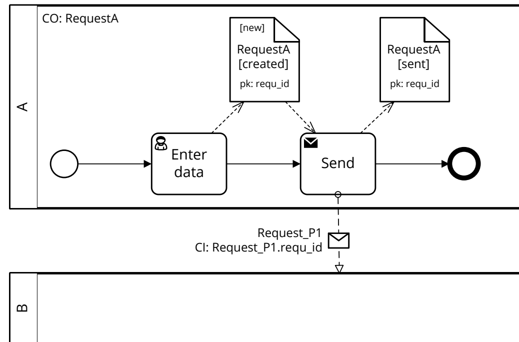


Figure 121: Pattern P1: send.

Figure 121 shows the pattern for A sending a message to B. Thereby the local object *RequestA* is transformed into the global object *Request_P1* and the correlation key *Request_P1.requ_id* is set from the primary key *requ_id* of *RequestA*.

Figure 122 shows the pattern for B receiving the message from A. Thereby, the message being received creates a new process instance; the global object *Request_P1* is mapped to the local object *RequestB* with its own primary key; the correlation information in *Request_P1.requ_id* is mapped to an attribute *RequestB.requ_id_from_A*. This correlation key is initialized upon receipt.

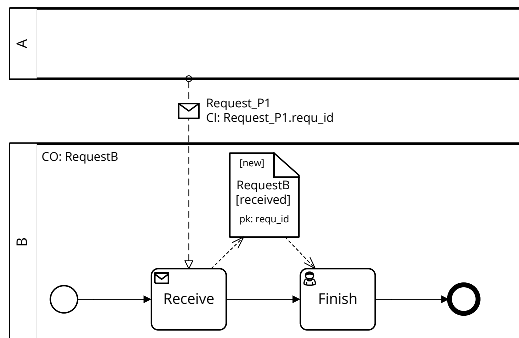


Figure 122: Pattern P2: receive.

Method. As shown in Figures 121 and 122, the first message in the interaction initializes the correlation key. A straightforward way of doing this is to use the *primary key of the source data object* (i. e., *RequestA.requ_id*) as the correlation key. Note that the source data object does not necessarily have to be the case object of the process.

P3 Send/Receive. Participant A sends a request to B and receives a response. The challenge is to correlate the response message to the instance of A that sent the message.

Figure 120 shows the data model for this choreography. Figure 123 shows the pattern for A sending a message to B and then awaiting the corresponding response. Correlation of the response to the request is achieved by including in the response message the correlation information *Request_P3.requ_id* that was sent to B in the request message. This way, only responses that match the request will be received. The response message uses the correlation key *Request_P3.requ_id* which had been initialized from the primary key *RequestA.requ_id* when sending the original request. Now, when receiving the response, the message is correlated to the instance of A where *RequestA.requ_id* matches the value of *Request_P3.requ_id*. Moreover, when generating the local object *ResponseA* upon receipt, the local foreign key *ResponseA.requ_id* is initialized from the foreign key *Response_P3.requ_id*, thus the new *ReponseA* object correctly points to its parent *RequestA* (see Figure 120).

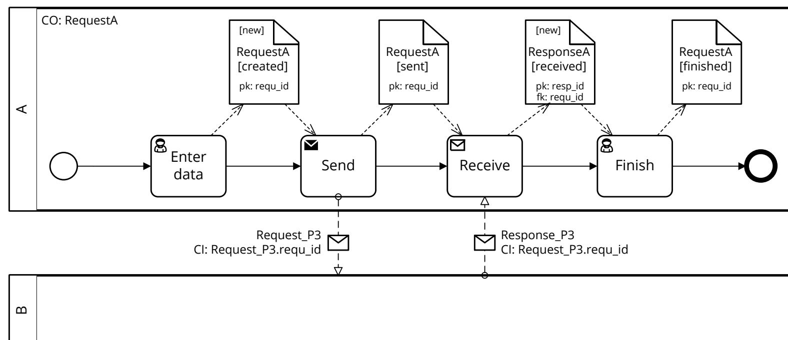


Figure 123: Pattern P3: send/receive (participant A).

Figure 124 shows the pattern for B receiving the message from A and producing a response. Thereby, the message being received creates a new process instance; the global object *Request_P3* is mapped to the local object *RequestB* with its own primary key; the correlation information in *Request_P3.requ_id* is mapped to an attribute *RequestB.requ_id_from_A*. This correlation key is initialized upon receipt and used again when generating the response message *Response_P3* from the local object *ResponseB*.

Method. Figure 124 shows a natural design for requests and responses. The response message from B to A is generated from the local *ResponseB* object which corresponds to the *RequestB* object received earlier. This way, the global and the local data model are consistent on the key re-

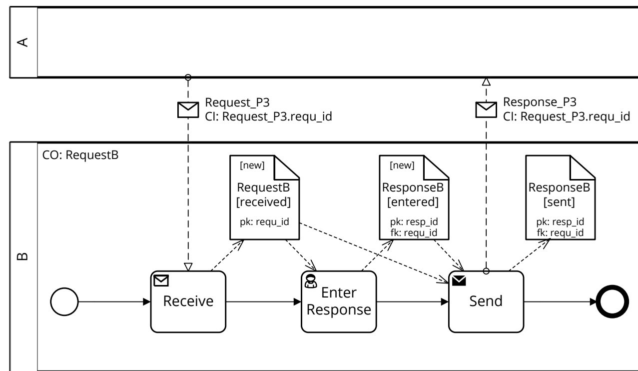


Figure 124: Pattern P3: send/receive (participant B).

lations between request and response. During modeling, the modeler has to ensure that (1) global attributes properly map to local attributes, and (2) the local attributes are accessible by the send and receive activities. In particular, receiving activities must not map correlation keys to primary keys of their local objects (as these are handled locally); rather, the correlation key is mapped to a separate attribute (i. e., *RequestB.requ_id_from_A*) which acts like a foreign key to the conversation between A and B. As a consequence, the replying activity must have access to the data object that holds the correlation key; e.g., *Send* in Figure 124 reads *RequestB*.

SINGLE-TRANSMISSION MULTILATERAL INTERACTION PATTERNS. Patterns in this category describe the interaction of participant A with multiple participants B1, B2, ..., each sending/receiving one message.

P4 Racing Incoming Messages.[12] Participant B expects one or more messages from participants A1, A2, ..., C1, C2, ... and will consume the first arriving message; messages arriving later may be discarded or queued. Optionally, a timeout is allowed in case no message arrives. The challenge is to ensure that B consumes exactly one message or the timeout occurs.

Figure 125 realizes the pattern for recipient B expecting requests from two different participants (A and C). The behavior of A, C, ... is described in Figure 121 (P1: send). Figure 120 shows the data model for this choreography for A and B; participant C has its own global request message *Request_P4c* and its own local data model, i. e., *RequestC*). B realizes the pattern by using an event-based gateway that has two subsequent intermediate message events. Each awaits a message from a different participant (A or C in this case). Whichever message ar-

12 Pattern P4 is not directly supported by the generic framework introduced in this thesis due to the necessity of intermediate events. Since they are supported by BPMN [243] and they integrate into the introduced concepts, our implementation supports P4. This also holds true for further patterns requiring intermediate events for communication.

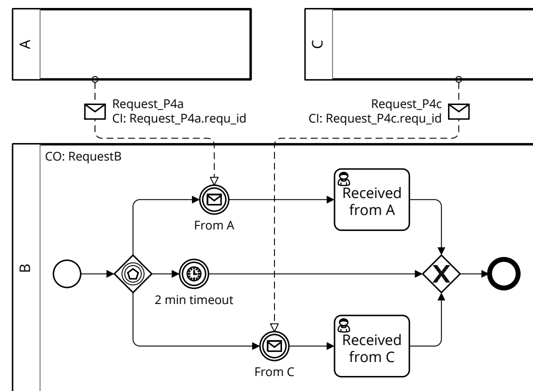


Figure 125: Pattern P4: racing incoming messages (participant B), with message events.

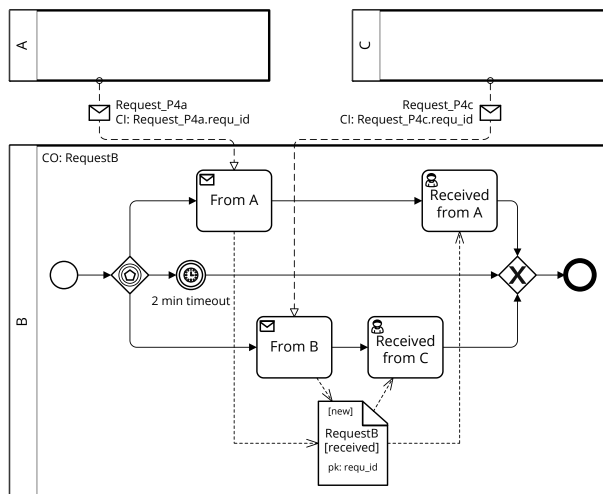


Figure 126: Pattern P4: racing incoming messages (participant B), with receive tasks.

rives first will be consumed and the corresponding path will be taken. The timer event after the event-based gateway implements the timeout mechanism. The model in Figure 125 only implements the correlation handling: depending on which message is received first, the corresponding correlation property is set. The model does not implement transformation of the message into local data, since our concepts (and the BPMN standard) do not allow data transformation at events.

Figure 126 shows an alternative realization where the message events are replaced by receive tasks. The event-based gateway will follow the sequence flow of the receive task which first has a message to consume. This also allows to transform data. However, by the time of our research the *camunda BPM platform* did not support receive tasks after an event-based gateway and thus this pattern could not be executed.

Method. When receiving messages from multiple senders, both can use different messages *and different correlation keys*; i. e., *Request_P4a.requ_id* and *Request_P4c.requ_id*). Our approach allows mapping different global

objects to the *same local object* at the receiver; this way the receiver can operate with a uniform case object, i. e., *RequestB*, in all circumstances. Any difference between incoming global objects can easily be mapped to different local child objects of the receiver that can be created by the receiving activity.

P5 One-to-Many Send, P7 One-to-Many Send/Receive. In P5, participant A sends out a request to multiple participants B1, B2, ..., so that each participant receives one request. In P7, each participant B1, B2, ... then sends a reply to its request. The challenge is to generate multiple messages with different correlation information and to then correlate the incoming responses to the original request.

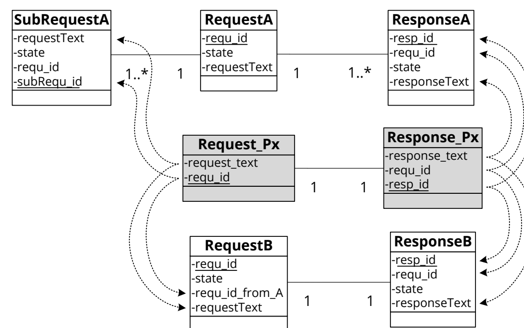


Figure 127: Data model for patterns P5 and P7.

Figure 127 shows the data model for these patterns; the difference to the model of Figure 120 is that the local *RequestA* now has multiple subrequests and related responses that each map to a global message.

Figure 128 realizes the behavior of A for P7 (and thus also for P5). First, we generate a separate instance of data object *SubRequestA* for each request that is going to be sent. The number of requests to be generated is set in the process variable *numSubRequests*. Then, for each instance of *SubRequestA*, we create a new instance of the multi-instance subprocess; each handling one instance of *SubRequestA* as case object. Each *SubRequestA* object is then mapped to a message *Request_P3* and sent. By construction of the data model, the primary key *subRequ_id* of the case object *SubRequestA* maps to the correlation key *Request_P3.requ_id* of the message. Thus, each subrequest has its own correlation key that is also held by the corresponding instance of the multi-instance subprocess.

Participant B can handle the message as described in Figure 124 and sends the response. The response is correlated by A to the subprocess instance where the case object *SubRequestA* with primary key *SubRequestA.subReq_id* has the same value as the correlation identifier *Request_P3.requ_id* of the response message. The received global object *Response_P3* is transformed to the local object *ResponseB* which is related to the top-level case object *RequestA* by the semantics of the data annotations.

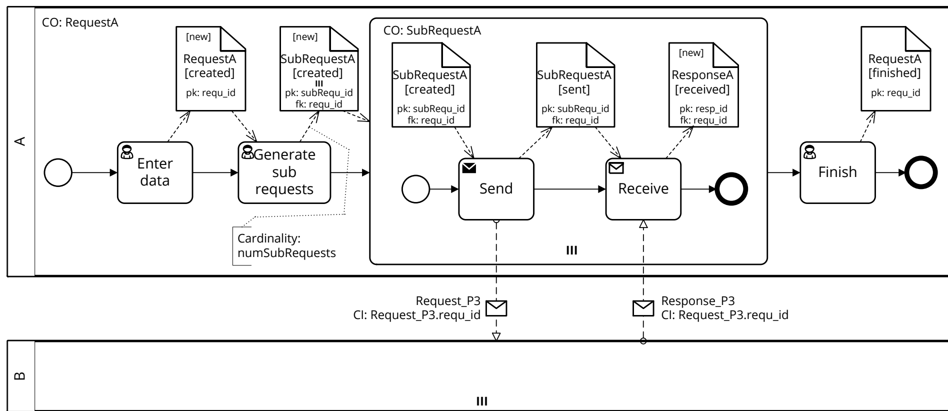


Figure 128: Pattern P5 and pattern P7: one-to-many send/receive.

Method. This pattern also shows that the same global data model of one request and one response message (used in Figures 120 and 127) can be implemented locally in very different ways. In P3, participant A sends just one request generated from the top-level case object, whereas in P7, participant A sends multiple requests to different recipients. Each of the requests in P7 is generated from a child object of the top-level case object. By using these child objects as case objects of the subprocess, we leverage their primary keys as canonical correlation identifiers for each subrequest and response. The recipient process B (Figure 124) cannot distinguish whether A follows pattern P3 or P7 as each request has its own correlation key and thus is handled by a different instance of B. This demonstrates that our approach allows for properly hiding implementation details of processes using the same global data model.

P6 One-from-Many Receive. In P6, participant A receives from an unknown number of participants B1, B2, B3, ... one message per participant. The sent messages logically correspond to each other and thus have to be correlated to the same instance of A. The challenge is to dynamically let the first message set the correlation information based on which the other messages are correlated to that instance. A message with a different correlation information has to be correlated to a different instance. Pattern P6 is not covered by Figure 128 as there the correlation information is set by A, whereas in P6, the correlation information is remotely set by B1, B2, etc.

Figure 129 shows the data model for this pattern. Here, we abstract from the senders B1, B2, ... and only focus on the recipient A receiving multiple items for a document.

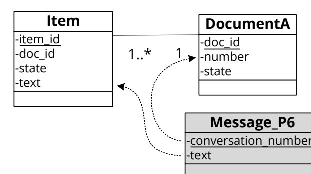


Figure 129: Data model for pattern P6.

We distinguish situations where A is already running from situations where the first received message with a new correlation value creates that instance. The model

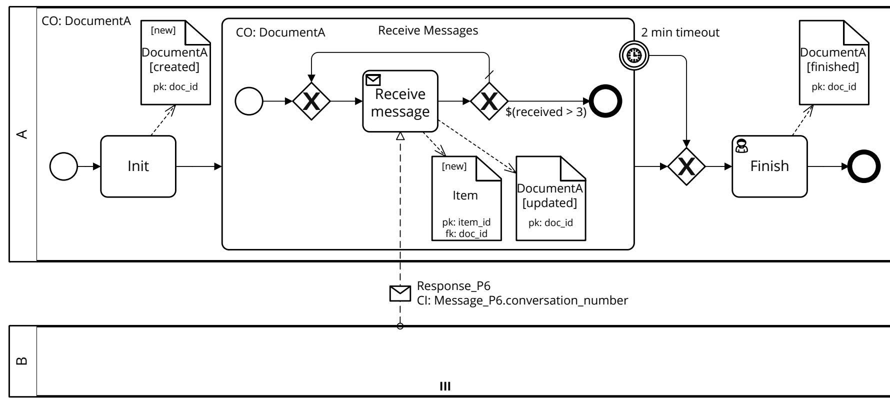


Figure 130: Pattern P6: one-from-many receive (running instance).

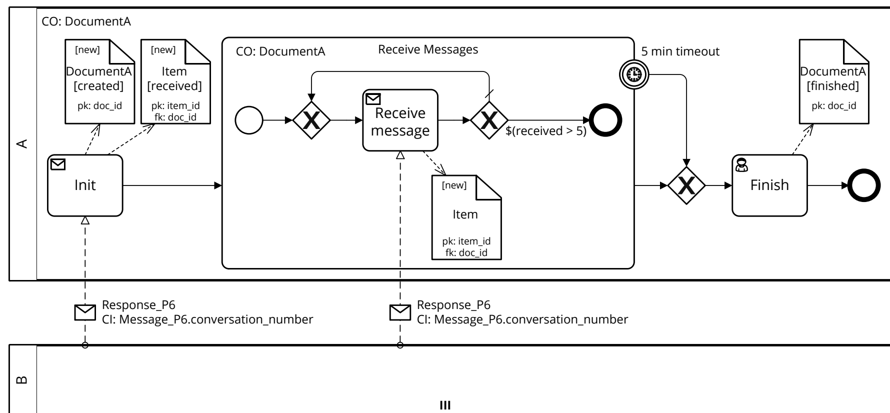


Figure 131: Pattern P6: one-from-many receive (create instance).

in Figure 130 realizes P6 for instances of A that are already running. The process uses *DocumentA* as case object. The subprocess is used to receive multiple messages containing a global object *Message_P6*. The contents of this global object is transformed to the local object *Item*; the correlation information *Message_P6.conversation_number* is mapped to the attribute *DocumentA.number* of the case object. For each received message, a new instance of *Item* is created. The first received message will initialize the correlation key *DocumentA.number* for the entire process. All subsequent messages that have the same key will be correlated to that instance. The subprocess has two termination criteria: receiving a certain number of messages and a timeout condition. The criterion on the received number of messages had to be implemented manually (by counting up a local variable *$received* in an execution listener of task *receive message*).

The model in Figure 131 realizes P6 for the situation when a new instance of A has to be created to receive the messages. Both receive activities can receive the same kind of messages. The first incoming message will be consumed by the instantiating receive task which also sets the correlation information. All subsequent messages that have the same correlation information will be routed to that instance. A mes-

sage with a different correlation information causes the creation of a new process instance.

Method. To correlate all messages with the same correlation key k to the same instance of A, k has to be mapped to an attribute of the case object of A; i. e., *DocumentA.number*. Technically, also a child object in a 1:1-relation to the case object will work. The semantics of our data annotations are in line with the correlation handling mechanism: receiving a new message technically always updates the local correlation key. However, as the correlation mechanism ensures correlation of messages only when keys match (or are uninitialized), the update does not change the key. Also, not every subprocess creates a new correlation key: by reusing the case object of the parent scope, the receive task in a subprocess inherits the correlation keys of the parent scope; i. e., the receive task in Figure 130 and Figure 131). This allows to use subprocesses for event handling without influencing the correlation mechanism.

MULTI-TRANSMISSION INTERACTION PATTERNS.
Patterns in this category describe scenarios where participant A directly exchanges multiple messages with one or more participants B1, B2, ...

P8 Multi-Responses. In P8, participant A sends a request to participant B and then receives one or more responses from B until a certain condition (based on received data or a timeout) holds. The challenge is to correlate each response of B to the instance of A that sent the request.

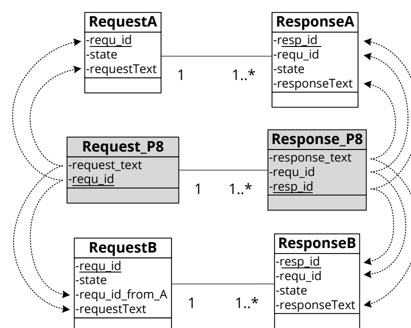


Figure 132: Data model for pattern P8.

Figure 132 shows the data model for this choreography which differs from the one in Figure 120 by allowing multiple responses per request (1:n relationship). Figures 133 and 134 realize pattern P8 for participants A and B respectively. A creates a global *Request_P8* object from their local *RequestA* object; the primary key *RequestA.requ_id* serves as correlation key.

The message is received by B (Figure 134) which can generate one or more responses in a loop. Each response carries again *RequestA.requ_id* as correlation identifier; its value is retrieved from the local object *RequestB* that was created when receiving *Request_P8*. When the response

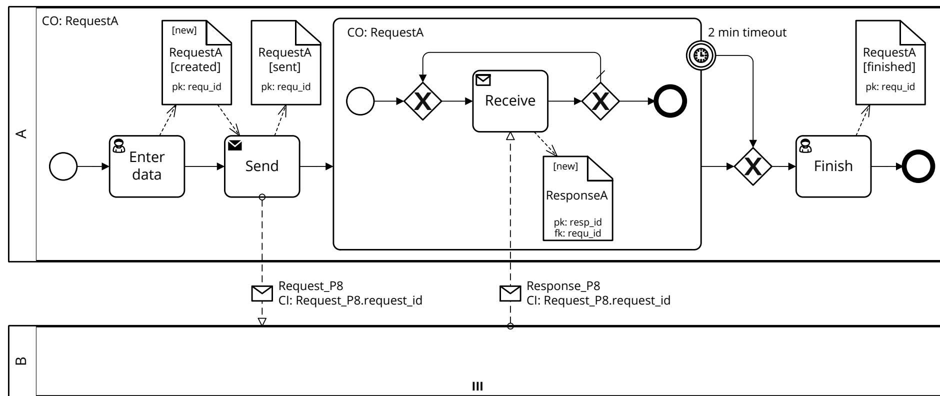


Figure 133: Pattern P8: multi-responses (participant A).

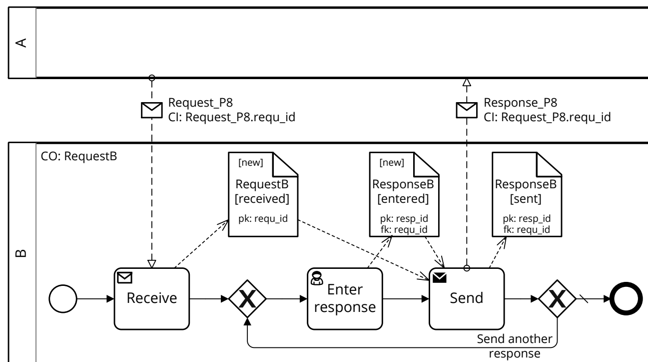


Figure 134: Pattern P8: multi-responses (participant B).

arrives at A, the correlation key only matches the correlation information of the sending instance that receives multiple messages until a timeout occurs (or an upper bound of messages has been received). Note that the upper bound of messages is not derived from the model shown in Figure 133, but implemented manually.

In general, P8 allows that B may respond with different message types. This can be achieved for B by replacing in the model of Figure 134 the send activity with a block of alternative send activities (a pair of XOR-gateways enclosing one send activity for each message type). The XOR-gateway chooses the corresponding type by following a specific path based on the kind of information entered. For A, replace in Figure 133 the receive activity with an event-based gateway followed by an intermediate message event or receive activity for each message type as shown in Figure 125 and Figure 126.

Method. From a correlation perspective, receiving multiple responses is identical to receiving a single response (as in P3): there is one correlation key to be set by the instance of A that B has to use for each response.

From a data perspective, one request has multiple responses, so there is a 1:n relationship from requests to responses in the data model; the recipient A stores each answer in a *new* local response object. If pro-

cess A wishes to store only the latest response, the local request and response objects of A should be in 1:1 relationship and the receive task only updates the existing response object (create an empty response object before receiving the first message).

The local subprocess of Figure 133 is only needed for scoping the timeout event and uses the same case object as the parent scope. B has no subprocess as its termination criterion is not event-based.

P9 Contingent Requests. In P9, participant A sends a request to participant B who shall send a response. If B's response does not arrive on time, then A will resend the request to another participant C, now expecting a response from C and discarding any response from B. If C's response does not arrive on time, the pattern is iterated with another participant D and so on until some response is received.

Figure 135 shows the data model for this pattern. It differs from the one in Figure 127 by having only 1 response for potentially n subrequests.

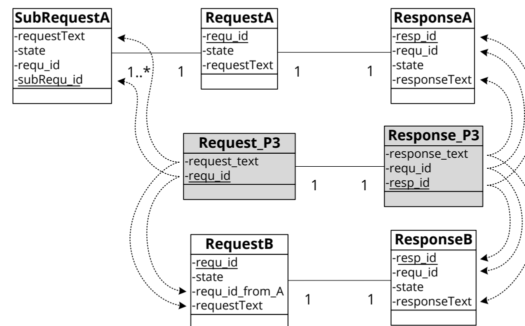


Figure 135: Data model for pattern P9.

The challenges in P9 are (1) to pick a different recipient each time a request is sent and (2) to ensure that at any point in time only the response to the latest request is considered as valid. The model in Figure 136 realizes this pattern as follows: Regarding (1), activity *Pick Recipient* stores the recipient's endpoint URL in a process variable; this variable is read when sending a message. Regarding (2), the case object *RequestA* has a single child object *SubRequestA* that is the case object of the subprocess. *SubRequestA* is mapped to the global object *Request_P3* and the primary key *subRequ_id* is mapped to the correlation identifier *Request_P3.requ_id*. The response *Response_P3* uses the same correlation identifier. For instance, the process of Figure 124 can receive the request and send a corresponding response.

As *SubRequestA* is the case object of the subprocess, the correlation key is only valid for the instance of the subprocess from which the message was sent. When the timeout occurs, the instance terminates and all its correlation information is removed. Then, the *SubRequestA* instance for this request is deleted by task *clear subrequest* before creating a new one. This ensures that at any point in time *RequestA* has a unique child *Sub-*

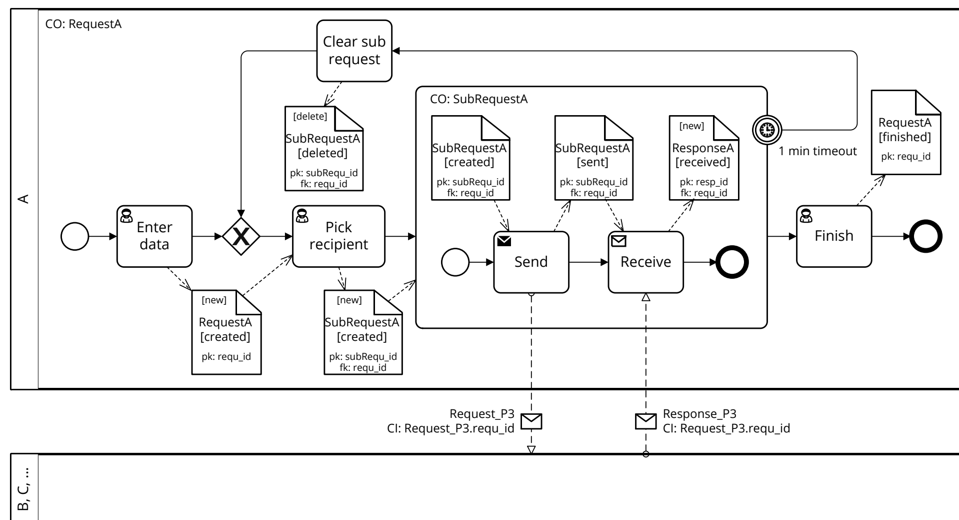


Figure 136: Pattern P9: contingent requests (participant A).

RequestA. The new child is used in the next iteration of the send/receive until a response arrives. As the correlation information is attached to the subprocess, only responses arriving during the lifetime of the sending subprocess instance will be correlated to the process.

Method. This pattern shows how one can *invalidate a correlation key* by storing the key locally in a child object (i. e., *SubRequestA.subRequ_id*) and deleting that child object when the correlation key is no longer to be used, i. e., when the timeout event occurred.

Making that child object the case object of the subprocess is good design practice. The object holding the correlation key is accessible for all send and receive tasks; all objects created or received in that subprocess can naturally be related to the subprocesses' case object via their foreign keys. Yet, it is possible to "jump out" of this subprocess: in Figure 136, the received *Response_P3* is stored as child object of *RequestA* which is the case object of the parent scope (ensured by the key relations of the data model of Figure 135). This pattern again shows a successful encapsulation of implementation details: participant B may implement the simple request/response pattern P3 not being aware of the complex correlation and data handling inside A.

P10 Atomic Multicast Notification. In P10, participant A sends a request to multiple participants B1, B2, ..., Bm; of these between n_{min} and n_{max} participants have to respond within a certain time interval. If less than n_{min} participants or more than n_{max} participants respond, then all participants of B1, B2, ..., Bm who already did respond have to be notified, e. g., by a cancellation message. In other words, this pattern has conditional transactional properties: from all the participants that do respond to A, either all continue successfully, or all are notified with a cancellation message. Which case occurs depends on the total number of responses received by A.

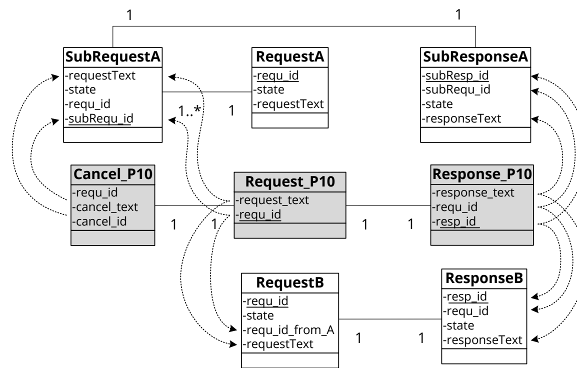


Figure 137: Data model for pattern P10.

Figure 137 shows the data model for this pattern which differs from Figure 127 by having a (sub-)response for every subrequest and by defining an additional global cancellation message.

The challenges in this pattern are to compute the number of received responses and depending on the outcome to either succeed or to notify all participants who did respond with a cancellation message.

The models in Figures 138 and 139 realize this pattern. In Figure 138, activity *Enter data* generates the local *RequestA* data object. The subsequent service task then generates multiple *SubRequestA* objects from the contents of *RequestA* (the corresponding handling of attributes has been implemented manually). For each *SubRequestA*, an instance of the first multi-instance subprocess is created in which that *SubRequestA* is transformed into a message carrying the global *Request_P10* object. The primary key *SubRequestA.requ_id* becomes the correlation key. When A receives a response for that correlation key, the *SubRequestA* object moves to state *received*; the content of the response is stored in the object *SubResponseA* being related to *SubRequestA*. When A did *not* receive a response until the timeout occurs, then task *Clear request* deletes the *SubRequestA* object for which there was no response. The multi-instance subprocess completes when for each *SubRequestA* either the response arrived (and hence *SubRequestA* is in state *received*) or the timeout occurred (and hence *SubRequestA* has been deleted).

Thus, when reaching task *Evaluate*, the case object *RequestA* of A has exactly one *SubRequestA* object instance for each received response. The task itself executes an SQL query to retrieve the number n of *SubRequestA* objects that are associated to the case object and sets the process variable *sendCancel* to *false* if and only if $n_{min} \leqslant n \leqslant n_{max}$. If *sendCancel* is *false*, the pattern terminates (or could be extended to interact with the responding partners). If *sendCancel* is *true*, the second multi-instance subprocess is started creating one instance for each *SubRequestA* object associated to the case object. Each subprocess instance carries the correlation key of a *SubRequestA* object for which a response has been received, i. e., *Request_P10.requ_id* which is mapped to *SubRequestA.subRequ_id*. This correlation key is used in the cancellation mes-

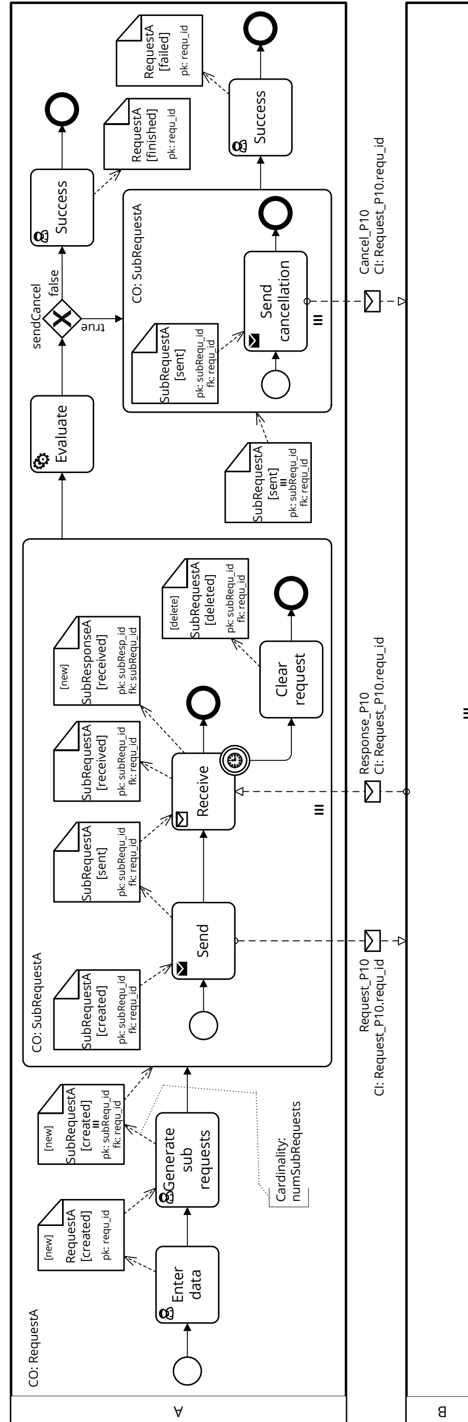


Figure 138: Pattern P10: atomic multicast notification (participant A).

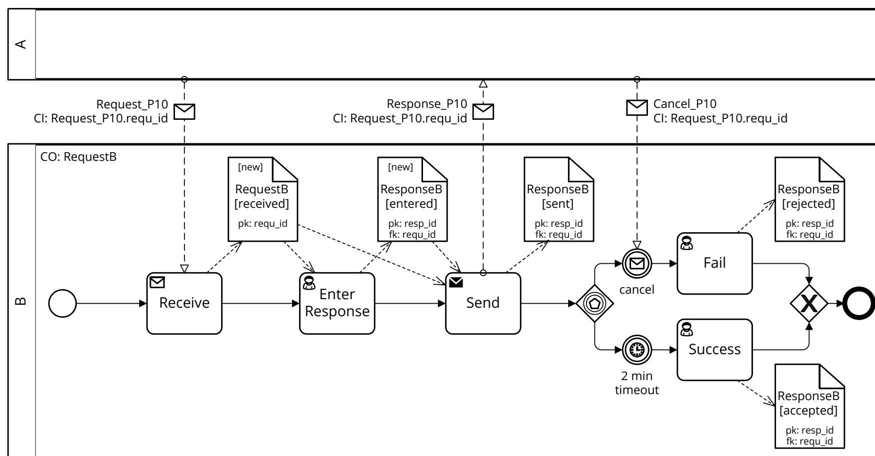


Figure 139: Pattern P10: atomic multicast notification (participant B).

sage so the receiving process B can correlate the cancellation message to exactly the instance that responded to the original request.

Participant B shown in Figure 139 generates a response for the request using the same correlation information as in the response. After that, B waits at the event-based gateway for either the cancellation message to arrive or a timeout to occur after which no cancellation message from A will arrive.

Note that Figures 138 and Figure 139 realize P10 using model-based concepts only, except for generating contents of the subrequests and for aggregating the number of received responses into a variable. These had to be defined manually.

Method. This pattern extends P9 and shows the strong expressive power of using child objects to store correlation keys. Here multiple keys are generated in parallel in the first subprocess. Depending on the timeliness of the response, some keys are preserved and some are deleted (a response arriving too late can no longer be correlated to that process instance).

The feature of our approach to create a specific number of subprocesses for a collection of multiple child objects allows to pause a set of conversations and simultaneously resume them at a later point in time: When the first subprocess of Figure 137 ends, all conversations are either suspended or terminated (depending on whether the *SubRequestA* object holding the correlation key still exists). Messages carrying these correlation keys can only be received by A while there is an running subprocess having the corresponding *SubRequestA* as case object (and listening for a specific message). In this way, some correlation keys can be temporarily muted and reactivated as long as there is a corresponding related object holding this key. When starting the second subprocess, the conversations involving these correlation keys resume.

ROUTING INTERACTION PATTERNS.

Patterns in this category describe scenarios where participant A sends messages to participants it does not know yet via an intermediate participant B; Participant B routes messages received from A to participants C,D,...

P11 Request with Referral. In P11, participant A sends to B a request that contains the address of a participant it would like to contact. Participant B takes the recipient information from this message and forwards the request to the right recipient. The challenges in this pattern are to forward a message to another recipient and to set the recipient's address from data in the message.

Figure 140 shows the data model for this pattern. The processes in Figures 141, 142, and 143 realize this pattern. In Figure 141, A generates the local object *RequestA* which also contains an attribute *endPoint* to which the request shall finally be routed; the local object *RequestA* is transformed into the global *AtoB_P11* which is sent to B.

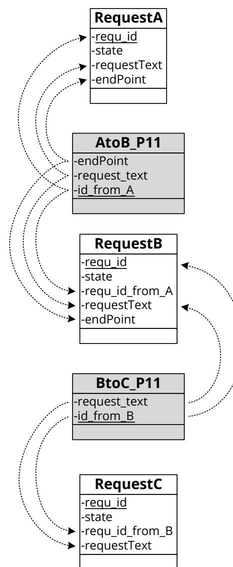


Figure 140: Data model for pattern P11.

In Figure 142, B receives the global object *AtoB_P11* and stores it in the local object *RequestB* including the attribute *endPoint*. The subsequent service task retrieves the value of *RequestB.endPoint* and stores it in a local variable *endPoint*. The subsequent send task generates the global object *BtoC_P11* from the stored *RequestB* and sends it to the URL in the variable *endPoint*. The process at that URL finally receives the request as shown in the process model in Figure 143.

The models can be extended to not only send a single endpoint URL but a list of URLs which B can then process one-by-one. Note that setting the process variable *endPoint* from the attribute of *RequestB* is not supported by our approach and had to be implemented manually.

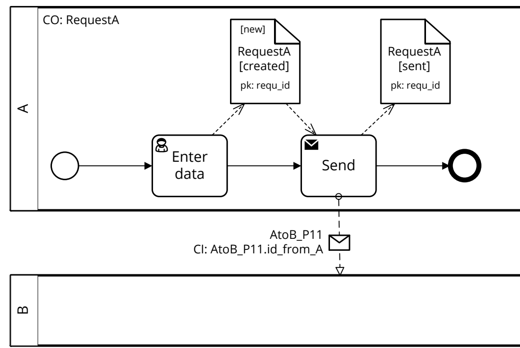


Figure 141: Pattern P11: request with referral (participant A).

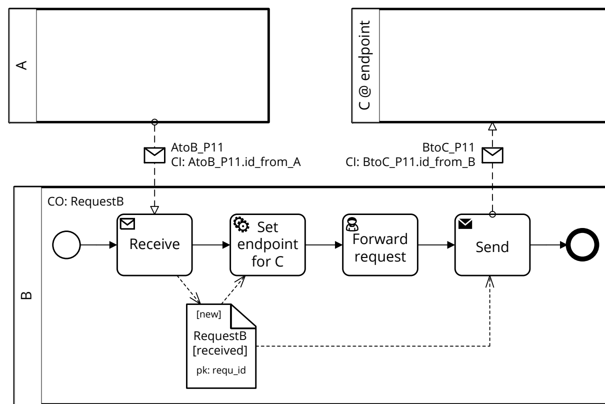


Figure 142: Pattern P11: request with referral (participant B).

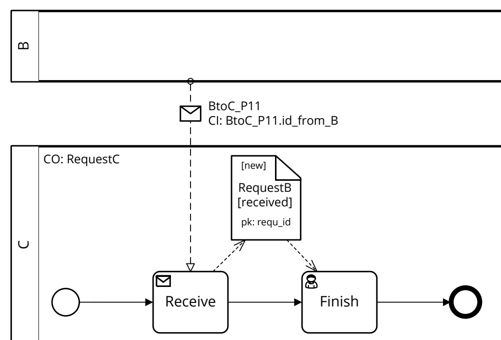


Figure 143: Pattern P11: request with referral (participant C).

Method. This pattern shows how to use multiple correlation keys between different participants. Each message exchange uses its own correlation key: *AtoB_P11* uses the primary key of A's local request object as key, allowing B to respond to this request. *BtoC_P11* uses the primary key of B's local request object which is different from the key used between A and B. C can only respond to the instance of B, not to A. This pattern demonstrates the usefulness of mapping correlation keys at the receiver side to normal attributes instead of primary keys. This way, the receiver is free to initialize a fresh correlation key, separating two different conversations.

P12 Relayed Request. In P12, participant A sends a request to B which is forwarded to C; the response of C is then sent directly to A and not via B. The challenge in this pattern is to provide C with the right correlation information so that the response from C can be correlated to the right instance of A.

Figure 144 shows the data model for this pattern. Here, participants A, B, and C first agreed that the message from A to B carries correlation information to identify the sending instance of A (via correlation key *AtoB_P12.id_from_A*). This correlation information is included in the message sent from B to C so that C can use the information in the final response *CtoA_P12*. The control and message flow are then straightforward as shown in Figures 145, 146, and 147 that realize this pattern.

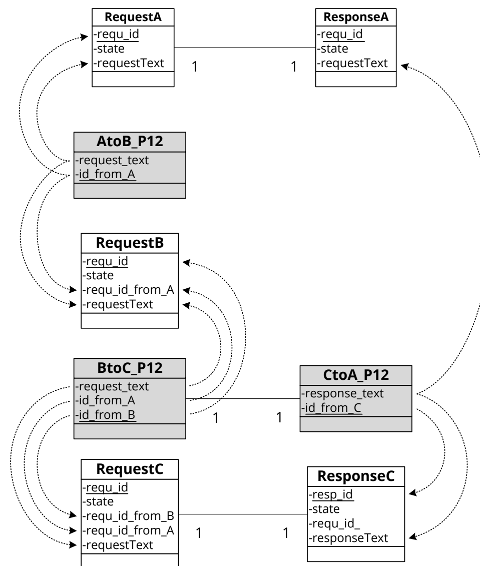


Figure 144: Data model for pattern P12.

In Figure 145, participant A sends the request with the correlation key *AtoB_P12.id_from_A* that has been set from the local attribute *RequestA.requ_id* and expects a response message *CtoA_P12* having the same correlation information. In other words, the value of *AtoB_P12.id_from_A* in *CtoA_P12* has to match the attribute *requ_id* of the case object *RequestA*.

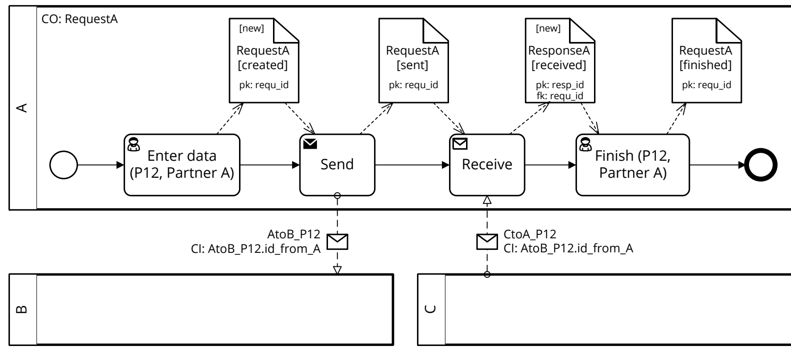


Figure 145: Pattern P12: relayed request (participant A).

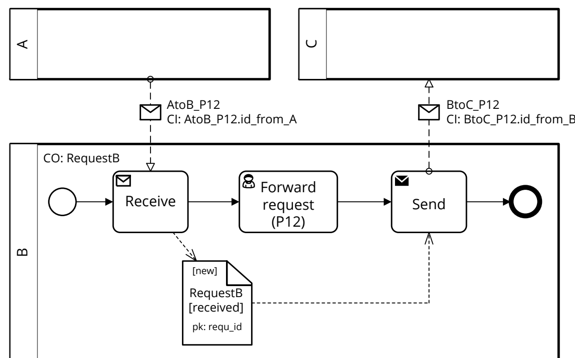


Figure 146: Pattern P12: relayed request (participant B).

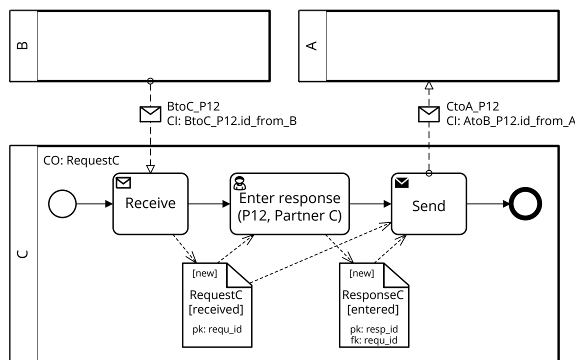


Figure 147: Pattern P12: relayed request (participant C).

In Figure 142, participant B receives the request from A and stores it in the local *RequestB*; in particular, attribute *AtoB_P12.id_from_A* is stored in the attribute *RequestB.id_from_A*. The entire request is sent to C in the global object *BtoC_P12* which has the attributes *id_from_A* (mapped from *RequestB.id_from_A*) and *id_from_B* (mapped from *RequestB.requ_id*). Thus, the correlation information of A is forwarded to C although the message *BtoC_P12* only carries *BtoC_P12.id_from_B* as correlation key.

In Figure 147, participant C receives the forwarded request from B and stores it in the local object *RequestC*; the attribute *BtoC_P12.id_from_A* is mapped to *RequestC.id_from_A*. Then C generates a response which is transformed to the global object *CtoA_P12*. The message also gets the correlation key *AtoB_P12.id_from_a* for which the value is mapped from *RequestC.idFromA*. Thus the final response contains the expected correlation information for the sending instance.

Method. In P11, we showed that correlation keys can be stored in regular attributes of local data objects. This pattern P12 shows that correlation keys can also be stored and forwarded in regular attributes of messages; i. e., attribute *id_from_A* in message *BtoC*). This way, correlation identifiers of messages can be set from such regular attributes as long as a valid mapping from local to global attributes is present. The correlation key used in a message is not necessarily an attribute of the message's object; i. e., *CtoA_P12* uses the key *AtoB_P12.id_from_A* that is not part of the message. Technically, correlation keys are globally identified by their object-attribute name and are added to the message (regardless of its contents). However, it is good practice upon initializing a correlation key to send both, correlation key and a data object holding the key. This allows the recipient to properly initialize the key through storage in a local object, e. g., participant B when receiving *AtoB_P12*).

P13 Dynamic Routing. In P13, participant A sends a request to B which forwards the request to participant C or D depending on the content of the message. This pattern differs from P11 in that C and D are known to B at design time and that the condition to whom to send the message is defined within B and not by A. The challenge here is to make an internal choice in B based on the content of a received message.

Figure 148 shows the data model of this pattern. The models in Figures 149, 150, and 151 realize this pattern. In Figure 149, participant A generates a *RequestA* object which is transformed into the global *AtoB_P13* object and then sent to B.

Participant B of Figure 150 receives the message *AtoB_P13* and transforms it into the local *RequestB* object which has an attribute *requestText* (mapped from *AtoB_P13.request_text*). The subsequent user task does not do anything but showing that the message has been received. The XOR gateway's outgoing arcs are annotated with a condition comparing attribute *requestText* of the case object *RequestB* to a value. Depending on

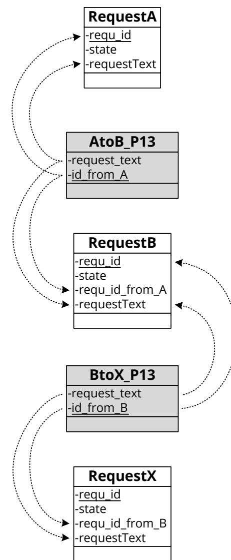


Figure 148: Data model for pattern P13.

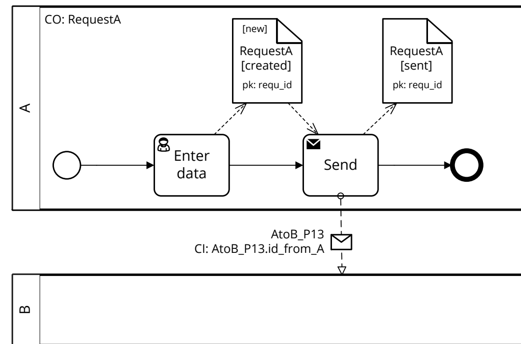


Figure 149: Pattern P13: dynamic routing (participant A).

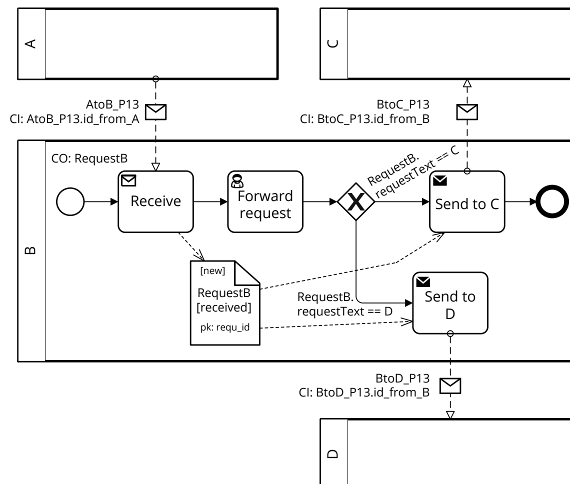


Figure 150: Pattern P13: dynamic routing (participant B).

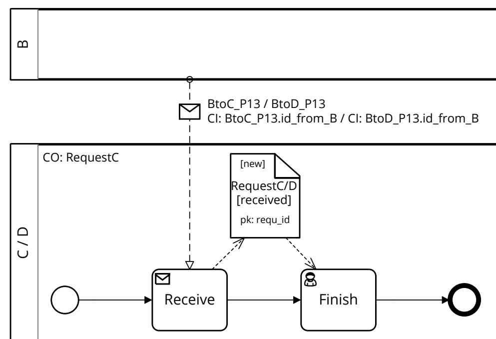


Figure 151: Pattern P13: dynamic Routing (participants C and D).

the value, a different path is taken leading to an activation of a different send task to which *RequestB* is forwarded.

This annotation at the XOR gateway matches pattern A2 in Section 8.2 and translates to the following behavior: when reaching the XOR gateway, an SQL query `SELECT requestText FROM RequestB WHERE requ_id = $ID` is generated (where `$ID` resolves to ID of the current process instance). The query retrieves the value of the attribute mentioned at the outgoing arc(s) in order to evaluate the guard expressions. Technically, we store the result of the query in a local process variable which is then used to evaluate the expression.

Then B forwards the request to either C or D (depending on the evaluation of the expression at the XOR gateway) and C or D receives the request as shown in Figure 151.

Method. The pattern reuses concepts shown earlier; we have shown that attributes of local data objects can be used in expressions at XOR gateways.

8.7 RELATED WORK

In the following, we first compare the concepts introduced in this chapter to other techniques for modeling and executing processes with data. Thereby, we focus on techniques with a graphical representation of the model. Our comparison includes all requirements for "object-aware process management" described in [173], two requirements targeting process choreographies on an abstract level, and three additional meta-factors. Second, we detail the view on process choreographies focusing on the communication between distributed partners, the transformation of data, and message correlation.

Comparison requirements

Altogether, the 19 requirements cover modeling and enactment of *data*, *processes*, and *activities* as well as authorization of *users* and support for *flexible* processes. (1) Data should be managed in terms of a data model defining object types (i. e., data classes), attributes, and relations; (2) cardinality constraints should restrict relations; (3) users can only read/write data they are authorized to access; (4) users can access

data not only during process execution; (5) content of messages should be specified for data exchange between different participants. Processes manage (6) the life cycle of data classes and (7) the interaction of different data objects; (8) processes are only executed by authorized users and (9) users see which task they may or have to execute in the form of a task list; (10) user may also communicate across organizational borders represented by process choreographies; (11) enabling sequencing of activities independently from the data flow. (12) One can define proper pre- and postconditions for service activities based on data classes and their attributes; (13) forms for user-interaction activities can be generated from the data dependencies; (14) activities can have a variable granularity with regards to data updates, i. e., an activity may read/write objects in 1:1, 1:n, and m:n fashion. (15) Whether a user is authorized to execute a task should depend on the role and on the authorization for the data this task accesses. (16) Flexible processes benefit from data integration in various ways, e. g., tasks that set mandatory data are scheduled when required, tasks can be re-executed.

In addition to these requirements, we consider *meta-factors* that influence the adaption of a technique, namely, (17) whether the process paradigm is activity-centric or object-centric, (18) whether the approach is backed by standards, and (19) to which extent it can reuse existing methods and tools for modeling, execution, simulation, and analysis.

Comparison

Classical activity-centric techniques such as *workflows* [340] lack a proper integration of data. Purely data-based approaches such as *active database systems* [305] allow updating data based on event-condition-action rules, but lack a genuine process perspective. Many approaches combine activity-centric process models with object life cycles, but are largely confined to 1:1 relationships between a process instance and the data objects it can handle, e. g., [97, 186, 363] and also BPMN [243]; some of these techniques allow flexible process execution [270].

Table 17 summarizes our comparison by showing how various techniques satisfy the above introduced requirements. We compare techniques that support at least a basic notion of data integration, i. e., techniques that consider both control flow and data aspects. *Proclets* [344] define object life cycles in an activity-centric way that interact through channels. In [356], the Product-based Workflow Support (PBWS), process execution and object interaction are derived from a product data model. *CorePro* [231], the *Object-Process Methodology* [81], *Object-Centric Process Modeling* [268], and the *Business Artifacts* approach [54, 237] define processes in terms of OLCs with various kinds of object interaction. Only artifacts support all notions of variable granularity (14), though it is given in a declarative form that cannot always be realized [60]. In *Case Handling* [347], process execution follows updating data such that particular goals are reached in a flexible manner. *PHILharmonic Flows* [173] is the most advanced proposal addressing variable granularity as well as flexible process execution through a combination of

Table 17: Comparison of data-aware process modeling and execution techniques with graphical model representation.

	requirement [in [173]]	Proclets [344]	CorePro [231]	OPM [81]	Obj.-Cent. [268]	PBWS [356]	Artifacts [54]	CH [347]	BPMN [243]	PH.Fl. [173]	**this**
data	1: data integration [R1]	o	o	o	o	o	+	o	-	+	**+**
	2: cardinalities [R2]	+	o	+	+	-	+	o	o	+	**+**
	3: data authorization [R10]	-	o	-	-	-	-	o	-	+	**-**
	4: data-oriented view [R8]	-	o	-	-	-	o	o	-	+	**o**
	5: data exchange between different participants	-	-	-	-	-	-	-	-	-	**+**
process	6: object behavior [R4]	o	+	+	+	-	o	o	o	+	**+**
	7: object interactions [R5]	+	+	+	+	o	o	o	o	+	**+**
	8: process authorization [R9]	+	+	+	+	+	o	+	o	+	**o**
	9: process-oriented view [R7]	+	+	+	+	+	+	+	+	+	**+**
	10: process choreography support	+	-	-	-	-	-	-	+	-	**+**
	11: explicit sequencing of activities	+	o	o	o	-	-	-	+	o	**+**
activity	12: service calls based on data [R14]	+	+	+	+	+	+	o	o	+	**+**
	13: forms based on data/flow in forms [R15/R18]	-	-	-	-	-	o/-	+/-	-	+	**+**
	14: variable granularity 1:1/1:n/m:n [R17]	-	-	-	-	-	o	o	-	o	**+**
users	15: authorization by data and roles [R11/R12]	-	-	-	-	-	-	-	-	+	**-**
flex	16: flexible execution [R3/R6/R13/R16/R19]	-	o	-	-	o	o	o	-	+	**-**
factors	17: process paradigm	A	D	D	D	D	D	D	A	D	**A**
	18: standards	o	o	o	o	-	-	o	+	-	**+**
	19: reusability of existing techniques	+	-	o	-	-	-	-	+	-	**+**

fully satisfied (+), partially satisfied (o), not satisfied (-), activity-centric (A), object-centric (D)

micro processes (object life cycles) and *macro processes* (object interactions); though variable granularity is not fully supported for service tasks and each activity must be coupled to changes in a data object (limits activity sequencing). More importantly, the focus on an object-centric approach limits the reusability of existing techniques and standards for modeling, execution, and analysis.

The novel techniques introduced in Sections 8.1 to 8.4 extend common activity-centric process description languages with data integration. Cardinalities can be set statically in the data model and dynamically as shown in the modeling example starting on page 205; a data-oriented view is available by the use of relational databases and SQL. Object behavior and their interactions are managed with variable granularity. Forms can be generated from the given data specification (cf. Section 8.3 and communication between different participants including data exchange (cf. Section 8.4) is established. Our work did not

Table 18: Comparison of concepts introduced in this chapter (i) against the requirements from [173] with our additions and (ii) with respect to their relation to the orchestration execution requirements (OER), process data requirements (PDR), and choreography execution requirements (CER).

	requirement [in [173]]	this	refers to OER, PDR, and CER
data	1: data integration [R1]	+	OER-2, PDR-1, PDR-2, PDR-4, CER-2
	2: cardinalities [R2]	+	
	3: data authorization [R10]	-	
	4: data-oriented view [R8]	o	
	5: data exchange between different participants	+	CER-1, CER-3, CER-5, CER-6
process	6: object behavior [R4]	+	OER-3
	7: object interactions [R5]	+	OER-4
	8: process authorization [R9]	o	
	9: process-oriented view [R7]	+	
	10: process choreography support	+	CER-1, CER-6
	11: explicit sequencing of activities	+	
activity	12: service calls based on data [R14]	+	OER-2
	13: forms based on data/flow in forms [R15/R18]	+	PDR-3
	14: variable granularity 1:1/1:n/m:n [R17]	+	OER-5
users	15: authorization by data and roles [R11/R12]	-	
flex	16: flexible execution [R3/R6/R13/R16/R19]	-	
factors	17: process paradigm	A	OER-1
	18: standards	+	OER-1
	19: reusability of existing techniques	+	

fully satisfied (+), partially satisfied (o), not satisfied (-), activity-centric (A)

focus on authorization aspects but this aspect can clearly be addressed in future work. Our approach builds on structured processes and thus, process flexibility was also not in the focus of this work. However, there exist attempts to make structured processes dealing with flexibility, e. g., Adept [270] and Production Case Management (PCM) [222]. PCM utilizes BPMN. Since we showed in this chapter how to apply the newly introduced concepts to BPMN, these concepts can also be combined with the PCM approach introducing flexibility. Independently from this, our work should primarily be applied in use cases requiring structured processes. Most importantly, we combine existing process description languages with industry standards for processes, data, and data exchange allowing to leverage on various techniques for modeling and analysis. We explicitly showed the combination with BPMN, the industry standard for process modeling. We demonstrated reusability by our implementation extending an existing process engine.

Requirement support

Table 18 relates the given requirements and the rating for our concepts to the orchestration execution requirements (OER), process data

requirements (PDR), and choreography execution requirements (CER) given in Sections 8.1 to 8.4. This overview shows that we support all defined requirements except for requirement CER-4 (message routing) that we put of scope and assume to be handled by standard technology of some underlying transport layer (e. g., web services). Requirement PDR-4 is only supported partly since we allow form generation and assume some data attribute specification but do not specify the exact technology to be used. The comparison also shows that our concepts cover more than the requirements raised in the corresponding sections.

Process choreographies

Detailing the view on process choreographies for activity-centric process models, we briefly discuss approaches being related to the presented concepts while focusing on the communication between distributed partners, the transformation of data, and message correlation. The service interaction patterns discussed in [19] describe a set of recurrent process choreography scenarios occurring in industry. Thus, they are a major source to validate choreography support of a modeling language. Besides BPMN [243] as used in this paper as example for the generic concept, there exist multiple solutions to cope with process choreographies. Most prominent are BPMN4Chor [66], Let's Dance [386], BPEL4Chor [70], and WS-CDL [379]. From these, only BPEL4Chor and WS-CDL realize operational semantics to handle message exchange by reusing respectively adapting the concepts defined in BPEL [240]. Though, message transformation to achieve interoperability between multiple participants is done with imperative constructs, i. e., the process engineer has to manually write these transformations and has to ensure their correctness. Additionally, BPEL4Chor and WS-CDL are not model-driven as our approach is.

In the area of business process management (BPM), there exist fundamental works describing the implementation of process choreographies [69, 350] with [350] ensuring correctness for inter-process communication. These works only describe the control flow side although the data part is equally important as messages contain the actual artifacts exchanged. [163] introduces a data-aware collaboration approach including formal correctness criteria. They define the data exchange using data-aware interaction nets, a proprietary notation exceeding the generic model specifications of this thesis, instead of, for instance, a widely accepted standard such as BPMN, the industry standard for process modeling.

Apart from process and service domains, distributed systems [318] describe the communication between IT systems via pre-specified interfaces similar to the global contract discussed Section 8.4. Usually, the corresponding data management is done by distributed databases [249] and their enhancements to data integration systems [181, 322] as well as parallel database systems [76] or is done by peer-to-peer systems [123, 329]. The database solution allows many participants to share data by working with a global schema which hides the local databases, but

unlike our approach, the participants work on the same database or some replication of it. Peer-to-peer systems take the database systems to a decentralized level and include mechanisms to deal with very dynamic situations as participants change rapidly. In process choreographies, the participants are known and predefined such that a centralized solution as presented in this chapter saves overhead since, in the worst case, the decentralized approach requires a schema mapping for each communication between two participants instead of only one mapping per participant to the global schema. The transformation of data between two participants can be achieved via schema mapping and matching [265, 303], a mediator [373], an adapter [383], or ontology-based integration [33, 239, 360]. For instance, [33] utilizes OWL [378] ontologies, which are similar to our global data model, and mappings from port types to attributes via XPath expressions to transform data between web services. We utilize schema matching due to the close integration of database support for data persistence.

Generally, the idea of generating code from models arises in the fields of model-driven architectures (MDA) [160, 241] and model-driven engineering (MDE) [158] respectively. We adopted this idea and generate SQL queries from process model information only. Additionally, we also generate messages from model information to allow automated data exchange between process participants.

8.8 Conclusion

We introduced an approach to execute a business process entirely model-driven utilizing activity-centric process models. While control flow execution works from existing approaches, the data side is entirely new. We incorporated complex data dependencies, even m:n relationships, with classical activity-centric modeling techniques for checking data existence upon activity enablement and ensuring data existence upon activity termination; the latter by initiating persistence. Based on these data dependencies, we introduced means to handle all process data by allowing retrieval and storage of required data represented through data attributes defined for data classes and data nodes and the generation of forms for user interaction. This covers the model-driven execution of process orchestrations. Since processes do usually communicate with other processes across organizational borders, we also provided a technique to automatically generate and correlate the messages exchanged during such communication.

Achieving our goal of entirely model-driven execution of the data perspective, we combined different proven modeling techniques: the idea of object life cycles, standard process description languages (e. g., BPMN), and relational data modeling together make classical activity-centric process models data-aware. The required information was put into the model by few extensions. (1) We utilize primary and foreign

keys for handling the data dependencies and distinguishing objects. (2) We utilize the concept of case object for correlating data objects to process instances. (3) We utilize expressions on data flow edges for cardinality specification. (4) We utilize an attribute mapping between global and local data models for message generation; indeed, any other mapping ensuring that each global attribute is mapped to some local attribute can also be utilized. (5) We utilize correlation identifiers, a list of fully qualified data attributes, annotated to the message flow to correlate process instances and messages. (6) We require specification of exchanged data objects annotated to the message flow.

We presented a set of 46 patterns covering all combinations of data operations (CRUD) on different types of data objects (case object, dependent object). A pattern describe how we generate SQL queries which then are used for accessing the data dynamically at run-time. These patterns exist in two versions for ensuring the data dependencies (check existence of data objects only considering the keys) and for handling all process data. For the communication between multiple process participants, we introduced a methodology to specify required information based on challenges of data heterogeneity, correlation, and 1:n communication. The partners agree on a global contract consisting of a data exchange format (data model) and a global collaboration diagram. Based thereon, each partner creates local counterparts of the process and data models.

Correctness of the process models and choreography models is mainly ensured by well known techniques. We presented an extensive discussion on these techniques and provided new techniques where required (consistency between global and local data models and correct message specification). Thereby, we reuse the notion of weak conformance for process orchestration correctness.

Reviewing the requirements stated in Sections 8.1, 8.3, and 8.4, most of them are fulfilled. Only requirement CER-4 (message routing between process participants) is not covered. It is out of scope of this chapter. Instead, we assume utilization of existing technologies as, for instance, web services. Our approach also provides additional functionality as discussed in Section 8.7 and shown in Table 18.

Our approach has been implemented by extending the camunda modeler and BPM platform. Both, data dependencies in process orchestrations as well as automated message exchange are completely covered. Thus, our implementation shows that an automated, entirely model-driven execution of the data perspective works. While completeness of orchestration handling is shown through consideration of all data operations and data object types, the choreography part required additional proof. Thus, we implemented the service interaction patterns [19] except for dynamically setting URLs of recipients and evaluating data conditions over aggregations of data objects; both are outside the scope of this chapter and deserve future work.

9 CONCLUSIONS

THE RESEARCH presented in this thesis is centered around the modeling, analysis, and execution of data in business processes answering the research question of HOW TO AUTOMATICALLY EXECUTE THE DATA PERSPECTIVE FROM ACTIVITY-DRIVEN PROCESS MODELS. Addressing this question, we introduced concepts basing on standard technologies allowing the generation of code for data access and data processing at run-time. The execution of the data perspective requires formal semantics steering process execution as well as sufficiently modeled and correct business processes. In this thesis, we provided *five contributions* targeting above research question. We summarize them in Section 9.1. Section 9.2 broadens the view and discusses the relevance of data with respect to some further fields of business process management (BPM). Finally, we discuss limitations and open problems with respect to the contributions of this thesis and outline future research directions including a brief discussion on the status of implementations for the concepts covered by this thesis's contributions in Section 9.3.

9.1 CONTRIBUTIONS OF THIS THESIS

Targeting the main research question of this thesis on the automatic execution of the data perspective, we identified several sub-research-questions defining the scope of the thesis. First, we identified the information required for automatic execution and described means to visualize them in the process model (cf. SRQ-1 and SRQ-2). The latter is required due to the goal of basing the execution on the process model only. Both have been achieved by few model extensions as, for instance, primary and foreign keys as well as instance and message correlation information in terms of a case object and correlation identifiers respectively. Since data specification in process models usually results in quite complex models, the stakeholder should be supported in adding the required information to the process model (cf. SRQ-3). Here, we allow the extraction of data nodes and their states from control flow information. Additionally, the stakeholder can get presented the current state of the process model in different views – represented following the activity-centric paradigm, represented following the object-centric paradigm, represented by a state-transition diagram referring to the object life cycles (OLCs) of data classes, and represented tailored or refined respectively with respect to stakeholders' needs. We achieved this by specifying appropriate model transformations (cf. SRQ-8). Modeling data refers to the design-time aspect. This must be correct to allow execution based on this information (cf. SRQ-7). We introduced the notion of weak conformance that can be computed by soundness checking integrating control flow and data flow correctness within a single check. For covering the run-time aspect, we defined formal semantics by a Petri net mapping that also shows the interplay of control flow and data (cf. SRQ-4). Finally, considering these contributions, we provided a technique to code from the correct process model to check for data existence, to retrieve and store data in databases, to generate forms for user interaction, to generate the message sent between multiple process participants, and to correlate data and messages to the correct process instances allowing the automatic execution of the data perspective (SRQ-4). Therefore, we used standard technologies (Structured Query Language (SQL) and XML Query Language (XQuery)) making the code generation quasi-platform-independent (cf. SRQ-6) contributing to our goal of generic concepts.

Summarized, we provided five major contributions that are briefly discussed next.

(1) Model-driven Business Process Execution

Targeting the entirely model-driven execution of the data perspective, we introduced few concepts to extend existing activity-centric process execution technology at well defined points. For process orchestrations, we introduced primary and foreign keys from the database domain into

process models to differentiate data objects at run-time, we borrowed the concept of one object driving the process execution – called case object – from the object-centric process modeling to allow correlation of data objects to process instances, and we utilize edge annotation to handle data cardinalities. For process choreographies, we introduced the specification of correlation identifiers and message content in terms of fully qualified attributes and data classes respectively. Overall, we require structural data information that we modeled by means of data models.

Utilizing these design-time concepts, we provided a set of 46 patterns covering all data operations (CRUD) for all types of data objects (case object, dependent object) to extract SQL queries handling complex data dependencies (even m:n relationships), data retrieval, and data storage. In addition, we introduced a methodology – adapting the Public-to-Private approach [342] – to get from a globally agreed contract with respect to data exchange to the actual message exchange automatically from model information only. Utilizing the patterns for data retrieval, required information to generate the message is available in the local representation that we then transform into the agreed global representation – the message. On the receiver side, the message is transformed from the global representation into the local one (which usually differs from the local one of the sender) and stored again using the introduced patterns.

To consistently interpret the modeled data information, we specified a formal semantics through a Petri net mapping that extends the existing one from Dijkman et al. [80]. Implementation shows the applicability of our entirely model-driven approach. For proofing completeness with respect to communication between different process participants, we also implemented all service interaction patterns [19] in our system. We showed that we can handle all of them with two small limitations: for the *atomic multicast* pattern, the m-out-of-n condition must be specified manually, and for the *request with referral* pattern, the endpoints URLs are defined through variables requiring a manual assignment from contents of the message.

(2) Formal Framework for Process and Data Integration

We formally introduced generic data concepts into traditional activity-centric process description languages through process scenarios, a formalism combining a process model and the corresponding synchronized object life cycle. The synchronized object life cycle contains of multiple single object life cycles, each referring to a single data class that is represented in the data model which in turn must be specified for a process scenario. The data model contains all data classes, their relations to each other and the data attributes representing a data class's structure. We provided a conceptual model to integrate all formal concepts. Targeting the operational semantics of these concepts,

we introduced a Petri net mapping for the data perspective of process orchestrations and for the message consideration in process choreographies.

(3) Data Flow Correctness

The notion of weak conformance enables checking for correctness between a process model and the synchronized object life cycle; i. e., data flow correctness of the process scenario. Weak refers to the capability of also checking abstracted or underspecified process models, where some data state transitions, i. e., data object manipulations, are given implicitly only. For computing the notion of weak conformance, we utilize soundness checking – a standard approach to check for behaviorally correctness. We map the process model and the synchronized object life cycle to Petri nets, combine them by matching places representing data states, and transform the resulting integrated Petri net into a workflow net by adding enabler and collector fragments as we call them. Since the workflow net comprises control flow and data information, we check for control flow correctness and data correctness by a single, integrated correctness check. Making the process scenarios executable, we highlight violations in both, the process model or the synchronized OLC, and provide correction proposals for data issues. Correcting control flow issues follow already existing works. Soundness checking and correction on Petri net level allows application of our approach to a multitude of process description languages due to existing mappings to Petri nets [190].

(4) Data Extraction from Control Flow

Supporting the process modeler in creating process models with data information, we introduced a generic approach to extract data nodes and their states from control flow information. We analyze the labels of activities to extract the actual objects being processed and their states after processing. Using this information, we specify output data nodes of activities. Considering these output data nodes and the partial ordering of control flow nodes, we specify input data nodes of activities. Additionally, we showed how to utilize concepts specific for a process description language to improve the quality of the extraction results.

Validating the results of our extraction approach, we performed an empirical study consisting of three experiments. Therewith, we could show that application of our approach provides process models that can directly be used for empirical research or as basis for process automation, if the labels are clear, concrete, and aligned with the process structure. Especially the latter proofed relevant in the context of this thesis. In contrast, in case the activity labels are ambiguous, very detailed, or very generic, the resulting process models may act as starting point to annotate the process model with data nodes and their states by

providing insights about the manipulations performed by the activities. Here, much additional effort would be required to create executable process models. Tooling helps in increasing process model quality and as such the quality of the results of our extraction result.

(5) Model Transformations

For provisioning different views on business processes, we introduced a set of algorithms allowing inter-view and intra-view transformations. The former ones target view transitions, i. e., changing the underlying process description language or process modeling paradigm, and the latter ones target view adaptations, i. e., changing the abstraction level by preserving the utilized language. We allow the transition of activity-centric process models to object-centric process models and vice versa by utilizing a synchronized object life cycle acting as mediator between both types of views. Changing the type of view puts different aspects in focus allowing explicit discussions on this detail. Furthermore, we allow the refinement of an activity-centric process model by adding modeling concepts to comprise all data manipulations given by some object life cycle while preserving the initial structure of the process model, i. e., it is not derived from the OLC directly. As second view adaptation, we allow the tailoring of an object life cycle with respect to the data manipulations given in an activity-centric process model by again preserving the object life cycle's initial structure. Adapting a view helps in understanding what happens within a business process (object life cycle tailoring) or helps to create an executable process model (refinement).

9.2 RELEVANCE OF DATA IN BUSINESS PROCESS MANAGEMENT

Integrating the data and control flow perspectives impacts multiple fields within the BPM domain. While most fields usually abstract from data information, data can largely contribute in a multitude of ways. Thus, we introduced data to six fields of BPM showing its value for these fields exemplarily. In the remainder of this section, we briefly summarize our findings.

Event Processing

Event processing is part of the process mining and process monitoring domains and deals with the creation, evaluation, and processing of events that indicate happenings within business processes, e. g., start or termination of an activity, or that indicate external happenings influencing the process execution, e. g., resource unavailability or some delay. Besides mining process models based on occurred events to show the actual executions, one can also utilize the events to predict the current state of a running process instance, e. g., [280], and monitor the process

execution in real-time [16, 59, 118, 134, 199]. These aspects benefit from increasing event log information. Contributing in this direction and especially targeting real-time monitoring of the business processes, we introduced *object state transition events* representing the change of data states of objects of some class. Based on the specification of input and output data nodes in the process model and these object state transition events, we can deduce from these events the enablement and termination of activities without explicit monitoring of the corresponding activity. The details are described in [133, 135, 136].

Batch Processing

Batch processing is a proven technology to combine and synchronize multiple instances to improve resource utilization and to save execution costs through reduced set-up costs [47, 198, 236, 326]. Applying this technology in the field of business process management allows to synchronize different instances of process models [173, 185, 194, 259, 294]. Without data consideration, it is hard to distinguish two process instances. For instance, considering our build-to-order and delivery process from Section 2.4, multiple orders (i. e., process instances) could be sent within a single package to reduce shipping costs. However, without information on the addressee for each order, one cannot decide which orders can be shipped together. Data allows to get this information. Here, each order has an associated customer; this information can be compared between multiple orders and the ones with identical information are synchronized by batch processing. To do so, we applied the concept of *instance data views* to process instances (also cf. Section 4.3) and batch processing as so-called *grouping characteristic*. For each batch, the stakeholer must specify the grouping characteristic based on which the process instances are distributed to corresponding batch clusters with each cluster only processing process instances with the same data specification – in above example, the same addressee. The details are described in [260, 261].

Business Process Architectures

Business process architectures (BPAs) describe the dependencies between multiple process models that may belong to multiple business processes [79, 86, 287]. Usually, the dependencies are specified by event flow, i. e., considering the control flow perspective. However, only considering the order of activities might induce dependencies between process models that do not occur at run-time; e. g., the control flow may indicate that some process model is executed after another one (payment process model and rejection notification process model after the checking order process model) but data specification restricts the flow towards either process model depending on the specific state of the corresponding data object. We introduced an approach to derive the

dependencies between process models from the data point of view (see [87]) that can be used as basis to combine the data view together with the event flow view providing an holistic process model dependency specification.

Business Process Model Abstraction

Business process model abstraction (BPMA) is a technique to provide different views with different pieces of information on the same business process. Referring to Section 1.3, views on process models can be provided on a horizontal and a vertical scale. In this thesis, we introduced algorithms for horizontal view creation while BPMA is mean to create views on the vertical scale. BPMA is well researched with respect to control flow abstraction, e. g., [96, 165, 255, 256, 308, 309], including the consideration of data information to abstract the control flow [309]. The abstraction of the data perspective itself is out of scope in these works. Targeting this shortcoming, we incorporated data into BPMA in two ways. First, we generically extended existing abstraction techniques that aggregate connected fragments of process models into single activities with means to abstract the data annotations as well [209]. Second, we introduced a set of abstraction criteria – similar to [307] for the abstraction of control flow – to abstract data nodes independently from control flow structures [131].

Flexible Business Processes

Flexibility is a core concern in business process management since real-world business processes need to be *elastic*, i. e., adaptable, in some way. Organizations must be able to adhere their business processes and the corresponding process models with respect to happenings during process execution, e. g., exception handling. However, not only exceptions induce changes to the business since different employees might perform the same work differently still leading to the same result. This difference may also arise from context information such as, for instance, the customer type with respect to our build-to-order and delivery scenario. In recent years, many research tackled process flexibility from different point of views [99, 124, 270, 285, 344, 347, 349, 365] with most focusing on control flow aspects and neglecting data or vice versa.

Similar to the oclet approach [99, 100], we distribute the process logic over multiple process components that are newly combined during process execution. This also allows to add new process components at run-time depending on business needs. The combination of multiple process components is driven through data dependencies – availability of data objects during process execution determines which components are combined in what way following the idea of *Production Case Management (PCM)* initially raised by Swenson [317]. In PCM, a process participant gets proposed a set of activities to be executed next from

which the participant then chooses one she thinks most suitable for the current process state. This allows some degree of flexibility but also guidance of the process participant through the execution enforcing defined constraints. The implementation framework consisting of the modeling methodology and the corresponding operational semantics to combine multiple components at run-time is described in [222].

9.3 LIMITATIONS & FUTURE RESEARCH

This thesis addressed the utilization of data in the business process management area with focusing on process modeling, analysis, and execution. Thus, considering the business process life cycle [346], this thesis addressed the design and analysis phase (modeling processes and ensuring their correctness) and the enactment phase (execution) while bypassing the configuration phase through deriving all execution-relevant data-information entirely from the process model. However, as indicated by the discussion in Section 9.2, data plays an important role in many more areas, since data provides a new perspective which may improve approach quality (e. g., process monitoring and process mining through additional event base) or helps fostering flexibility (e. g., PCM). These discussed areas as well as additional ones deserve intensive studying. For instance, data may be another factor in process model search or process model alignment to evaluate similarity or equivalence of processes or activities.

Input & Output sets

Regarding the core of this thesis, we applied some limitations which might be resolved by additional research. Most influential is the assumption on conjunctions and disjunctions of data nodes with respect to their data class[1]. This limitation forbids optional data nodes. For instance, recall the build-to-order and delivery scenario of this thesis. The fact that customs information are required to be entered on the package for specific shipments, e. g., overseas shipments, could not be represented in conjunction with a *Ship order* task since then – due to the different data class of the added data node – all packages would get this additional information. To resolve this limitation, a concept similar to Business Process Model and Notation (BPMN)'s input and output sets could be used. This would allow to specify all combinations of data nodes based on the ones given in the process model. The hard part is to find some good visualization with respect to this thesis's goal which executes processes entirely model-driven. Indeed, hiding some information in properties still allows them to be used for our purposes but this increases the modeling challenges.

The impact on this thesis's contributions would require adaptations to the Petri net mapping, the transformations, and the SQL derivation patterns. In the Petri net mapping (see Figure 40 on page 88), rules four

1 Data nodes referring to the same data class are considered as disjunctive while data nodes referring to different data classes are considered conjunctive.

to seven could be reused but they require different alignment – not with respect to data classes referenced by data nodes but with respect to the input or output sets specifying conjunctions and disjunctions. The transformation algorithms between a synchronized object life cycle and an activity-centric process model as well as the patterns need to be adapted analogously to ensure correct reading and writing of information with respect to the given input and output sets.

Multi-instance data

Additionally, the handling of multi-instance data objects requires research; especially the handling of potential infinity if no cardinality is specified. Since we rely on cardinalities, this part of research was considered out of scope for this thesis. In the context of multi-instancy, the need of sets with different states becomes apparent when looking into practice. Again, consider a build-to-order and delivery process where the customer orders multiple items from which some can be delivered and some not. Following the concepts in this thesis, either all items are delivered or none if such exception is not handled by explicit modeling – sending the respecting notifications might still work but succeeding, for instance, with the step *Archive order*, the order could be in state *"partially delivered"*. However, it is not clear which items have been delivered and which not such that archiving might take place on item level. This is cannot be realized so far since the respecting task may only read objects of a single data state. This limitation can be resolved by utilizing above mentioned input and output sets, if a single input or output condition may contain data nodes referring the same data class but having different data states, e. g., `{order.accepted, order.rejected, shipment.delivered}` with *order* being a multi-instance data node (object) such that objects of class order in state accepted or rejected and a delivered shipment referring to each other via foreign key relationships are required to execute the task.

In the same direction, we assume different objects of the same class to be manipulated by the same activity. There may exist multiple options for manipulating an object, but all must take the same path except if the manipulations take place on distinct paths of the corresponding object life cycle. Utilizing colored Petri nets for representing the formal semantics with token colors distinguishing objects based on the primary key concept introduced in this thesis. Alternatively, the Petri net mapping could consider all possibilities of data manipulation based on different creation points and create a set of paths for each such option. This may result in even more complex Petri nets. Based thereon, the formal verification could be extended to cover primary key distinction.

Events

The generic process model introduced in this thesis does not cover intermediate events and different types of events in general as known from, for instance, the industry standard in activity-driven process modeling: BPMN. These events are proper means to cover handle disruptions and exceptions, to link multiple events, or to react based on some input. Considering them might influence some of our concepts and

deserves additional research. With respect to send and receive events, they could complement the respecting activity types for communication between different process participants. Thereby, send and receive events from BPMN could not be adopted as they are, since BPMN does not allow transformation work within such event. Extending them in this direction would be an opportunity to widen the support of modeling concepts covered by the process model definition. However, we did neglect the event extension since they are not covered in many activity-driven process description languages which was a base requirement for our basic process model that got extended with data-specific concepts.

Transformations

The presented inter-view transformations provide behaviorally equivalent representations of the models of the business process. After applying a roundtrip from one object-centric process model via an activity-centric process model back to the object-centric process model may reveal a different structure with respect to the handling of tasks from object-centric process model. Two tasks of one business rule are summarized into a single one during the roundtrip. Starting from an activity-centric process model does not reveal such issues. Thus, future work could make these transformations isomorphic.

Additionally, the transformations assume manual label adjustments. These may also be derived by natural language processing to increase the automation of the transformation. Third, the current representation of a synchronized object life cycle does not support attribute information. Extending this formalism and adapting the algorithms producing a synchronized object life cycle would remove dependencies currently existing, since attribute information is retrieved from a view that is technically not involved in the transformation.

Implementations

Reviewing the contributions of this thesis, four approaches have been introduced from which two have been implemented as proof of concept. We showed how to execute a sufficiently data-annotated process model entirely model-based and we showed how to extract data information from control flow. For the former, we abstracted from user form generation. Implementations in additional project work[2] however show the applicability [143]. In contrast, the weak conformance correctness check and the model transformation lack an implementation. For the model transformations, the given algorithms could be implemented to allow their application in practice. The mapping from an activity-centric process model to a synchronized object life cycle and vice versa following the algorithms presented in this thesis are also shown in above mentioned project work [120]. For the correctness check, the Petri net mappings for the process model and the synchronized object life cycle followed by their integration must be implemented. For the actual check, existing tools, e. g., the LoLA model checker [301, 375], can be utilized by providing the process model and mapping the counterexamples on the required violation representations as discussed in Chapter 6.

2 https://bpt.hpi.uni-potsdam.de/Public/JEngineDoc/

Part IV

APPENDIX

A

INTER-VIEW TRANSFORMATION ALGORITHMS

Algorithm 6 Transformation of an object-centric process model $ocp = (AS, U, BR)$ into a synchronized object life cycle $\mathcal{L} = (L, SE)$

1: initialize synchronized object life cycle $\mathcal{L}$
2: **for all** data classes $c \in CS$ **do**
3: create object life cycle l
4: **end for**
5: **for all** business rules $br \in BR$ **do**
6: **for all** data classes $c \in CS$ **do**
7: determine set of precondition states from the $instate$ functions
8: determine set of postcondition states from the $instate$ functions
9: add not existing data states to the corresponding object life cycle $l = \eta^{-1}(c)$
10: **for all** tasks $u \in U$ utilizing data class c **do**
11: add task label of u to label lab by concatenation where lab indicates the action performed
12: **end for**
13: create a data state transition t from each precondition state to each postcondition state and label it lab
14: **end for**
15: **for all** pairs of added data state transitions for br **do**
16: add synchronization edge $se = (t_1, t_2)$ between each two transitions $t_1 \in \mathfrak{T}_{S,l_1}$ and $t_2 \in \mathfrak{T}_{S,l_2}$ of different data classes c_1 and c_2 with $\eta(c_1) = l_1$ and $\eta(c_2) = l_2$
17: **end for**
18: **if** state s of class c is used in the precondition br_{pre} of the business rule br and not in its postcondition br_{post} **then**
19: add synchronization edge $se = (s, s', currently)$ of type *currently* from data state s to each postcondition state s' of some other data class
20: **end if**
21: **end for**
22: **for all** object life cycles l **do**
23: the state without an incoming transition becomes the initial state s_i
24: states without an outgoing transition become a final state $s \in S_F$
25: add l to the synchronized object life cycle $\mathcal{L}$
26: **end for**

Algorithm 7 Transformation of a synchronized object life cycle $\mathcal{L} = (L, SE)$ into an activity-centric process model $pm = (N, D, Q, \mathfrak{C}, \mathfrak{F}, type_g, \mu, DCF)$ with $DCF = (\xi)$.

1: group all transitions $t \in \bigcup_{\mathcal{L}} \mathfrak{T}_{S,l}$ executed together into combined transitions CT
2: initialize activity-centric process model pm and add the start event to pm
3: **repeat**
4: **for all** nodes $n \in N_{pm}$ that have not been marked as checked for combined transitions **do**
5: determine all combined transitions $ct \in CT_n$ which are enabled after termination of node n
6: determine all combined transitions $ct' \in CT'_n$ which are enabled after termination of all currently enabled nodes
7: **for all** output data nodes in a final state **do**
8: add no operation activity labeled $nop : state$ without data associations to process model pm
9: add $nop : state$ as combined transition to the no operation set NOP_n
10: mark this activity as checked for combined transitions
11: **end for**
12: mark node n as checked for combined transitions
13: **end for**
14: **for all** combined transitions $\hat{ct} \in ((\bigcup_n CT_n) \cup (\bigcup_n CT'_n))$ **do**
15: create activity $a \in A_{pm}$
16: **for all** transitions t being combined in $\hat{ct}$ **do**
17: add as input to activity a a data node of the data class t belongs to with the state being the source of t
18: add as output to activity a a data node of the data class t belongs to with the state being the target of t
19: add/append the activity label with the action of transition t
20: **end for**
21: **end for**
22: **for all** nodes $n \in N_{pm}$ without incoming nor outgoing control flow edge **do**
23: **if** $|\tilde{ct}| = 1$ (number of enabled combined transitions in $(\bigcup_n CT_n) \cup (\bigcup_n CT'_n) \cup NOP_n)$ **then**
24: add control flow edge from n to activity a being created for the respective combined transition $\tilde{ct}$
25: **else if** $|\tilde{ct}| > 1$ **then**
26: **if** $|NOP_n| > 0$ **then**
27: add XOR gateway $g \in G_{pm}$ to pm
28: **else if** all created activities write data nodes of distinct classes **then**
29: add AND gateway $g \in G_{pm}$ to pm
30: **else**
31: add XOR gateway $g \in G_{pm}$ to pm
32: **end if**
33: add control flow edge from n to g
34: add control flow edges from g to each activity a being created for the respective combined transitions $\tilde{ct}$
35: **if** g is an XOR gateway **then**
36: **for all** control flow edges cf originating from g **do**
37: assign as data condition to cf the data node d that is input to the target of cf and that is output to the activity being the predecessor of g, the source of cf; if the target of cf is a placeholder activity, the state mentioned in the label (labeled placeholder activity) or the state of the output data node (unlabeled placeholder activity) is utilized
38: **end for**
39: **end if**
40: mark gateway g as checked for combined transitions
41: **end if**
42: **end for**
43: resolve multiple incoming edges to an activity by adding additional gateways and rerouting the control flow edges
44: **until** all nodes $n \in N_{pm}$ have been checked for enabled combined transitions

Algorithm 7 Transformation of a synchronized object life cycle $\mathcal{L} = (L, SE)$ into an activity-centric process model $pm = (N, D, Q, \mathfrak{C}, \mathfrak{F}, type_g, \mu, DCF)$ with $DCF = (\xi)$ (cont.).

45: **if** # nodes $n \in N_{pm}$ without outgoing control flow edge = 1 **then**
46: **if** node n contains a label of type $nop : state$ **then**
47: add end event $e \in E_{pm}$ to process model pm
48: reroute the control flow edge targeting n towards that end event e
49: remove n from the process model
50: **else**
51: add end event $e \in E_{pm}$ to process model pm
52: add control flow edge from n to e
53: **end if**
54: **else**
55: add XOR gateway $g \in G_{pm}$ to process model
56: add end event $e \in E_{pm}$ to process model
57: add control flow edge from g to e
58: **for all** nodes $n \in N_{pm}$ without outgoing control flow edges **do**
59: **if** node n contains a label of type $nop : state$ **then**
60: reroute the control flow edge targeting n towards g
61: remove n from the process model
62: **else**
63: add control flow edge from n to g
64: **end if**
65: **end for**
66: **end if**
67: change the source node of a control flow edge originating from an unlabeled activity to the XOR gateway directly preceding it
68: remove all unlabeled activities, the associated output data nodes, and the control flow edges targeting them

Algorithm 8 Transformation of an activity-centric process model $pm = (N, D, Q, \mathfrak{C}, \mathfrak{F}, type_g, \mu, DCF)$ with $DCF = (\xi)$ into a synchronized object life cycle $\mathcal{L} = (L, SE)$.

1: identify data classes c in process model pm
2: **for all** data class $c \in C$ **do**
3: initialize object life cycle $l = \eta^{-1}(c)$
4: identify distinct data states $s \in S_c$ and add them to corresponding sets S_l
5: **end for**
6: extract all traces σ_A of pm from control flow specification (loops are executed exactly once)
7: **for all** traces σ_A **do**
8: **for all** data classes c utilized on σ_A **do**
9: initialize collection K_c with c being the current data class to store data state s of each corresponding data node $d = \varphi_D^{-1}(c)$ and add the initial data state
10: **end for**
11: **repeat**
12: **if** node $n \in N$ is an activity **then**
13: get data states of all input data nodes d of n grouped by data class $c = \varphi_D(d)$
14: **for all** data classes c **do**
15: add the data states (if not existing yet) identified for c to object life cycle $l = \eta^{-1}(c)$
16: add one transition (if not existing yet) from each entry in collection K_c to each data state identified for c to object life cycle $l = \eta^{-1}(c)$ except source and target would be the same data state
17: add τ as action to each transition
18: replace the entries in K_c with data states of the input data nodes of the current activity
19: **end for**
20: add undirected synchronization edge between each two transitions, which have been added above and belong to different object life cycles l_1 and l_2
21: get data states of all input data nodes d of n grouped by data class $c = \varphi_D(d)$
22: **for all** data classes c **do**
23: **if** predecessor of node n is an XOR gateway $g \in G_{pm} \subseteq N_{pm}$ **then**
24: **if** data class c is part of the data condition $dc = \xi(g, h)$ with $h \in N$ being the direct successor of g in trace σ_A **then**
25: remove all data states from the result for c that do not match the state of data node d used as data condition dc
26: **end if**
27: **end if**
28: add the remaining data states (if not existing yet) identified for c to object life cycle $l = \eta^{-1}(c)$
29: add one transition (if not existing yet) from each entry in collection K_C to each data state identified for c to object life cycle $l = \eta^{-1}(c)$
30: add the label of n as action to each transition (priorly existing or not)
31: replace the entries in K_c with data states of the input data nodes of the current activity
32: **end for**
33: add undirected synchronization edge between each two transitions, which have been added above and belong to different object life cycles l_1 and l_2
34: **if** c **then**urrently handled activity has multiple output data nodes referring to multiple data classes and at least one data class c' used as input does not have a corresponding output data node
35: add *concurrently*-typed synchronization edge between the state of each data node of class c' and each state of a data node of a class other than c
36: **end if**
37: **else if** node $n \in N$ is an XOR split **then**
38: update K_c for data class c used in data condition $dc = \xi(n, h)$ with $h \in N_{pm}$ being the direct successor of n in trace σ_A with data node d used as edge condition dc
39: **end if**
40: **until** trace σ_A has no next node $n \in N_{pm}$
41: **end for**
42: **for all** object life cycles $l \in \mathcal{L}$ **do**
43: data state $s \in S_l$ with no incoming transition gets the initial data state $s_i \in S_l$ of object life cycle l
44: all data states $s \in S_l$ with no outgoing transition comprise the set of final states $S_F \subseteq S_l$ of object life cycle l
45: **end for**

Algorithm 9 Transformation of a synchronized object life cycle $\mathcal{L} = (L, SE)$ into an object-centric process model ocp $= (AS, U, BR)$.

1: group all transitions $t \in \bigcup_{\mathcal{L}} \mathfrak{T}_{S,l}$ executed together into combined transitions CT
2: **for all** object life cycles $l \in \mathcal{L}$ **do**
3: create corresponding data class $c = \eta(l)$
4: add S_l to c
5: create empty map $K_c =< s + ID, collection < \mathcal{J} >>$ with $c = \varphi_D(d)$ being a current data class, s being the state of d, and $c.x \in \mathcal{J}$ representing the attribute x of data class c
6: **end for**
7: parse activity-centric process model with attribute definition pm'
8: **for all** data nodes $d \in D_{pm'}$ **do**
9: **for all** attribute $x \in J_{M,d} \cup \{pk_d\} \cup FK_d$ **do**
10: add x to $\mathcal{J}_c$ with $c = \varphi_D(d)$
11: add x to map K_c
12: **end for**
13: **end for**
14: **for all** combined transitions $ct \in CT$ **do**
15: create business rule br
16: extract task from transition labels of ct and add to business rule br
17: **for all** transitions t being combined in ct **do**
18: derive $instate_{pre}$ with parameters *class* corresponding to the object life cycle t is part of and the *source state* of t
19: extract $defined_{pre}$ from K_c for the *class* corresponding to the object life cycle t is part of and the *source state* of t
20: derive $instate_{pre}$ with parameters *class* corresponding to the object life cycle t is part of and the *target state* of t
21: extract $defined_{pre}$ from K_c for the *class* corresponding to the object life cycle t is part of and the *target state* of t
22: **end for**
23: group all $instate_{pre}$ functions with respect to their data class and combine elements within one group by $\vee$ operator and combine groups by $\wedge$ operator
24: combine all $defined_{pre}$ functions by $\vee$ operator and combine groups by $\wedge$ operator
25: combine the groups of all $instate_{pre}$ and $defined_{pre}$ functions by $\wedge$ operator and add as precondition to business rule br
26: group all $instate_{post}$ functions with respect to their data class and combine elements within one group by $\vee$ operator and combine groups by $\wedge$ operator
27: combine all $defined_{post}$ functions by $\vee$ operator and combine groups by $\wedge$ operator
28: combine the groups of all $instate_{post}$ and $defined_{post}$ functions by $\wedge$ operator and add as precondition to business rule br
29: **end for**
30: build object-centric process model $ocp = (AS, U, BR)$ with schema AS comprising all data classes c, the business rules BR, and the set of tasks U utilized in the business rules

BIBLIOGRAPHY

[1] Norris Syed Abdullah, Shazia Sadiq, and Marta Indulska. Emerging challenges in information systems research for regulatory compliance management. In *Advanced Information Systems Engineering (CAiSE)*, pages 251–265. Springer, 2010. (Cited on page 23.)

[2] Activiti. Activiti BPM Platform. URL https://www.activiti.org/. (Cited on pages 11, 23, 60, 195, and 278.)

[3] Rakesh Agrawal, Christopher Johnson, Jerry Kiernan, and Frank Leymann. Taming compliance with Sarbanes-Oxley internal controls using database technology. In *Data Engineering (ICDE)*, pages 92–101. IEEE, 2006. (Cited on page 150.)

[4] Rajeev Alur, Kousha Etessami, and Mihalis Yannakakis. Realizability and verification of MSC graphs. *Theoretical Computer Science*, 331(1):97–114, 2005. (Cited on page 266.)

[5] Andrea Arcuri. On the automation of fixing software bugs. In *Software Engineering (ICSE)*, pages 1003–1006. ACM, 2008. (Cited on page 142.)

[6] Andrea Arcuri and Xin Yao. A novel co-evolutionary approach to automatic software bug fixing. In *Evolutionary Computation (CEC)*, pages 162–168. IEEE, 2008. (Cited on page 142.)

[7] İsmaïlcem Budak Arpinar, Uğur Halici, Sena Arpinar, and Asuman Doğaç. Formalization of workflows and correctness issues in the presence of concurrency. *Distributed and Parallel Databases*, 7(2):199–248, 1999. (Cited on page 149.)

[8] Ahmed Awad. BPMN-Q: A language to query business processes. In *Enterprise Modelling and Information Systems Architectures (EMISA)*, pages 115–128. Gesellschaft für Informatik e.V. (GI), 2007. (Cited on page 150.)

[9] Ahmed Awad. *A compliance management framework for business process models*. PhD thesis, Hasso Plattner Institute at the University of Potsdam, 2010. (Cited on page 23.)

[10] Ahmed Awad, Gero Decker, and Mathias Weske. Efficient compliance checking using BPMN-Q and temporal logic. In *Business Process Management (BPM)*, pages 326–341. Springer, 2008. (Cited on page 150.)

[11] Ahmed Awad, Alexander Grosskopf, Andreas Meyer, and Mathias Weske. Enabling resource assignment constraints in BPMN. Technical report, Hasso Plattner Institute at the University of Potsdam, 2009. (Cited on page 4.)

[12] Ahmed Awad, Sergey Smirnov, and Mathias Weske. Resolution of compliance violation in business process models: A planning-based approach. In *On the Move to Meaningful Internet Systems (OTM)*, pages 6–23. Springer, 2009. (Cited on page 150.)

[13] Ahmed Awad, Matthias Weidlich, and Mathias Weske. Specification, verification and explanation of violation for data aware compliance rules. In *Service-Oriented Computing (ICSOC)*, pages 500–515. Springer, 2009. (Cited on page 150.)

[14] Ahmed Awad, Gero Decker, and Niels Lohmann. Diagnosing and repairing data anomalies in process models. In *Business Process Management (BPM) Workshops*, pages 5–16. Springer, 2010. (Cited on pages 96, 127, 142, and 152.)

[15] Ahmed Awad, Matthias Weidlich, and Mathias Weske. Visually specifying compliance rules and explaining their violations for business processes. *Journal of Visual Languages & Computing*, 22(1):30–55, 2011. (Cited on page 23.)

[16] Ben Azvine, Zhan Cui, Detlef D. Nauck, and Basim A. Majeed. Real time business intelligence for the adaptive enterprise. In *E-Commerce Technology / Enterprise Computing, E-Commerce and E-Services (CEC/EEE)*, page 29. IEEE, 2006. (Cited on page 314.)

[17] Michael Backmann, Anne Baumgrass, Nico Herzberg, Andreas Meyer, and Mathias Weske. Model-driven event query generation for business process monitoring. In *Service-Oriented Computing (ICSOC) Workshops*, pages 406–418. Springer, 2013. (Cited on page 24.)

[18] Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT press Cambridge, 2008. (Cited on pages 23 and 123.)

[19] Alistair Barros, Marlon Dumas, and Arthur H. M. ter Hofstede. Service interaction patterns. In *Business Process Management (BPM)*, pages 302–318. Springer, 2005. (Cited on pages 278, 281, 306, 308, and 311.)

[20] Twan Basten and Wil M. P. van der Aalst. Inheritance of behavior. *The Journal of Logic and Algebraic Programming*, 47(2):47–145, 2001. (Cited on pages 6 and 273.)

[21] John A. Bateman. Enabling technology for multilingual natural language generation: the KPML development environment. *Natural Language Engineering*, 3(1):15–55, 1997. (Cited on page 121.)

[22] Anne Baumgrass, Mark Strembeck, and Stefanie Rinderle-Ma. Deriving role engineering artifacts from business processes and scenario models. In *Access Control Models and Technologies (SACMAT)*, pages 11–20. ACM, 2011. (Cited on page 4.)

[23] Anne Baumgrass, Nico Herzberg, Andreas Meyer, and Mathias Weske. BPMN extension for business process monitoring. In *Enterprise Modelling and Information Systems Architectures (EMISA)*, pages 85–98. Gesellschaft für Informatik e.V. (GI), 2014. (Cited on page 24.)

[24] Jörg Becker, Martin Kugeler, and Michael Rosemann. *Process management: A guide for the design of business processes*. Springer, 2003. (Cited on pages 4, 13, 21, and 22.)

[25] Francesco Belardinelli, Alessio Lomuscio, and Fabio Patrizi. Verification of deployed artifact systems via data abstraction. In *Service-Oriented Computing (ICSOC)*, pages 142–156. Springer, 2011. (Cited on page 152.)

[26] Elisa Bertino, Elena Ferrari, and Vijay Atluri. The specification and enforcement of authorization constraints in workflow management systems. *ACM Transactions on Information and System Security (TISSEC)*, 2(1): 65–104, 1999. (Cited on page 4.)

[27] Eike Best. Structure theory of Petri nets: The free choice hiatus. In *Petri Nets: Central Models and Their Properties*, pages 168–205. Springer, 1987. (Cited on page 52.)

[28] Kamal Bhattacharya, Robert Guttman, Kelly Lyman, Fenno Terry Heath III, Santhosh Kumaran, Prabir Nandi, Frederick Wu, Prasanna Athma, Christoph Freiberg, Lars Johannsen, and Andreas Staudt. A model-driven approach to industrializing discovery processes in pharmaceutical research. *IBM Systems Journal*, 44(1):145–162, 2005. (Cited on page 11.)

[29] Kamal Bhattacharya, Cagdas E. Gerede, Richard Hull, Rong Liu, and Jianwen Su. Towards formal analysis of artifact-centric business process models. In *Business Process Management (BPM)*, pages 288–304. Springer, 2007. (Cited on page 152.)

[30] Kamal Bhattacharya, Richard Hull, and Jianwen Su. A data-centric design methodology for business processes. In *Handbook of Research on Business Process Modeling*, pages 503–531. IGI Global, 2009. (Cited on page 60.)

[31] Bizagi. Bizagi BPM suite. URL `https://www.bizagi.org/`. (Cited on page 11.)

[32] Bonitasoft. Bonita process engine. URL `https://www.bonitasoft.com/`. (Cited on pages 11 and 195.)

[33] Shawn Bowers and Bertram Ludäscher. An ontology-driven framework for data transformation in scientific workflows. In *Data Integration in the Life Sciences*, pages 1–16. Springer, 2004. (Cited on page 307.)

[34] BPM Academic Initiative. BPMAI process model collection. URL `http://bpmai.org/`. (Cited on pages 99 and 116.)

[35] Janis A. Bubenko and Benkt Wangler. Objectives driven capture of business rules and of information systems requirements. In *Systems, Man and Cybernetics*, pages 670–677. IEEE, 1993. (Cited on pages 8 and 23.)

[36] Susanne Bülow, Michael Backmann, Nico Herzberg, Thomas Hille, Andreas Meyer, Benjamin Ulm, Tsun Yin Wong, and Mathias Weske. Monitoring of business processes with complex event processing. In *Business Process Management (BPM) Workshops*, pages 277–290. Springer, 2013. (Cited on page 24.)

[37] Tevfik Bultan and Xiang Fu. Specification of realizable service conversations using collaboration diagrams. *Service Oriented Computing and Applications (SOCA)*, 2(1):27–39, 2008. (Cited on page 265.)

[38] Christoph Bussler. *B2B integration: Concepts and architecture*. Springer, 2003. (Cited on page 24.)

[39] Cristina Cabanillas. Enhancing the management of resource-aware business processes. *AI Communications*, 2015. (Cited on page 4.)

[40] Cristina Cabanillas, Manuel Resinas, and Antonio Ruiz-Cortés. Automated resource assignment in BPMN models using RACI matrices. In *On the Move to Meaningful Internet Systems (OTM)*, pages 56–73. Springer, 2012. (Cited on page 4.)

[41] Cristina Cabanillas, José María García, Manuel Resinas, David Ruiz, Jan Mendling, and Antonio Ruiz-Cortés. Priority-based human resource allocation in business processes. In *Service-Oriented Computing (ICSOC)*, pages 374–388. Springer, 2013. (Cited on page 4.)

[42] Cristina Cabanillas, Claudio Di Ciccio, Jan Mendling, and Anne Baumgrass. Predictive task monitoring for business processes. In *Business Process Management (BPM)*, pages 424–432. Springer, 2014. (Cited on page 24.)

[43] Cristina Cabanillas, Alex Norta, Manuel Resinas, Jan Mendling, and Antonio Ruiz-Cortés. Towards process-aware cross-organizational human resource management. In *Enterprise, Business-Process and Information Systems Modeling (EMMSAD/BPMDS)*, pages 79–93. Springer, 2014. (Cited on page 4.)

[44] Javier Cámara, Carlos Canal, Javier Cubo, and Antonio Vallecillo. Formalizing WS-BPEL business processes using process algebra. *Electronic Notes in Theoretical Computer Science*, 154(1):159–173, 2006. (Cited on page 96.)

[45] Camunda. Camunda BPM Platform. URL `https://www.camunda.org/`. (Cited on pages 11, 23, and 195.)

[46] Donald D. Chamberlin and Raymond F. Boyce. SEQUEL: A structured English query language. In *SIGFIDET (now SIGMOD) workshop*, pages 249–264. ACM, 1974. (Cited on page 278.)

[47] T. C. Edwin Cheng, Valery S. Gordon, and Mikhail Y. Kovalyov. Single machine scheduling with batch deliveries. *European Journal of Operational Research*, 94(2):277–283, 1996. (Cited on page 314.)

[48] Carolina Ming Chiao, Vera Künzle, and Manfred Reichert. A tool for supporting object-aware processes. In *Enterprise Distributed Object Computing Workshops and Demonstrations (EDOCW)*, pages 410–413. IEEE, 2014. (Cited on page 11.)

[49] Michele Chinosi and Alberto Trombetta. BPMN: An introduction to the standard. *Computer Standards & Interfaces*, 34(1):124–134, 2012. (Cited on page 26.)

[50] Andrzej Cichocki, Helal A. Ansari, Marek Rusinkiewicz, and Darrel Woelk. *Workflow and process automation: Concepts and technology*. Springer, 1998. (Cited on page 6.)

[51] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model checking*. MIT press, 2000. (Cited on page 123.)

[52] Anne Cleven and Felix Wortmann. Uncovering four strategies to approach master data management. In *System Sciences (HICSS)*, pages 1–10. IEEE, 2010. (Cited on page 60.)

[53] Edgar F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970. (Cited on page 278.)

[54] David Cohn and Richard Hull. Business artifacts: A data-centric approach to modeling business operations and processes. *IEEE Data Engineering Bulletin*, 32(3):3–9, 2009. (Cited on pages 94, 121, 152, 156, 190, 199, 303, and 304.)

[55] David Cohn, Pankaj Dhoolia, Fenno Terry Heath III, Florian Pinel, and John Vergo. Siena: From powerpoint to web app in 5 minutes. In *Service-Oriented Computing (ICSOC)*, pages 722–723. Springer, 2008. (Cited on page 11.)

[56] Thomas A. Curran, Gerhard Keller, and Andrew Ladd. *SAP R/3 business blueprint: Understanding the business process reference model*. Prentice-Hall, Inc., 1997. (Cited on pages 99 and 101.)

[57] Krzysztof Czarnecki and Simon Helsen. Feature-based survey of model transformation approaches. *IBM Systems Journal*, 45(3):621–645, 2006. (Cited on page 191.)

[58] Krzysztof Czarnecki, J. Nathan Foster, Zhenjiang Hu, Ralf Lämmel, Andy Schürr, and James F. Terwilliger. Bidirectional transformations: A cross-discipline perspective. In *Theory and Practice of Model Transformations*, pages 260–283. Springer, 2009. (Cited on page 191.)

[59] Ajantha Dahanayake, Richard J. Welke, and Gabriel Cavalheiro. Improving the understanding of BAM technology for real-time decision support. *International Journal of Business Information Systems (IJBIS)*, 7(1): 1–26, 2011. (Cited on page 314.)

[60] Elio Damaggio, Richard Hull, and Roman Vaculín. On the equivalence of incremental and fixpoint semantics for business artifacts with Guard–Stage–Milestone lifecycles. *Information Systems*, 38(4):561–584, 2013. (Cited on page 303.)

[61] Suresh Damodaran. B2B integration over the Internet with XML: RosettaNet successes and challenges. In *World Wide Web (WWW)*, pages 188–195. ACM, 2004. (Cited on page 24.)

[62] Thomas H. Davenport. *Process innovation: Reengineering work through information technology*. Harvard Business Review Press, 1992. (Cited on pages 3, 4, and 5.)

[63] Thomas H. Davenport and J. Short. Information technology and business process redesign. *Operations Management: Critical Perspectives on Business and Management*, 1:97, 2003. (Cited on pages 6 and 25.)

[64] Massimiliano de Leoni, Wil M. P. van der Aalst, and Boudewijn F. van Dongen. Data-and resource-aware conformance checking of business processes. In *Business Information Systems (BIS)*, pages 48–59. Springer, 2012. (Cited on page 152.)

[65] Rina Dechter and Judea Pearl. Generalized best-first search strategies and the optimality of A*. *Journal of the ACM (JACM)*, 32(3):505–536, 1985. (Cited on page 152.)

[66] Gero Decker and Alistair Barros. Interaction modeling using BPMN. In *Business Process Management (BPM) Workshops*, pages 208–219. Springer, 2008. (Cited on page 306.)

[67] Gero Decker and Mathias Weske. Behavioral consistency for B2B process integration. In *Advanced Information Systems Engineering (CAiSE)*, pages 81–95. Springer, 2007. (Cited on pages 93 and 270.)

[68] Gero Decker and Mathias Weske. Local enforceability in interaction Petri nets. In *Business Process Management (BPM)*, pages 305–319. Springer, 2007. (Cited on pages 265 and 272.)

[69] Gero Decker and Mathias Weske. Interaction-centric modeling of process choreographies. *Information Systems*, 36(2):292–312, 2011. (Cited on pages 24, 81, 249, 265, 272, and 306.)

[70] Gero Decker, Oliver Kopp, Frank Leymann, and Mathias Weske. BPEL4Chor: Extending BPEL for modeling choreographies. In *Web Services (ICWS)*, pages 296–303. IEEE, 2007. (Cited on pages 253 and 306.)

[71] Gero Decker, Remco Dijkman, Marlon Dumas, and Luciano García-Bañuelos. Transforming BPMN diagrams into YAWL nets. In *Business Process Management (BPM)*, pages 386–389. Springer, 2008. (Cited on page 96.)

[72] Juliane Dehnert and Peter Rittgen. Relaxed soundness of business processes. In *Advanced Information Systems Engineering (CAiSE)*, pages 157–170. Springer Berlin / Heidelberg, 2001. (Cited on page 266.)

[73] Adela del Río-Ortega, Manuel Resinas, Cristina Cabanillas, and Antonio Ruiz-Cortés. On the definition and design-time analysis of process performance indicators. *Information Systems*, 38(4):470–490, 2013. (Cited on page 9.)

[74] Jörg Desel and Javier Esparza. *Free choice Petri nets*, volume 40 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1995. (Cited on pages 49, 52, 53, and 123.)

[75] Alin Deutsch, Richard Hull, Fabio Patrizi, and Victor Vianu. Automatic verification of data-centric business processes. In *International Conference on Database Theory*, pages 252–267. ACM, 2009. (Cited on page 152.)

[76] David DeWitt and Jim Gray. Parallel database systems: The future of high performance database systems. *Communications of the ACM*, 35(6):85–98, 1992. (Cited on page 306.)

[77] Volker Diekert and Grzegorz Rozenberg. *The book of traces*. World Scientific, 1995. (Cited on page 51.)

[78] Remco Dijkman, Marcello La Rosa, and Hajo A. Reijers. Managing large collections of business process models-current techniques and challenges. *Computers in Industry*, 63(2):91–97, 2012. (Cited on page 3.)

[79] Remco Dijkman, Irene Vanderfeesten, and Hajo A. Reijers. Business process architectures: Overview, comparison and framework. *Enterprise Information Systems*, pages 1–30, 2014. (Cited on pages 22, 81, 153, 181, and 314.)

[80] Remco M. Dijkman, Marlon Dumas, and Chun Ouyang. Semantics and analysis of business process models in BPMN. *Information & Software Technology*, 50(12):1281–1294, 2008. (Cited on pages 15, 28, 42, 53, 86, 87, 89, 91, 95, 96, 97, 131, 141, 266, 267, 275, 276, and 311.)

[81] Dov Dori. *Object process methodology: A holistic systems paradigm*. Springer, 2002. (Cited on pages 303 and 304.)

[82] Marlon Dumas, Wil M. P. van der Aalst, and Arthur H. M. ter Hofstede. *Process-aware information systems: Bridging people and software through process technology*. John Wiley & Sons, 2005. (Cited on pages 4, 6, 11, 23, and 25.)

[83] Marlon Dumas, Marcello La Rosa, Jan Mendling, and Hajo A. Reijers. *Fundamentals of business process management*. Springer, 2013. (Cited on pages 4, 5, and 23.)

[84] Effektif. Effektif workflow engine. URL http://www.effektif.com/. (Cited on page 11.)

[85] Rami-Habib Eid-Sabbagh and Mathias Weske. From process models to business process architectures: Connecting the layers. In *Service-Oriented Computing (ICSOC) Workshops*, pages 4–15. Springer, 2013. (Cited on page 181.)

[86] Rami-Habib Eid-Sabbagh, Remco M. Dijkman, and Mathias Weske. Business process architecture: Use and correctness. In *Business Process Management (BPM)*, pages 65–81. Springer, 2012. (Cited on pages 22, 153, 181, and 314.)

[87] Rami-Habib Eid-Sabbagh, Marcin Hewelt, Andreas Meyer, and Mathias Weske. Deriving business process data architectures from process model collections. In *Service-Oriented Computing (ICSOC)*, pages 533–540. Springer, 2013. (Cited on pages 22, 153, and 315.)

[88] Rami-Habib Eid-Sabbagh, Marcin Hewelt, and Mathias Weske. Business process architectures with multiplicities: Transformation and correctness. In *Business Process Management (BPM)*, pages 227–234. Springer, 2013. (Cited on pages 81, 153, and 181.)

[89] Marwane El Kharbili, Ana Karla A. de Medeiros, Sebastian Stein, and Wil M. P. van der Aalst. Business process compliance checking: Current state and future challenges. In *Modellierung betrieblicher Informationssysteme (MobIS)*, volume 141, pages 107–113. Gesellschaft für Informatik e.V. (GI), 2008. (Cited on page 149.)

[90] John Ellson, Emden Gansner, Lefteris Koutsofios, Stephen C. North, and Gordon Woodhull. Graphviz—open source graph drawing tools. In *Graph Drawing*, pages 483–484. Springer, 2002. (Cited on pages 114 and 115.)

[91] D. Jack Elzinga, Thomas Horak, Chung-Yee Lee, and Charles Bruner. Business process management: Survey and methodology. *IEEE Transactions on Engineering Management*, 42(2):119–128, 1995. (Cited on page 3.)

[92] E. Allen Emerson and Joseph Y. Halpern. Decision procedures and expressiveness in the temporal logic of branching time. *Journal of Computer and System Sciences*, 30(1):1–24, 1985. (Cited on page 153.)

[93] Thomas Erl. *SOA: Principles of service design*. Prentice Hall, 2008. (Cited on page 9.)

[94] Rik Eshuis. Symbolic model checking of UML activity diagrams. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 15(1):1–38, 2006. (Cited on page 150.)

[95] Rik Eshuis and Paul Grefen. Structural matching of BPEL processes. In *Web Services (ECOWS)*, pages 171–180, Washington, DC, USA, 2007. IEEE Computer Society. (Cited on page 48.)

[96] Rik Eshuis and Paul Grefen. Constructing customized process views. *Data & Knowledge Engineering*, 64(2):419–438, 2008. (Cited on pages 23, 181, and 315.)

[97] Rik Eshuis and Peter van Gorp. Synthesizing object life cycles from business process models. In *Conceptual Modeling (ER)*, pages 307–320. Springer, 2012. (Cited on pages 100, 121, 156, 191, 199, 208, and 303.)

[98] Rik Eshuis and Pieter van Gorp. Synthesizing object-centric models from business process models. In *Business Process Management (BPM) Workshops*, pages 155–166. Springer, 2014. (Cited on pages 100, 121, 156, and 191.)

[99] Dirk Fahland. *From scenarios to components*. PhD thesis, Humboldt-Universität zu Berlin, 2010. (Cited on page 315.)

[100] Dirk Fahland and Robert Prüfer. Data and abstraction for scenario-based modeling with Petri nets. In *Application and Theory of Petri Nets (ICATPN)*, pages 168–187. Springer, 2012. (Cited on page 315.)

[101] Dirk Fahland, Cédric Favre, Barbara Jobstmann, Jana Koehler, Niels Lohmann, Hagen Völzer, and Karsten Wolf. Instantaneous soundness checking of industrial business process models. In *Business Process Management (BPM)*, pages 278–293. Springer, 2009. (Cited on page 126.)

[102] Dirk Fahland, Massimiliano de Leoni, Boudewijn F. Dongen, and Wil M. P. van der Aalst. Conformance checking of interacting processes with overlapping instances. In *Business Process Management (BPM)*, pages 345–361. Springer, 2011. (Cited on page 152.)

[103] Dirk Fahland, Cédric Favre, Jana Koehler, Niels Lohmann, Hagen Völzer, and Karsten Wolf. Analysis on demand: Instantaneous soundness checking of industrial business process models. *Data & Knowledge Engineering*, 70(5):448–466, 2011. (Cited on page 276.)

[104] Shaokun Fan, Wanchun Dou, and Jinjun Chen. Dual workflow nets: Mixed control/data-flow representation for workflow modeling and verification. In *Advances in Web and Network Technologies, and Information Management*, pages 433–444. Springer, 2007. (Cited on page 151.)

[105] Cédric Favre and Hagen Völzer. The difficulty of replacing an inclusive OR-join. In *Business Process Management (BPM)*, pages 156–171, 2012. (Cited on page 112.)

[106] Leonard Fortuin. Performance indicators – Why, where and how? *European Journal of Operational Research*, 34(1):1–9, 1988. (Cited on page 9.)

[107] Howard Foster, Sebastian Uchitel, Jeff Magee, and Jeff Kramer. Model-based verification of web service compositions. In *Automated Software Engineering*, pages 152–161. IEEE, 2003. (Cited on page 96.)

[108] Heinz Frank and Johann Eder. Integration of statecharts. In *On the Move to Meaningful Internet Systems (OTM)*, pages 364–372. IEEE, 1998. (Cited on pages 80, 151, and 189.)

[109] Jakob Freund and Bernd Rücker. *Real-life BPMN: Using BPMN2.0 to analyze, improve, and automate processes in your company*. CreateSpace Independent Publishing Platform, 2012. (Cited on pages 4 and 5.)

[110] Jakob Freund and Bernd Rücker. *Praxishandbuch BPMN 2.0*. Hanser Fachbuchverlag, 2014. (Cited on page 26.)

[111] Xiang Fu, Tevfik Bultan, and Jianwen Su. Conversation protocols: a formalism for specification and verification of reactive electronic services. *Theoretical Computer Science*, 328(1):19–37, 2004. (Cited on page 265.)

[112] Timo Füermann and Carsten Dammasch. *Prozessmanagement: Anleitung zur ständigen Prozessverbesserung*. Hanser Verlag, 2008. (Cited on pages 4 and 5.)

[113] Michael Gaitanides. *Prozeßorganisation: Entwicklung, Ansätze und Programme prozeßorientierter Organisationsgestaltung*. Vahlen, 1983. (Cited on page 3.)

[114] Mauro Gambini, Marcello La Rosa, Sara Migliorini, and Arthur H. M. ter Hofstede. Automated error correction of business process models. In *Business Process Management (BPM)*, pages 148–165. Springer, 2011. (Cited on page 141.)

[115] Cagdas E. Gerede and Jianwen Su. Specification and verification of artifact behaviors in business process models. In *Service-Oriented Computing (ICSOC)*, pages 181–192. Springer, 2007. (Cited on page 152.)

[116] Stijn Goedertier and Jan Vanthienen. Designing compliant business processes with obligations and permissions. In *Business Process Management (BPM) Workshops*, pages 5–14. Springer, 2006. (Cited on page 150.)

[117] Guido Governatori, Zoran Milosevic, and Shazia Sadiq. Compliance checking between business processes and business contracts. In *Enterprise Distributed Object Computing (EDOC)*, pages 221–232. IEEE, 2006. (Cited on pages 23 and 150.)

[118] Daniela Grigori, Fabio Casati, Malu Castellanos, Umeshwar Dayal, Mehmet Sayal, and Ming-Chien Shan. Business process intelligence. *Computers in Industry*, 53(3):321–343, April 2004. (Cited on pages 9 and 314.)

[119] Varun Grover, Kirk D. Fiedler, and James T. C. Teng. Exploring the success of information technology enabled business process reengineering. *IEEE Transactions on Engineering Management*, 41(3):276–284, 1994. (Cited on pages 8 and 23.)

[120] Stephan Haarmann. Implementation of an alignment procedure between synchronized object life cycles and production case management sceanrios. Bachelor's thesis, Hasso Plattner Institute at the University of Potsdam, June 2009. (Cited on page 318.)

[121] Michel H. T. Hack. Analysis of production schemata by Petri nets. Technical report, Massachusetts Institute of Technology, 1972. (Cited on page 52.)

[122] Michel H. T. Hack. *Decidability questions for Petri nets*. PhD thesis, Massachusetts Institute of Technology. Dept. of Electrical Engineering and Computer Science, 1976. (Cited on pages 49, 51, and 53.)

[123] Alon Y. Halevy, Zachary G. Ives, Dan Suciu, and Igor Tatarinov. Schema mediation in peer data management systems. In *Data Engineering (ICDE)*, pages 505–516. IEEE, 2003. (Cited on page 306.)

[124] Alena Hallerbach, Thomas Bauer, and Manfred Reichert. Capturing variability in business process models: The Provop approach. *Journal of Software Maintenance and Evolution: Research and Practice*, 22(6-7):519–546, 2010. (Cited on page 315.)

[125] Michael Hammer. What is business process management? In *Handbook on Business Process Management 1*, pages 3–16. Springer, Berlin Heidelberg, 2010. (Cited on page 5.)

[126] Michael Hammer and James Champy. *Reengineering the corporation: Manifesto for business revolution*. HarperBusiness, 1993. (Cited on pages 3, 4, and 5.)

[127] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987. (Cited on page 47.)

[128] David Harel and Bernhard Rumpe. Meaningful modeling: What's the semantics of "semantics"? *Computer*, 37(10):64–72, 2004. (Cited on page 21.)

[129] Paul Harmon and Celia Wolf. The state of business process management. Technical report, BPTrends, 2014. (Cited on page 11.)

[130] H. James Harrington. *Business process improvement*. McGraw, 1991. (Cited on page 6.)

[131] Josefine Harzmann, Andreas Meyer, and Mathias Weske. Deciding data object relevance for business process model abstraction. In *Conceptual Modeling (ER)*, pages 121–129. Springer, 2013. (Cited on pages 100, 157, 181, and 315.)

[132] Fenno Terry Heath III, David Boaz, Manmohan Gupta, Roman Vaculín, Yutian Sun, Richard Hull, and Lior Limonad. Barcelona: A design and runtime environment for declarative artifact-centric BPM. In *Service-Oriented Computing (ICSOC)*, pages 705–709. Springer, 2013. (Cited on page 11.)

[133] Nico Herzberg, Andreas Meyer, Oleh Khovalko, and Mathias Weske. Improving business process intelligence with object state transition events. In *Conceptual Modeling (ER)*, pages 146–160. Springer, 2013. (Cited on page 314.)

[134] Nico Herzberg, Andreas Meyer, and Mathias Weske. An event processing platform for business process management. In *Enterprise Distributed Object Computing (EDOC)*, pages 107–116. IEEE, 2013. (Cited on pages 24 and 314.)

[135] Nico Herzberg, Andreas Meyer, and Mathias Weske. Improving process monitoring and progress prediction with data state transition events. In *Services and their Composition (ZEUS)*, pages 20–23. CEUR-WS, 2013. (Cited on page 314.)

[136] Nico Herzberg, Andreas Meyer, and Mathias Weske. Improving business process intelligence by observing object state transitions. *Data & Knowledge Engineering (DKE)*, 98:144–164, 2015. (Cited on pages 24 and 314.)

[137] Sebastian Hinz, Karsten Schmidt, and Christian Stahl. Transforming BPEL to Petri nets. In *Business Process Management (BPM)*, pages 220–235. Springer, 2005. (Cited on page 95.)

[138] Charles A. R. Hoare. A model for communicating sequential processes. Technical report 80-1, Department of Computing Science, University of Wollongong, 1980. (Cited on page 51.)

[139] Charles Antony Richard Hoare. *Communicating sequential processes*, volume 178. Prentice-Hall, 1985. (Cited on page 95.)

[140] Gregor Hohpe and Bobby Woolf. *Enterprise integration patterns: Designing, building, and deploying messaging solutions*. Addison-Wesley, 2004. (Cited on page 249.)

[141] Richard Hull, Elio Damaggio, Fabiana Fournier, Manmohan Gupta, Fenno (Terry) Heath III, Stacy Hobson, Mark Linehan, Sridhar Maradugu, Anil Nigam, Piyawadee Sukaviriya, and Roman Vaculin. Introducing the Guard-Stage-Milestone approach for specifying business entity lifecycles. In *Web Services and Formal Methods (WS-FM)*, pages 1–24. Springer, 2011. (Cited on pages 44 and 156.)

[142] IBM. BIT process library. URL `http://www.zurich.ibm.com/csc/bit/downloads.html/`. (Cited on page 99.)

[143] Sven Ihde. Datenobjekte und Formulargenerierung im Kontext von Production Case Management. Bachelor's thesis, Hasso Plattner Institute at the University of Potsdam, June 2009. (Cited on page 318.)

[144] Marta Indulska, Peter Green, Jan Recker, and Michael Rosemann. Business process modeling: Perceived benefits. In *Conceptual Modeling (ER)*, pages 458–471. Springer, 2009. (Cited on pages 13, 22, and 24.)

[145] Marta Indulska, Jan Recker, Michael Rosemann, and Peter Green. Business process modeling: Current issues and future challenges. In *Advanced Information Systems Engineering (CAiSE)*, pages 501–514. Springer, 2009. (Cited on pages 10, 11, and 24.)

[146] Internet Engineering Task Force (IETF). The JavaScript Object Notation (JSON) Data Interchange Format (RFC 7159), 2014. (Cited on page 281.)

[147] ISO/IEC. ISO/IEC 9075-14:2011: Information technology – database languages – SQL – Part 14: XML-related specifications (SQL/XML), February 2011. URL http://www.iso.org/iso/home/store/catalogue_tc/catalogue_detail.htm?csnumber=53686. (Cited on pages 200 and 278.)

[148] ISO/IEC. ISO 9001:2008: Quality management systems – requirements, May 2012. URL http://www.iso.org/iso/catalogue_detail?csnumber=46486. (Cited on page 11.)

[149] JBoss. jBPM process engine. URL https://www.jboss.org/jbpm/. (Cited on page 11.)

[150] Richard C. Johnson, David Pearson, and Keshav Pingali. Finding regions fast: Single entry single exit and control regions in linear time. Technical report, Cornell University, 1993. (Cited on page 90.)

[151] Richard C. Johnson, David Pearson, and Keshav Pingali. The program structure tree: Computing control regions in linear time. In *Programming Language Design and Implementation (PLDI)*, pages 171–185. ACM, 1994. (Cited on page 90.)

[152] jQueryFoundation. jQuery Core – fast, small, and feature-rich JavaScript library. URL https://www.jquery.com/. (Cited on page 281.)

[153] Mohan Kamath and Krithi Ramamritham. Correctness issues in workflow management. *Distributed Systems Engineering*, 3(4):213–221, 1996. (Cited on page 149.)

[154] Gerti Kappel and Michael Schrefl. Object/behavior diagrams. In *Data Engineering (ICDE)*, pages 530–539. IEEE, 1991. (Cited on pages 11 and 266.)

[155] Roland Kaschek. A little theory of abstraction. In *Modellierung*, pages 75–92, 2004. (Cited on page 3.)

[156] Raman Kazhamiakin and Marco Pistore. Analysis of realizability conditions for web service choreographies. In *Formal Techniques for Networked and Distributed Systems (FORTE)*, pages 61–76. Springer, 2006. (Cited on page 266.)

[157] Gerhard Keller, Markus Nüttgens, and August-Wilhelm Scheer. Semantische Prozeßmodellierung auf der Basis Ereignisgesteuerter Prozeßketten (EPK). *Veröffentlichungen des Instituts für Wirtschaftsinformatik*, 89, 1992. (Cited on pages 11 and 86.)

[158] Stuart Kent. Model driven engineering. In *Integrated formal methods*, pages 286–298. Springer, 2002. (Cited on pages 191 and 307.)

[159] Bartek Kiepuszewski, Arthur H. M. ter Hofstede, and Wil M. P. van der Aalst. Fundamentals of control flow in workflows. *Acta Informatica*, 39 (3):143–209, 2003. (Cited on page 52.)

[160] Anneke G. Kleppe, Jos B. Warmer, and Wim Bast. *MDA explained: The model driven architecture: Practice and promise*. Addison-Wesley, 2003. (Cited on page 307.)

[161] Ralph L. Kliem. Risk management for business process reengineering projects. *Information Systems Management*, 17(4):71–73, 2000. (Cited on page 23.)

[162] Gerhard Knolmayer, Rainer Endl, and Marcel Pfahrer. Modeling processes and workflows by business rules. In *Business Process Management (BPM)*, pages 16–29. Springer, 2000. (Cited on page 23.)

[163] David Knuplesch, Rudiger Pryss, and Manfred Reichert. Data-aware interaction in distributed and collaborative workflows: Modeling, semantics, correctness. In *Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom)*, pages 223–232. IEEE, 2012. (Cited on page 306.)

[164] Jana Koehler, Giuliano Tirenni, and Santhosh Kumaran. From business process model to consistent implementation: A case for formal verification methods. In *Enterprise Distributed Object Computing (EDOC)*, pages 96–106. IEEE, 2002. (Cited on page 23.)

[165] Jens Kolb and Manfred Reichert. A flexible approach for abstracting and personalizing large business process models. *ACM SIGAPP Applied Computing Review*, 13(1):6–18, 2013. (Cited on pages 181 and 315.)

[166] Andrei Kovalyov. Concurrency relations and the safety problem for Petri nets. In *Application and Theory of Petri Nets (ICATPN)*, pages 299–309. Springer, 1992. (Cited on page 145.)

[167] Andrei Kovalyov and Javier Esparza. A polynomial algorithm to compute the concurrency relation of free-choice Signal Transition Graphs. Technical report SFB-Bericht Nr. 342/15/1995 A, Technical University Munich, 1995. (Cited on page 145.)

[168] Hans-Ulrich. Krause and Dayanand Arora. Key performance indicators, 2010. (Cited on page 9.)

[169] Thomas Kühne. Matters of (meta-) modeling. *Software & Systems Modeling*, 5(4):369–385, 2006. (Cited on pages 3 and 21.)

[170] Santhosh Kumaran, Rong Liu, and Frederick Y. Wu. On the duality of information-centric and activity-centric models of business processes. In *Advanced Information Systems Engineering (CAiSE)*, pages 32–47. Springer, 2008. (Cited on page 156.)

[171] Matthias Kunze, Alexander Luebbe, Matthias Weidlich, and Mathias Weske. Towards understanding process modeling – the case of the BPM Academic Initiative. In *Business Process Model and Notation (BPMN)*, pages 44–58. Springer, 2011. (Cited on pages 39, 69, 101, and 111.)

[172] Vera Künzle. *Object-aware process management*. PhD thesis, University of Ulm, 2013. (Cited on page 11.)

[173] Vera Künzle and Manfred Reichert. PHILharmonicFlows: Towards a framework for object-aware process management. *Journal of Software Maintenance and Evolution: Research and Practice*, 23(4):205–244, 2011. (Cited on pages 26, 94, 156, 190, 199, 200, 302, 303, 304, 305, and 314.)

[174] Vera Künzle and Manfred Reichert. Striving for object-aware process support: How existing approaches fit together. In *Data-Driven Process Discovery and Analysis (SIMPDA)*, pages 169–188. Springer, 2011. (Cited on page 60.)

[175] Vera Künzle, Barbara Weber, and Manfred Reichert. Object-aware business processes: Fundamental requirements and their support in existing approaches. *International Journal of Information System Modeling and Design*, 2(2):19–46, 2011. (Cited on pages 94 and 199.)

[176] Jochen Küster, Ksenia Ryndina, and Harald Gall. Generation of business process models for object life cycle compliance. In *Business Process Management (BPM)*, pages 165–181. Springer, 2007. (Cited on pages 16, 95, 100, 124, 126, 149, 150, 152, 153, 156, and 176.)

[177] Andreas Lanz, Manfred Reichert, and Peter Dadam. Robust and flexible error handling in the AristaFlow BPM suite. In *Advanced Information Systems Engineering (CAiSE) Forum*, volume 72, pages 174–189. Springer, 2011. (Cited on pages 11 and 195.)

[178] Alexei Lapouchnian. Goal-oriented requirements engineering: An overview of the current research. Technical report, University of Toronto, 2005. (Cited on page 8.)

[179] Jill H. Larkin and Herbert A. Simon. Why a diagram is (sometimes) worth ten thousand words. *Cognitive Science*, 11(1):65–100, 1987. (Cited on page 96.)

[180] Maria Leitner, Stefanie Rinderle-Ma, and Juergen Mangler. Responsibility-driven design and development of process-aware security policies. In *Availability, Reliability and Security (ARES)*, pages 334–341. IEEE, 2011. (Cited on page 4.)

[181] Maurizio Lenzerini. Data integration: A theoretical perspective. In *Principles of Database Systems (PODS)*, pages 233–246. ACM, 2002. (Cited on page 306.)

[182] Henrik Leopold, Sergey Smirnov, and Jan Mendling. Recognising activity labeling styles in business process models. *Enterprise Modelling and Information Systems Architectures (EMISA)*, 6(1):16–29, 2011. (Cited on pages 102 and 121.)

[183] Henrik Leopold, Sergey Smirnov, and Jan Mendling. On the refactoring of activity labels in business process models. *Information Systems*, 37(5): 443–459, 2012. (Cited on pages 41, 102, and 121.)

[184] Frank Leymann and Dieter Roller. *Production workflow: Concepts and techniques*. Prentice Hall, 2000. (Cited on pages 4 and 23.)

[185] Jianxun Liu and Jinmin Hu. Dynamic batch processing in workflows: Model and implementation. *Future Generation Computer Systems*, 23(3): 338–347, 2007. (Cited on page 314.)

[186] Rong Liu, Frederick Y. Wu, and Santhosh Kumaran. Transforming activity-centric business process models into information-centric models for SOA solutions. *Journal of Database Management*, 21(4):14–34, 2010. (Cited on pages 95, 100, 121, 122, 156, 191, 199, 208, and 303.)

[187] Niels Lohmann. Compliance by design for artifact-centric business processes. In *Business Process Management (BPM)*, pages 99–115. Springer, 2011. (Cited on page 150.)

[188] Niels Lohmann. Compliance by design for artifact-centric business processes. *Information Systems*, 38(4):606–618, 2013. (Cited on page 150.)

[189] Niels Lohmann and Karsten Wolf. From artifacts to activities. In *Web Services Foundations*, pages 109–135. Springer, 2014. (Cited on page 191.)

[190] Niels Lohmann, Eric Verbeek, and Remco Dijkman. Petri net transformations for business processes – a survey. In *Transactions on Petri Nets and Other Models of Concurrency II, Special Issue on Concurrency in Process-Aware Information Systems*, pages 46–63. Springer, 2009. (Cited on pages 23, 86, 95, 154, 276, and 312.)

[191] David Loshin. *Master data management*. Morgan Kaufmann, 2010. (Cited on page 60.)

[192] Matteo Magnani and Danilo Montesi. BPMN: How much does it cost? An incremental approach. In *Business Process Management (BPM)*, pages 80–87. Springer, 2007. (Cited on page 23.)

[193] Thomas W. Malone, Kevin Crowston, and George Arthur Herman. *Organizing business knowledge: The MIT process handbook*. MIT press, 2003. (Cited on page 120.)

[194] Jürgen Mangler and Stefanie Rinderle-Ma. Rule-based synchronization of process activities. In *Commerce and Enterprise Computing (CEC)*, pages 121–128. IEEE, 2011. (Cited on page 314.)

[195] Axel Martens. On usability of web services. In *Web Information Systems Engineering Workshops*, pages 182–190, Rome, Italy, 2003. IEEE. (Cited on pages 54, 136, 139, 266, and 271.)

[196] Antoni Mazurkiewicz. Concurrent program schemes and their interpretations. Technical report DAIMI Report PB-78, Department of Computer Science, Aarhus University, 1977. (Cited on page 51.)

[197] Hema S. Meda, Anup Kumar Sen, and Amitava Bagchi. On detecting data flow errors in workflows. *Journal of Data and Information Quality (JDIQ)*, 2(1):4, 2010. (Cited on page 153.)

[198] Jyotiprasad Medhi. *Stochastic models in queueing theory*. Academic Press, 2002. (Cited on page 314.)

[199] Florian Melchert, Robert Winter, and Mario Klesse. Aligning process automation and business intelligence to support corporate performance management. In *Information Systems (AMCIS)*, pages 4053–4063. Association for Information Systems, 2004. (Cited on page 314.)

[200] Jan Mendling. *Detection and prediction of errors in epc business process models*. PhD thesis, Vienna University of Economics and Business Administration, 2007. (Cited on page 5.)

[201] Jan Mendling and Michael Hafner. From WS-CDL choreography to BPEL process orchestration. *Journal of Enterprise Information Management*, 21(5):525–542, 2008. (Cited on page 249.)

[202] Jan Mendling, Kristian Bisgaard Lassen, and Uwe Zdun. Transformation strategies between block-oriented and graph-oriented process modelling languages. In *Multikonferenz Wirtschaftsinformatik*, volume 2, pages 297–312. GITO-Verlag, 2006. (Cited on page 97.)

[203] Jan Mendling, Hajo A. Reijers, and Jorge Cardoso. What makes process models understandable? In *Business Process Management (BPM)*, pages 48–63. Springer, 2007. (Cited on page 120.)

[204] Jan Mendling, Hajo A. Reijers, and Jan Recker. Activity labeling in process modeling: Empirical insights and recommendations. *Information Systems*, 35(4):467–482, 2010. (Cited on pages 40, 100, and 121.)

[205] Jan Mendling, Hajo A. Reijers, and Wil M. P. van der Aalst. Seven process modeling guidelines (7PMG). *Information & Software Technology*, 52(2):127–136, 2010. (Cited on page 120.)

[206] Tom Mens. Model transformation: A survey of the state of the art. In *Model-Driven Engineering for Distributed Real-Time Systems*, pages 1–19. John Wiley & Sons, 2013. (Cited on page 191.)

[207] Tom Mens and Pieter van Gorp. A taxonomy of model transformation. *Electronic Notes in Theoretical Computer Science*, 152:125–142, 2006. (Cited on page 191.)

[208] Andreas Meyer. Resource perspective in BPMN. Master's thesis, Hasso Plattner Institute at the University of Potsdam, March 2009. (Cited on pages 4 and 69.)

[209] Andreas Meyer and Mathias Weske. Data support in process model abstraction. In *Conceptual Modeling (ER)*, pages 292–306. Springer, 2012. (Cited on pages 100, 157, 181, and 315.)

[210] Andreas Meyer and Mathias Weske. Activity-centric and artifact-centric process model roundtrip. In *Business Process Management (BPM) Workshops*, pages 167–181. Springer, 2013. (Cited on pages 59, 155, and 199.)

[211] Andreas Meyer and Mathias Weske. Activity-centric and artifact-centric process model roundtrip. Technical report, Hasso Plattner Institute at the University of Potsdam, 2013. (Cited on pages 59 and 155.)

[212] Andreas Meyer and Mathias Weske. Extracting data objects and their states from process models. In *Enterprise Distributed Object Computing (EDOC)*, pages 27–36. IEEE, 2013. (Cited on pages 15, 59, and 99.)

[213] Andreas Meyer and Mathias Weske. Weak conformance between process models and synchronized object life cycles. In *Service-Oriented Computing (ICSOC)*, pages 359–367. Springer, 2014. (Cited on pages 15, 59, and 123.)

[214] Andreas Meyer and Mathias Weske. Weak conformance between process models and synchronized object life cycles. Technical report 91, Hasso Plattner Institute at the University of Potsdam, 2014. (Cited on pages 59 and 123.)

[215] Andreas Meyer, Sergey Smirnov, and Mathias Weske. Data in business processes. *Enterprise Modelling and Information Systems Architectures (EMISA) Forum*, 31(3):5–31, 2011. (Cited on pages 13 and 59.)

[216] Andreas Meyer, Sergey Smirnov, and Mathias Weske. Data in business processes. Technical report 50, Hasso Plattner Institute at the University of Potsdam, 2011. (Cited on pages 25 and 59.)

[217] Andreas Meyer, Artem Polyvyanyy, and Mathias Weske. Weak conformance of process models with respect to data objects. In *Services and their Composition (ZEUS)*, pages 74–80. CEUR-WS, 2012. (Cited on pages 59 and 123.)

[218] Andreas Meyer, Luise Pufahl, Kimon Batoulis, Sebastian Kruse, Thorben Lindhauer, Thomas Stoff, Dirk Fahland, and Mathias Weske. Data perspective in process choreographies: Modeling and execution. Technical report BPM-13-29, BPMcenter.org, 2013. (Cited on pages 59 and 195.)

[219] Andreas Meyer, Luise Pufahl, Dirk Fahland, and Mathias Weske. Enacting complex data dependencies from activity-centric business process models. In *Business Process Management (BPM) Demos*, pages 11–15. CEUR-WS, 2013. (Cited on pages 59 and 195.)

[220] Andreas Meyer, Luise Pufahl, Dirk Fahland, and Mathias Weske. Modeling and enacting complex data dependencies in business processes. In *Business Process Management (BPM)*, pages 171–186. Springer, 2013. (Cited on pages 15, 59, and 195.)

[221] Andreas Meyer, Luise Pufahl, Dirk Fahland, and Mathias Weske. Modeling and enacting complex data dependencies in business processes. Technical report 74, Hasso Plattner Institute at the University of Potsdam, 2013. (Cited on pages 59 and 195.)

[222] Andreas Meyer, Nico Herzberg, Frank Puhlmann, and Mathias Weske. Implementation framework for production case management: Modeling and execution. In *Enterprise Distributed Object Computing (EDOC)*, pages 190–199. IEEE, 2014. (Cited on pages 305 and 316.)

[223] Andreas Meyer, Luise Pufahl, Kimon Batoulis, Sebastian Kruse, Thorben Lindhauer, Thomas Stoff, Dirk Fahland, and Mathias Weske. Automating data exchange in process choreographies. In *Advanced Information Systems Engineering (CAiSE)*, pages 316–331. Springer, 2014. (Cited on pages 15, 59, and 195.)

[224] Andreas Meyer, Luise Pufahl, Kimon Batoulis, Dirk Fahland, and Mathias Weske. Automating data exchange in process choreographies. *Information Systems*, 53:296–329, 2015. (Cited on pages 15, 59, and 195.)

[225] Bertrand Meyer. *Introduction to the theory of programming languages*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1990. (Cited on pages 86 and 95.)

[226] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, I. *Information and Computation*, 100(1):1–40, 1992. (Cited on page 95.)

[227] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, II. *Information and Computation*, 100(1):41–77, 1992. (Cited on page 95.)

[228] Simon Moser, Axel Martens, Katharina Gorlach, Wolfram Amme, and Artur Godlinski. Advanced verification of distributed WS-BPEL business processes incorporating CSSA-based data flow analysis. In *Services Computing (SCC)*, pages 98–105. IEEE, 2007. (Cited on page 151.)

[229] Gero Mühl, Ludger Fiege, and Peter Pietzuch. *Distributed event-based systems*. Springer, 2006. (Cited on page 24.)

[230] Dominic Müller. *Management datengetriebener Prozessstrukturen*. PhD thesis, University of Ulm, 2009. (Cited on page 11.)

[231] Dominic Müller, Manfred Reichert, and Joachim Herbst. Data-driven modeling and coordination of large process structures. In *On the Move to Meaningful Internet Systems (OTM)*, pages 131–149. Springer, 2007. (Cited on pages 60, 94, 122, 156, 190, 199, 303, and 304.)

[232] Dominic Müller, Manfred Reichert, and Joachim Herbst. A new paradigm for the enactment and dynamic adaptation of data-driven process structures. In *Advanced Information Systems Engineering (CAiSE)*, pages 48–63. Springer, 2008. (Cited on pages 26, 95, and 199.)

[233] Tadao Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, 1989. (Cited on pages 42, 49, 86, 91, 92, and 95.)

[234] Bela Mutschler and Manfred Reichert. Aktuelles Schlagwort: Business Process Intelligence. *Enterprise Modelling and Information Systems Architectures (EMISA) Forum*, 26(1):27–31, 2006. (Cited on page 9.)

[235] Shiva Nejati, Mehrdad Sabetzadeh, Marsha Chechik, Steve Easterbrook, and Pamela Zave. Matching and merging of statecharts specifications. In *International Conference on Software Engineering*, pages 54–64. IEEE Computer Society, 2007. (Cited on pages 80, 151, and 189.)

[236] Marcel F. Neuts. A general class of bulk queues with Poisson input. *The Annals of Mathematical Statistics*, 38(3):759–770, 1967. (Cited on page 314.)

[237] Anil Nigam and Nathan S. Caswell. Business artifacts: An approach to operational specification. *IBM Systems Journal*, 42(3):428–445, 2003. (Cited on pages 70, 94, 121, 122, 156, 190, 204, 256, and 303.)

[238] Fritz Nordsieck. *Die schaubildliche Erfassung und Untersuchung der Betriebsorganisation*. Poeschel, 1932. (Cited on page 3.)

[239] Natalya F. Noy. Semantic integration: A survey of ontology-based approaches. *ACM Sigmod Record*, 33(4):65–70, 2004. (Cited on page 307.)

[240] OASIS. Web services business process execution language, Version 2.0, April 2007. URL http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html. (Cited on pages 11, 252, 254, 262, and 306.)

[241] Object Management Group. Model driven architecture (MDA), 2001. URL http://www.omg.org/mda/. (Cited on page 307.)

[242] Object Management Group. Business Process Modeling Notation (BPMN), Version 1.0, OMG Final Adopted Specification, February 2006. URL http://www.bpmn.org/. (Cited on page 86.)

[243] Object Management Group. Business Process Model and Notation (BPMN), Version 2.0, January 2011. URL http://www.omg.org/spec/BPMN/2.0/. (Cited on pages 10, 11, 26, 41, 74, 86, 89, 158, 170, 190, 198, 200, 209, 248, 254, 260, 262, 284, 303, 304, and 306.)

[244] Object Management Group. Unified Modeling Language (UML), Version 2.4.1, August 2011. URL http://www.omg.org/spec/UML/2.4.1/. (Cited on pages 11, 63, 86, 150, 203, 250, and 253.)

[245] Object Management Group. Case Management Model and Notation (CMMN), Version 1.0, January 2013. URL http://www.omg.org/spec/CMMN/. (Cited on pages 11, 44, 46, 156, and 199.)

[246] Avner Ottensooser, Alan Fekete, Hajo A. Reijers, Jan Mendling, and Con Menictas. Making sense of business process descriptions: An experimental comparison of graphical and textual notations. *Journal of Systems and Software*, 85(3):596–606, 2012. (Cited on page 96.)

[247] Chun Ouyang, Eric Verbeek, Wil M. P. van der Aalst, Stephan Breutel, Marlon Dumas, and Arthur H. M. ter Hofstede. Formal semantics and analysis of control flow in WS-BPEL. *Science of Computer Programming*, 67(2):162–198, 2007. (Cited on pages 23 and 95.)

[248] Chun Ouyang, Marlon Dumas, Arthur H. M. ter Hofstede, and Wil M. P. van der Aalst. Pattern-based translation of BPMN process models to BPEL web services. *International Journal of Web Services Research (IJWSR)*, 5(1):42–62, 2008. (Cited on page 96.)

[249] M. Tamer Özsu and Patrick Valduriez. *Principles of distributed database systems*. Springer, 2011. (Cited on page 306.)

[250] Mike P. Papazoglou and Willem-Jan van den Heuvel. Service oriented architectures: Approaches, technologies and research issues. *VLDB Journal*, 16(3):389–415, 2007. (Cited on page 9.)

[251] Chris Peltz. Web services orchestration and choreography. *Computer*, 36 (10):46–52, 2003. (Cited on page 249.)

[252] Edith Tilton Penrose. *The theory of the growth of the firm*. Wiley, 1959. (Cited on page 9.)

[253] Carl A. Petri. *Kommunikation mit Automaten*. PhD thesis, Institut für Instrumentelle Mathematik, University of Bonn, 1962. (Cited on pages 15, 42, 49, and 86.)

[254] Artem Polyvyanyy and Matthias Weidlich. Towards a compendium of process technologies: The jBPT library for Pprocess model analysis. In *Advanced Information Systems Engineering (CAiSE) Forum*, volume 998, pages 106–113. CEUR Workshop Proceedings, 2013. (Cited on pages 6 and 113.)

[255] Artem Polyvyanyy, Sergey Smirnov, and Mathias Weske. The triconnected abstraction of process models. In *Business Process Management (BPM)*, pages 229–244. Springer, 2009. (Cited on pages 23, 181, and 315.)

[256] Artem Polyvyanyy, Sergey Smirnov, and Mathias Weske. Business process model abstraction. In *Handbook on Business Process Management 1*, pages 149–166. Springer, 2010. (Cited on page 315.)

[257] Artem Polyvyanyy, Luciano García-Bañuelos, and Marlon Dumas. Structuring acyclic process models. *Information Systems*, 37(6):518–538, 2012. (Cited on page 6.)

[258] Günter Preuner, Stefan Conrad, and Michael Schrefl. View integration of behavior in object-oriented databases. *Data & Knowledge Engineering*, 36(2):153–183, 2001. (Cited on pages 80, 151, and 189.)

[259] Luise Pufahl and Mathias Weske. Batch activities in process modeling and execution. In *Service-Oriented Computing (ICSOC)*, pages 283–297. Springer, 2013. (Cited on page 314.)

[260] Luise Pufahl, Andreas Meyer, and Mathias Weske. Batch regions: Process instance synchronization based on data. Technical report 86, Hasso Plattner Institute at the University of Potsdam, 2013. (Cited on pages 59 and 314.)

[261] Luise Pufahl, Andreas Meyer, and Mathias Weske. Batch regions: Process instance synchronization based on data. In *Enterprise Distributed Object Computing (EDOC)*, pages 150–159. IEEE, 2014. (Cited on pages 59 and 314.)

[262] Frank Puhlmann and Mathias Weske. Investigations on soundness regarding lazy activities. In *Business Process Management (BPM)*, pages 145–160. Springer Berlin / Heidelberg, 2006. (Cited on pages 95 and 266.)

[263] Frank Puhlmann and Mathias Weske. A look around the corner: The pi-calculus. In *Transactions on Petri Nets and Other Models of Concurrency II*, pages 64–78. Springer, 2009. (Cited on page 95.)

[264] Zongyan Qiu, Xiangpeng Zhao, Chao Cai, and Hongli Yang. Towards the theoretical foundation of choreography. In *World Wide Web*, pages 973–982. ACM, 2007. (Cited on pages 81 and 266.)

[265] Erhard Rahm and Philip A. Bernstein. A survey of approaches to automatic schema matching. *The VLDB Journal*, 10(4):334–350, 2001. (Cited on page 307.)

[266] Jan C. Recker and Jan Mendling. On the translation between BPMN and BPEL: Conceptual mismatch between process modeling languages. In *Proceedings of Workshops and Doctoral Consortium of the International Conference on Advanced Information Systems Engineering (CAiSE)*, pages 521–532. Namur University Press, 2006. (Cited on page 96.)

[267] Guy Redding, Marlon Dumas, Arthur H. M. ter Hofstede, and Adrian Iordachescu. Transforming object-oriented models to process-oriented models. In *Business Process Management (BPM) Workshops*, pages 132–143. Springer, 2008. (Cited on page 60.)

[268] Guy Redding, Marlon Dumas, Arthur H. M. ter Hofstede, and Adrian Iordachescu. A flexible, object-centric approach for business process modelling. *Service Oriented Computing and Applications (SOCA)*, 4(3):191–201, 2010. (Cited on pages 60, 303, and 304.)

[269] Manfred Reichert. Process and data: Two sides of the same coin? In *On the Move to Meaningful Internet Systems (OTM)*, pages 2–19. Springer, 2012. (Cited on page 60.)

[270] Manfred Reichert, Stefanie Rinderle-Ma, and Peter Dadam. Flexibility in process-aware information systems. *Transactions on Petri Nets and Other Models of Concurrency (ToPNoC)*, 5460:115–135, 2009. (Cited on pages 303, 305, and 315.)

[271] Hajo A. Reijers. *Design and control of workflow processes: Business process management for the service industry*. Springer, 2003. (Cited on pages 7 and 25.)

[272] Hajo A. Reijers and S. Liman Mansar. Best practices in business process redesign: An overview and qualitative evaluation of successful redesign heuristics. *Omega*, 33(4):283–306, 2005. (Cited on page 25.)

[273] Hajo A. Reijers, Selma Limam, and Wil M. P. van der Aalst. Product-based workflow design. *Management Information Systems*, 20(1):229–262, 2003. (Cited on page 60.)

[274] Wolfgang Reisig. *Petri nets: An introduction*, volume 4 of *Monographs in Theoretical Computer Science*. Springer, 1985. (Cited on page 49.)

[275] Wolfgang Reisig. *Petrinetze: Modellierungstechnik, Analysemethoden, Fallstudien*. Vieweg+Teubner, 2010. (Cited on page 49.)

[276] Wolfgang Reisig and Grzegorz Rozenberg. *Lectures on Petri nets I: Basic models, advances in Petri nets*. Springer, 1998. (Cited on page 49.)

[277] Clay Richardson. Warning: Don't assume your business processes use master data. In *Business Process Management (BPM)*, pages 11–12. Springer, 2010. (Cited on pages 13 and 60.)

[278] Stefanie Rinderle and Manfred Reichert. Data – driven process control and exception handling in process management systems. In *Advanced Information Systems Engineering (CAiSE)*, pages 273–287. Springer, 2006. (Cited on page 60.)

[279] Stefanie Rinderle-Ma, Linh Thao Ly, and Peter Dadam. Business process compliance. In *Enterprise Modelling and Information Systems Architectures (EMISA) Forum*, pages 24–29. Gesellschaft für Informatik e.V. (GI), 2008. (Cited on page 23.)

[280] Andreas Rogge-Solti and Mathias Weske. Enabling probabilistic process monitoring in non-automated environments. In *Enterprise, Business-Process and Information Systems Modeling (BPMDS/EMMSAD)*, pages 226–240. Springer, 2012. (Cited on page 313.)

[281] Andreas Rogge-Solti, Matthias Kunze, Ahmed Awad, and Mathias Weske. Business process configuration wizard and consistency checker for BPMN 2.0. In *Enterprise, Business-Process and Information Systems Modeling (BPMDS/EMMSAD)*, pages 231–245. Springer, 2011. (Cited on pages 100 and 152.)

[282] Colette Rolland and Naveen Prakash. Bridging the gap between organisational needs and ERP functionality. *Requirements Engineering*, 5(3): 180–193, 2000. (Cited on pages 8 and 23.)

[283] Colette Rolland, Carine Souveyet, and Camille Ben Achour. Guiding goal modeling using scenarios. *Transactions on Software Engineering*, 24 (12):1055–1071, 1998. (Cited on page 8.)

[284] Andrew William Roscoe, Charles A. R. Hoare, and Richard Bird. *The theory and practice of concurrency*, volume 169. Prentice-Hall, 1997. (Cited on page 95.)

[285] Michael Rosemann and Wil M. P. van der Aalst. A configurable reference modelling language. *Information Systems*, 32(1):1–23, 2007. (Cited on page 315.)

[286] Anne Rozinat and Wil M. P. van der Aalst. Conformance checking of processes based on monitoring real behavior. *Information Systems*, 33(1): 64–95, 2008. (Cited on page 150.)

[287] Andreas Rulle and Juliane Siegeris. From a family of state-centric PAIS to a configurable and parameterized business process architecture. In *Business Process Management (BPM)*, pages 333–348. Springer, 2014. (Cited on pages 22, 153, 181, and 314.)

[288] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William E. Lorensen. *Object-oriented modeling and design*. Prentice-Hall, Upper Saddle River, NJ, USA, 1991. (Cited on pages 11, 61, and 266.)

[289] Nick Russell, Arthur H. M. ter Hofstede, David Edmond, and Wil M. P. van der Aalst. Workflow data patterns. Technical report, Queensland University of Technology, 2004. (Cited on page 13.)

[290] Nick Russell, Arthur H. M. Ter Hofstede, David Edmond, and Wil M. P. van der Aalst. Workflow resource patterns. BETA Working Paper Series WP 127, Eindhoven University of Technology, 2004. (Cited on pages 4, 40, and 69.)

[291] Nick Russell, Wil M. P. van der Aalst, Arthur H. M. ter Hofstede, and Petia Wohed. On the suitability of UML 2.0 activity diagrams for business process modelling. In *Conceptual Modeling (ER)*, pages 95–104. Australian Computer Society, Inc., 2006. (Cited on page 13.)

[292] Ksenia Ryndina, Jochen Küster, and Harald Gall. Consistency of business process models and object life cycles. In *MoDELS Workshops*, pages 80–90. Springer, 2006. (Cited on pages 95, 100, 121, 124, 126, 149, 152, 156, 191, and 199.)

[293] Shazia Sadiq, Maria E. Orlowska, Wasim Sadiq, and Cameron Foulger. Data flow and validation in workflow modelling. In *Australasian Database Conference*, pages 207–214. Australian Computer Society, 2004. (Cited on pages 142 and 152.)

[294] Shazia Sadiq, Maria Orlowska, Wasim Sadiq, and Karsten Schulz. When workflows will not deliver: The case of contradicting work practice. In *Business Information Systems (BIS)*, pages 69–84, 2005. (Cited on page 314.)

[295] Shazia Sadiq, Guido Governatori, and Kioumars Namiri. Modeling control objectives for business process compliance. In *Business Process Management (BPM)*, pages 149–164. Springer, 2007. (Cited on page 149.)

[296] Gwen Salaün and Tevfik Bultan. Realizability of choreographies using process algebra encodings. In *Integrated Formal Methods*, pages 167–182. Springer, 2009. (Cited on page 266.)

[297] Partha Sampath and Martin Wirsing. Computing the cost of business processes. In *Information Systems: Modeling, Development, and Integration*, pages 178–183. Springer, 2009. (Cited on page 23.)

[298] Beatrice Santorini. Part-of-speech tagging guidelines for the Penn Treebank Project (3rd revision). Technical report MS-CIS-90-47, University of Pennsylvania, 1990. (Cited on pages 104 and 114.)

[299] Sigrid Schefer, Mark Strembeck, Jan Mendling, and Anne Baumgrass. Detecting and resolving conflicts of mutual-exclusion and binding constraints in a business process context. In *On the Move to Meaningful Internet Systems (OTM)*, pages 329–346. Springer, 2011. (Cited on page 4.)

[300] Hermann J. Schmelzer and Wolfgang Sesselmann. *Geschäftsprozessmanagement in der Praxis*. Hanser, 2001. (Cited on page 3.)

[301] Karsten Schmidt. LoLA A Low Level Analyser. In *Application and Theory of Petri Nets (ICATPN)*, pages 465–474. Springer, 2000. (Cited on pages 139 and 318.)

[302] Alec Sharp and Patrick McDermott. *Workflow modeling: Tools for process improvement and applications development*. Artech House, 2009. (Cited on page 120.)

[303] Pavel Shvaiko and Jérôme Euzenat. A survey of schema-based matching approaches. *Journal on Data Semantics IV*, 3730:146–171, 2005. (Cited on page 307.)

[304] Natalia Sidorova, Christian Stahl, and Nikola Trčka. Soundness verification for conceptual workflow nets with data: Early detection of errors with the most precision possible. *Information Systems*, 36(7):1026–1043, 2011. (Cited on pages 96 and 150.)

[305] Abraham Silberschatz, Henry F. Korth, and S. Sudarshan. *Database system concepts*. McGraw-Hill Book Company, 2010. (Cited on pages 200, 202, and 303.)

[306] Bruce Silver. *BPMN method and style (Second Edition)*. Cody-Cassidy Press, 2009. (Cited on page 26.)

[307] Sergey Smirnov, Hajo A. Reijers, Thijs Nugteren, and Mathias Weske. Business process model abstraction: Theory and practice. Technical Report 35, Hasso Plattner Institute at the University of Potsdam, 2010. (Cited on page 315.)

[308] Sergey Smirnov, Hajo A. Reijers, and Mathias Weske. A semantic approach for business process model abstraction. In *Advanced Information Systems Engineering (CAiSE)*, pages 497–511. Springer, 2011. (Cited on page 315.)

[309] Sergey Smirnov, Hajo A. Reijers, and Mathias Weske. From fine-grained to abstract process models: A semantic approach. *Information Systems*, 37(8):784–797, 2012. (Cited on pages 23, 181, and 315.)

[310] Adam Smith. *An inquiry into the nature and causes of the wealth of nations*. Strahan, 1776. (Cited on page 3.)

[311] Howard Smith and Peter Fingar. *Business process management: The third wave*. Meghan-Kiffer Press, 2003. (Cited on pages 3, 4, 23, and 25.)

[312] Martín Soto, Alexis Ocampo, and Jürgen Münch. The secret life of a process description: A look into the evolution of a large process model. In *Making Globally Distributed Software Development a Success Story, Software Processes (ICSP)*, pages 257–268. Springer, 2008. (Cited on page 116.)

[313] Herbert Stachowiak. *Allgemeine Modelltheorie*. Springer, 1973. (Cited on pages 3 and 21.)

[314] Silvia von Stackelberg, Susanne Putze, Jutta Mülle, and Klemens Böhm. Detecting data-flow errors in BPMN 2.0. Technical report, Karlsruhe Institute of Technology, 2014. (Cited on pages 96 and 153.)

[315] Mark Strembeck and Jan Mendling. Modeling process-related RBAC models with extended UML activity models. *Information and Software Technology*, 53(5):456–483, 2011. (Cited on page 4.)

[316] Sherry X. Sun, J. Leon Zhao, Jay F. Nunamaker, and Olivia R. L. Sheng. Formulating the data-flow perspective for business process management. *Information Systems Research*, 17(4):374–391, 2006. (Cited on pages 142 and 152.)

[317] Keith D. Swenson. State of the art in case management. Technical report, Fujitsu, March 2013. (Cited on page 315.)

[318] Andrew S. Tanenbaum and Maarten van Steen. *Distributed systems: Principles and paradigms*. Prentice Hall, 2006. (Cited on page 306.)

[319] Arthur H. M. ter Hofstede and Henderik A. Proper. How to formalize it?: Formalization principles for information system development methods. *Information and Software Technology*, 40(10):519–540, 1998. (Cited on pages 86 and 95.)

[320] Arthur H. M. ter Hofstede, Wil M. P. van der Aalst, Michael Adams, and Nick Russell. *Modern business process automation: YAWL and its support environment*. Springer Science & Business Media, 2009. (Cited on pages 4 and 23.)

[321] P. S. Thiagarajan and Klaus Voss. In praise of free choice nets. In *Advances in Petri Nets*, pages 438–454. Springer, 1984. (Cited on page 52.)

[322] Anthony Tomasic, Louiqa Raschid, and Patrick Valduriez. Scaling access to heterogeneous data sources with DISCO. *IEEE Transactions on Knowledge and Data Engineering*, 10(5):808–823, 1998. (Cited on page 306.)

[323] Kristina Toutanova and Christopher D. Manning. Enriching the knowledge sources used in a maximum entropy part-of-speech tagger. In *Empirical methods in natural language processing and very large corpora (EMNLP/VLC)*, pages 63–70. Association for Computational Linguistics, 2000. (Cited on pages 104, 113, and 121.)

[324] Kristina Toutanova, Dan Klein, Christopher D. Manning, and Yoram Singer. Feature-rich part-of-speech tagging with a cyclic dependency network. In *North American Chapter of the Association for Computational Linguistics on Human Language Technology (NAACL)*, pages 173–180, 2003. (Cited on pages 104, 113, and 121.)

[325] Nikola Trčka, Wil M. P. van der Aalst, and Natalia Sidorova. Data-flow anti-patterns: Discovering data-flow errors in workflows. In *Advanced Information Systems Engineering (CAiSE)*, pages 425–439. Springer, 2009. (Cited on pages 26, 96, and 150.)

[326] C.-Y. Tsai, James J. H. Liou, and T.-M. Huang. Using a multiple-GA method to solve the batch picking problem: Considering travel distance and order due time. *International Journal of Production Research*, 46(22): 6533–6555, 2008. (Cited on page 314.)

[327] Willi Tscheschner. Transformation from EPC to BPMN. *Business Process Technology*, 1(3):7–21, 2006. (Cited on page 96.)

[328] Sebastian Uchitel, Jeff Kramer, and Jeff Magee. Incremental elaboration of scenario-based specifications and behavior models using implied scenarios. *ACM Transactions on Software Engineering and Methodology*, 13(1): 37–85, 2004. (Cited on pages 80, 151, and 189.)

[329] Patrick Valduriez and Esther Pacitti. Data management in large-scale P2P systems. In *High Performance Computing for Computational Science (VECPAR)*, pages 104–118. Springer, 2005. (Cited on page 306.)

[330] Franck van Breugel and Maria Koshkina. Models and verification of BPEL. *Monograph on Testing & Analysis of Web Services*, 2006. (Cited on page 96.)

[331] Wil M. P. van der Aalst. Verification of workflow nets. In *Application and Theory of Petri Nets (ICATPN)*, pages 407–426. Springer, 1997. (Cited on pages 6, 23, 52, 53, 86, 90, 123, 136, 138, 265, 266, 272, and 276.)

[332] Wil M. P. van der Aalst. The application of Petri nets to workflow management. *Circuits, Systems, and Computers*, 8:21–66, 1998. (Cited on pages 26, 52, and 271.)

[333] Wil M. P. van der Aalst. Formalization and verification of event-driven process chains. *Information and Software Technology*, 41(10):639–650, 1999. (Cited on page 95.)

[334] Wil M. P. van der Aalst. Workflow verification: Finding control-flow errors using Petri-net-based techniques. In *Business Process Management (BPM)*, pages 161–183. Springer, 2000. (Cited on pages 131, 141, 266, and 275.)

[335] Wil M. P. van der Aalst. *Process mining - discovery, conformance and enhancement of business processes*. Springer, 2011. (Cited on pages 6 and 25.)

[336] Wil M. P. van der Aalst. Process mining: Overview and opportunities. *Transactions on Management Information Systems (TMIS)*, 3(2):7:1—7:17, 2012. (Cited on pages 7 and 25.)

[337] Wil M. P. van der Aalst and Twan Basten. Identifying commonalities and differences in object life cycles using behavioral inheritance. In *Application and Theory of Petri Nets (ICATPN)*, pages 32–52. Springer, 2001. (Cited on page 187.)

[338] Wil M. P. van der Aalst and Twan Basten. Inheritance of workflows: An approach to tackling problems related to change. *Theoretical Computer Science*, 270(1):125–203, 2002. (Cited on page 273.)

[339] Wil M. P. van der Aalst and Arthur H. M. ter Hofstede. Verification of workflow task structures: A Petri-net-based approach. *Information systems*, 25(1):43–69, 2000. (Cited on page 95.)

[340] Wil M. P. van der Aalst and Arthur H. M. ter Hofstede. YAWL: Yet Another Workflow Language. *Information Systems*, 30(4):245–275, 2005. (Cited on pages 11 and 303.)

[341] Wil M. P. van der Aalst and Kees van Hee. *Workflow management: Models, methods, and systems*. MIT press, 2002. (Cited on pages 4 and 5.)

[342] Wil M. P. van der Aalst and Mathias Weske. The P2P approach to interorganizational workflows. In *Advanced Information Systems Engineering (CAiSE)*, pages 140–156. Springer, 2001. (Cited on pages 248, 249, 250, 253, 255, 272, and 311.)

[343] Wil M. P. van der Aalst, Paulo Barthelmess, Clarence A. Ellis, and Jacques Wainer. Workflow modeling using Proclets. In *On the Move to Meaningful Internet Systems (OTM)*, pages 198–209. Springer, 2000. (Cited on page 60.)

[344] Wil M. P. van der Aalst, Paulo Barthelmess, Clarence A. Ellis, and Jacques Wainer. Proclets: A framework for lightweight interacting workflow processes. *International Journal of Cooperative Information Systems*, 10 (4):443–481, 2001. (Cited on pages 94, 152, 199, 303, 304, and 315.)

[345] Wil M. P. van der Aalst, Arthur H. M. ter Hofstede, Bartek Kiepuszewski, and Alistair P. Barros. Workflow patterns. *Distributed and Parallel Databases*, 14(1):5–51, 2003. (Cited on pages 4, 27, and 95.)

[346] Wil M. P. van der Aalst, Arthur H. M. ter Hofstede, and Mathias Weske. Business process management: A survey. In *Business Process Management (BPM)*, pages 1–12. Springer, 2003. (Cited on pages 4, 5, 23, and 316.)

[347] Wil M. P. van der Aalst, Mathias Weske, and Dolf Grünbauer. Case handling: A new paradigm for business process support. *Data & Knowledge Engineering*, 53(2):129–162, 2005. (Cited on pages 26, 60, 191, 303, 304, and 315.)

[348] Wil M. P. van der Aalst, K.M. van Hee, Arthur H. M. ter Hofstede, N. Sidorova, H.M.W. Verbeek, M. Voorhoeve, and M.T. Wynn. Soundness of workflow nets: Classification, decidability, and analysis. Technical report, Eindhoven University of Technology, 2008. (Cited on page 6.)

[349] Wil M. P. van der Aalst, Maja Pesic, and Helen Schonenberg. Declarative workflows: Balancing between flexibility and support. *Computer Science-Research and Development*, 23(2):99–113, 2009. (Cited on page 315.)

[350] Wil M. P. van der Aalst, Niels Lohmann, Peter Massuthe, Christian Stahl, and Karsten Wolf. Multiparty contracts: Agreeing and implementing interorganizational processes. *The Computer Journal*, 53(1):90–106, 2010. (Cited on pages 24, 249, and 306.)

[351] Wil M. P. van der Aalst, Kees M. van Hee, Arthur H. M. ter Hofstede, Natalia Sidorova, H. M. W. Verbeek, Marc Voorhoeve, and Moe T. Wynn. Soundness of workflow nets: Classification, decidability, and analysis. *Formal Aspects of Computing*, 23(3):333–363, 2011. (Cited on page 55.)

[352] Wil M. P. van der Aalst, Arya Adriansyah, and Boudewijn F. van Dongen. Replaying history on process models for conformance checking and performance analysis. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 2(2):182–192, 2012. (Cited on page 151.)

[353] Rob J. van Glabbeek and W. Peter Weijland. Branching time and abstraction in bisimulation semantics. *Journal of the ACM (JACM)*, 43(3):555–600, 1996. (Cited on page 273.)

[354] Kees van Hee, Natalia Sidorova, Lou Somers, and Marc Voorhoeve. Consistency in model integration. *Data & Knowledge Engineering*, 56(1):4–22, 2006. (Cited on page 151.)

[355] Axel van Lamsweerde and Emmanuel Letier. From object orientation to goal orientation: A paradigm shift for requirements engineering. In *Radical Innovations of Software and Systems Engineering in the Future*, pages 325–340. Springer, 2004. (Cited on page 8.)

[356] Irene Vanderfeesten, Hajo A. Reijers, and Wil M. P. van der Aalst. Product-based workflow support. *Information Systems*, 36(2):517–535, 2011. (Cited on pages 60, 191, 303, and 304.)

[357] Jussi Vanhatalo, Hagen Völzer, Frank Leymann, and Simon Moser. Automatic workflow graph refactoring and completion. In *Service-Oriented Computing (ICSOC)*, pages 100–115. Springer, 2008. (Cited on page 53.)

[358] Jussi Vanhatalo, Hagen Völzer, and Jana Koehler. The refined process structure tree. *Data & Knowledge Engineering*, 68(9):793–818, 2009. (Cited on page 90.)

[359] Hagen Völzer. A new semantics for the inclusive converging gateway in safe processes. In *Business Process Management (BPM)*, pages 294–309. Springer, 2010. (Cited on page 28.)

[360] Holger Wache, Thomas Voegele, Ubbo Visser, Heiner Stuckenschmidt, Gerhard Schuster, Holger Neumann, and Sebastian Hübner. Ontology-based integration of information – a survey of existing approaches. In *Ontologies and information sharing (IJCAI) workshop*, volume 2001, pages 108–117, 2001. (Cited on page 307.)

[361] Ksenia Wahler and Jochen Küster. Predicting coupling of object-centric business process implementations. In *Business Process Management (BPM)*, pages 148–163. Springer, 2008. (Cited on page 95.)

[362] Yair Wand and Ron Weber. Research commentary: Information systems and conceptual modeling – a research agenda. *Information Systems Research*, 13(4):363–376, 2002. (Cited on page 21.)

[363] Jianrui Wang and Akhil Kumar. A framework for document-driven workflow systems. In *Business Process Management (BPM)*, pages 285–301. Springer, 2005. (Cited on pages 26, 191, and 303.)

[364] Zhaoxia Wang, Arthur H. M. ter Hofstede, Chun Ouyang, Moe Wynn, Jianmin Wang, and Xiaochen Zhu. How to guarantee compliance between workflows and product lifecycles? Technical report BPM-11-10, BPMcenter.org, 2011. (Cited on pages 100, 126, 149, 150, and 153.)

[365] Barbara Weber, Manfred Reichert, and Stefanie Rinderle-Ma. Change patterns and change support features – enhancing flexibility in process-aware information systems. *Data & Knowledge Engineering*, 66(3):438–466, 2008. (Cited on page 315.)

[366] Ingo Weber, Jörg Hoffmann, and Jan Mendling. Beyond soundness: On the verification of semantic business process models. *Distributed and Parallel Databases*, 27(3):271–343, 2010. (Cited on page 151.)

[367] Matthias Weidlich, Gero Decker, Alexander Groß kopf, and Mathias Weske. BPEL to BPMN: The myth of a straight-forward mapping. In *On the Move to Meaningful Internet Systems (OTM)*, pages 265–282. Springer, 2008. (Cited on pages 23 and 96.)

[368] Matthias Weidlich, Mathias Weske, and Jan Mendling. Change propagation in process models using behavioural profiles. In *Services Computing (SCC)*, pages 33–40. IEEE, 2009. (Cited on pages 48 and 49.)

[369] Matthias Weidlich, Jan Mendling, and Mathias Weske. Efficient consistency measurement based on behavioral profiles of process models. *IEEE Transactions on Software Engineering*, 37(3):410–429, 2011. (Cited on pages 6 and 48.)

[370] Mathias Weske. *Business process management: Concepts, languages, architectures (Second Edition)*. Springer, 2012. (Cited on pages 3, 4, 5, 6, 7, 8, 9, 13, 21, 22, 23, 26, 37, 73, 74, 101, 190, 196, 248, 260, and 278.)

[371] Stephen White. Using BPMN to model a BPEL process. *BPTrends*, 3(3): 1–18, 2005. (Cited on page 23.)

[372] Stephen A. White and Derek Miers. *BPMN modeling and reference guide: Understanding and using BPMN*. Future Strategies Inc., 2008. (Cited on page 26.)

[373] Gio Wiederhold. Mediators in the architecture of future information systems. *Computer*, 25(3):38–49, 1992. (Cited on page 307.)

[374] Petia Wohed, Wil M. P. van der Aalst, Marlon Dumas, Arthur H. M. ter Hofstede, and Nick Russell. On the suitability of BPMN for business process modelling. In *Business Process Management (BPM)*, pages 161–176. Springer, 2006. (Cited on page 13.)

[375] Karsten Wolf. Generating Petri net state spaces. In *Petri Nets and Other Models of Concurrency (ICATPN)*, pages 29–42. Springer, 2007. (Cited on pages 139 and 318.)

[376] Peter Y. H. Wong and Jeremy Gibbons. A process semantics for BPMN. In *Formal Methods and Software Engineering (ICFEM)*, pages 355–374. Springer, 2008. (Cited on page 95.)

[377] World Wide Web Consortium. Web Services Description Language (WSDL) 1.1, March 2001. URL http://www.w3.org/TR/wsdl. (Cited on pages 252 and 253.)

[378] World Wide Web Consortium. OWL Web Ontology Language, February 2004. URL http://www.w3.org/TR/owl-features/. (Cited on page 307.)

[379] World Wide Web Consortium. Web Services Choreography Description Language, Version 1.0, November 2005. URL http://www.w3.org/TR/ws-cdl-10/. (Cited on pages 253 and 306.)

[380] World Wide Web Consortium. XQuery 1.0: An XML Query Language (Second Edition), December 2010. URL http://www.w3.org/TR/2010/REC-xquery-20101214/. (Cited on pages 251, 252, 253, 277, and 281.)

[381] World Wide Web Consortium. XML Schema Definition Language (XSD) 1.1, April 2012. URL http://www.w3.org/standards/techs/xmlschema. (Cited on page 280.)

[382] Moe Thandar Wynn, H. M. W. Verbeek, Wil M. P. van der Aalst, Arthur H. M. ter Hofstede, and David Edmond. Business process verification – finally a reality! *Business Process Management Journal*, 15(1):74–92, 2009. (Cited on page 123.)

[383] Daniel M. Yellin and Robert E. Strom. Protocol specifications and component adaptors. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 19(2):292–333, 1997. (Cited on page 307.)

[384] Sira Yongchareon, Chengfei Liu, and Xiaohui Zhao. A framework for behavior-consistent specialization of artifact-centric business processes. In *Business Process Management (BPM)*, pages 285–301. Springer, 2012. (Cited on pages 44, 45, 94, 156, and 190.)

[385] Eric Yu. *Modelling strategic relationships for process reengineering*. PhD thesis, University of Toronto, 1995. (Cited on page 8.)

[386] Johannes Maria Zaha, Alistair Barros, Marlon Dumas, and Arthur H. M. ter Hofstede. Let's dance: A language for service behavior modeling. In *On the Move to Meaningful Internet Systems (OTM)*, pages 145–162. Springer, 2006. (Cited on page 306.)

[387] Johannes Maria Zaha, Marlon Dumas, Arthur H. M. ter Hofstede, Alistair Barros, and Gero Decker. Service interaction modeling: Bridging global and local views. In *Enterprise Distributed Object Computing (EDOC)*, pages 45–55. IEEE, 2006. (Cited on page 272.)

[388] Michael zur Muehlen and Danny Ting-Yi Ho. Risk management in the BPM lifecycle. In *Business Process Management (BPM) Workshops*, pages 454–466. Springer, 2006. (Cited on page 23.)

[389] Michael zur Muehlen and Jan Recker. How much language is enough? Theoretical and practical use of the business process modeling notation. In *Advanced Information Systems Engineering (CAiSE)*, pages 465–479. Springer, 2008. (Cited on pages 39, 69, 101, and 111.)

All links were last followed on June 10, 2015.

LIST OF SYMBOLS

Functions

α	Value function (correlation data exchange automation).
β	Resource assignment to activity.
γ	Resource assignment to data node.
δ	Return of all data objects for given data node.
Δ	Database query.
η	Mapping object life cycle to data class.
θ	Schema mapping between global and local data models.
ι	Return object for given fully qualified data attribute.
κ	Probability of control flow edge.
$\varkappa$	Expression assignment to data flow edge.
λ	Probability of data node access.
μ	Label assignment.
ν	Guard for database queries.
ξ	Assignment of data condition to control flow edges originating from XOR split gateways.
ϖ	Assignment of control flow nodes to node life cycles.
Π	Synchronization validation function.
ρ	Mapping process model to business process.
ρ_{BP}	Mapping business process to process choreography.
ρ_{I}	Mapping process instance to process model.
ρ_{PM}	Mapping process model to process scenario.
ρ_{PS}	Mapping process scenario to business process.
φ_{D}	Mapping data node to data class.
φ_{O}	Mapping data object to data class.
ψ	Data view function.

$case$	Case object function.
$defined$	Defined function (data attribute of class defined, object-centric process models).
$instate$	Instate function (data class in state, object-centric process models).
$type_a$	Activity type assignment.
$type_d$	Instance multiplicity property assignment (data node).
$type_g$	Gateway type assignment.
$type_{op}$	Operation type assignment to data node.
$type_t$	Task type assignment.
z_i	Process instance state function.
s	Data state function.

Number sets

$\mathbb{N}$	Natural numbers.
$\mathbb{N}^+$	Positive natural numbers excluding zero.
$\mathbb{N}_0$	Natural numbers including zero.
$\mathbb{R}$	Real numbers.
$\mathbb{R}^+$	Positive real numbers excluding zero.
$\mathbb{R}_0^+$	Positive real numbers including zero.

Relations

$\succ$	Weak order relation.
$\rightsquigarrow$	Strict order relation.
$\|\|$	Interleaving relation.
$+$	Exclusiveness relation.
$\mathfrak{C}$	Control flow relation.
$\mathfrak{C}^+$	Transitive closure over the control flow relation.
$\mathfrak{E}$	Flow relation (Petri net).
$\mathfrak{F}$	Data flow relation.
$\mathfrak{M}$	Message flow relation.
$\mathfrak{P}$	Persistence relation.

$\mathfrak{R}$	Data relation.
$\mathfrak{R}_{Aggr}$	Aggregation (data relation).
$\mathfrak{R}_{Assoc}$	Undirected association (data relation).
$\mathfrak{R}_{Comp}$	Composition (data relation).
$\mathfrak{R}_{Gen}$	Generalization (data relation).
$\mathfrak{R}_{P}$	Parental data relations.
$\mathfrak{T}$	Transition.
$\mathfrak{T}_{NS}$	Node state transition relation.
$\mathfrak{T}_{S}$	Data state transition (object life cycle).

Single Entities, Sets, and Sequences

$\mathcal{A}$	Set of activities.
$\mathcal{AS}$	Schema of object-centric process model.
bp	Business process.
BP	Set of business processes.
br	business rule.
BR	Set of business rules.
c	Data class.
C	Set of data classes.
CI	Correlation identifier in message.
CS	Set of data classes in object-centric process model.
CT	Set of combined transitions.
d	Data node.
D	Set of data nodes.
db	Database.
DB	Set of databases.
DCF	Data-specific configurations of a process model.
dep	Dependency type (synchronization edge).
df	Data flow edge.
dm	Data model.

DS	Set of data stores.
E	Set of event models.
f	Petri net state or marking.
f_i	Initial marking (Petri net).
f_o	Final marking (Petri net).
F	Set of Petri net states or markings.
FK	Set of foreign keys.
FK^*	All-quantified foreign keys.
G	Set of gateways.
H	Set of data states in process instance.
i	Process instance.
I	Set of process instances.
id	Identifier.
J	Set of attributes of a data node.
J_M	Set of mandatory attributes of a data node.
J_O	Set of optional attributes of a data node.
$\mathcal{J}$	Set of run-time attributes of a data class.
k	Key (correlation identifier).
K	Map with data states plus id as key and lists of fully qualified data attributes as values.
l	Object life cycle.
L	Set of object life cycles.
$\mathcal{L}$	Synchronized object life cycle.
$label$	Task label in object-centric process model.
m	Process state or marking.
M	Set of process states or markings.
msg	Message in message flow.
N	Set of control flow nodes.
$name$	Name.

nl	Node life cycle.
NL	Set of node life cycles.
NS	Set of node states.
o	Data object.
O	Set of data objects.
ocp	Object-centric process model.
p	Place (Petri net).
p_i	Source place (Petri net).
p_o	Sink place (Petri net).
P	Set of places (Petri net).
pc	Process choreography.
PC	Set of process choreographies.
pf	Process fragment.
PF	Set of process fragments.
pid	Process instance id.
pk	Primary key.
PK	Set of primary keys.
pm	(Activity-centric) process model.
PM	Set of (activity-centric) process models.
pn	Petri net.
PN	Set of Petri nets.
$post$	Postcondition in in object-centric process model.
pre	Precondition in object-centric process model.
ps	Process scenario.
PS	Set of process scenarios.
Q	Set of activity labels.
R	Set of resources.
RC	Relation cluster.
s_i	Initial data state (object life cycle).

S	Set of data states.
S_F	Set of final data states (object life cycle).
$\mathcal{S}$	Net system.
se	Synchronization edge.
SE	Set of synchronization edges.
src	Source (synchronization edge).
SU	Subset of tasks (object-centric process model).
T	Sequence.
T_P	Set of traces (net system).
T_S	Sequence of data states.
T_Z	Sequence of process instance states.
$\mathcal{T}$	Set of transitions (Petri net).
tgt	Target (synchronization edge).
u	Task (object-centric process model).
U	Set of tasks (object-centric process model).
v	Data attribute value.
V	Universe of data attribute values.
w	Data view cluster.
W	Set of data view clusters.
X	Data view definition.
z	Process instance state.
Z	Set of process instance states.
$\mathcal{Z}$	Set of process instance states of one case.
σ	Execution sequence.
σ_A	Execution sequence of control flow nodes.
σ_S	Execution sequence of data states.
$\sigma_{\mathcal{T}}$	Execution sequence of transitions (Petri net).
Σ	Action (object life cycle).
τ	Empty/unspecified label for an action or activity.