WICKR

A Joint Semantics for Flexible Business Processes and Data

Stephan Haarmann

Business Process Technology Group
Hasso Plattner Institute
Digital Engineering Faculty
University of Potsdam
Potsdam, Germany

Dissertation
zur Erlangung des akademischen Grades eines
Doktor der Naturwissenschaften
—Dr. rer. nat.—

Date of Defense: March 23rd, 2022
November, 2021

Supervisor: Prof. Dr. Mathias Weske, University of Potsdam
Reviewers: Prof. Dr. Diego Calvanese, Free University of Bozen-Bolzano
and
Prof. Dr. Stefanie Rinderle-Ma, Technical University of Munich

Published online on the Publication Server of the University of Potsdam:
https://doi.org/10.25932/publishup-54613
https://nbn-resolving.org/urn:nbn:de:kobv:517-opus4-546137

Abstract

Knowledge-intensive business processes are flexible and data-driven. Therefore, traditional process modeling languages do not meet their requirements: These languages focus on highly structured processes in which data plays a minor role. As a result, process-oriented information systems fail to assist knowledge workers on executing their processes. We propose a novel case management approach that combines flexible activity-centric processes with data models, and we provide a joint semantics using colored Petri nets. The approach is suited to model, verify, and enact knowledge-intensive processes and can aid the development of information systems that support knowledge work.

Knowledge-intensive processes are human-centered, multi-variant, and data-driven. Typical domains include healthcare, insurances, and law. The processes cannot be fully modeled, since the underlying knowledge is too vast and changes too quickly. Thus, models for knowledge-intensive processes are necessarily underspecified. In fact, a case emerges gradually as knowledge workers make informed decisions. Knowledge work impose special requirements on modeling and managing respective processes. They include flexibility during design and execution, ad-hoc adaption to unforeseen situations, and the integration of behavior and data. However, the predominantly used process modeling languages (e.g., BPMN) are unsuited for this task.

Therefore, novel modeling languages have been proposed. Many of them focus on activities' data requirements and declarative constraints rather than imperative control flow. *Fragment-Based Case Management*, for example, combines activity-centric imperative process fragments with declarative data requirements. At runtime, fragments can be combined dynamically, and new ones can be added. Yet, no integrated semantics for flexible activity-centric process models and data models exists.

In this thesis, *Wickr*, a novel case modeling approach extending fragment-based Case Management, is presented. It supports batch processing of data, sharing data among cases, and a full-fledged data model with associations and multiplicity constraints. We develop a translational semantics for *Wickr* targeting (colored) Petri nets. The semantics assert that a case adheres to the constraints in both the process fragments and the data models. Among other things, multiplicity constraints must not be violated. Furthermore, the semantics are extended to multiple cases that operate on shared data. *Wickr* shows that the data structure may reflect process behavior and vice versa. Based on its semantics, prototypes for executing and verifying case models showcase the feasibility of *Wickr*. Its applicability to knowledge-intensive and to data-centric processes is evaluated using well-known requirements from related work.

Zusammenfassung

Traditionelle Prozessmodellierungssprachen sind auf hoch strukturierte Prozesse ausgelegt, in denen Daten nur eine Nebenrolle spielen. Sie eignen sich daher nicht für wissensintensive Prozesse, die flexibel und datengetrieben sind. Deshalb können prozessorientierte Informationssysteme Fachexperten nicht gänzlich unterstützen. Diese Arbeit beinhaltet eine neue Modellierungssprache, die flexible Prozessmodelle mit Datenmodellen kombiniert. Die Semantik dieser Sprache ist mittels gefärbten Petri-Netzen formal definiert. Wissensintensive Prozesse können so modelliert, verifiziert und ausgeführt werden.

Wissensintensive Prozesse sind variantenreich und involvieren Fachexperten, die mit ihren Entscheidungen die Prozessausführung prägen. Typische Anwendungsbereiche sind das Gesundheitswesen, Rechtswesen und Versicherungen. Diese Prozesse können i.d.R. nicht vollständig spezifiziert werden, da das zugrundeliegende Wissen zu umfangreich ist und sich außerdem zu schnell verändert. Die genaue Reihenfolge der Aktivitäten wird erst durch die Fachexperten zur Laufzeit festgelegt. Deshalb erfordern dieser Prozesse Flexibilität sowohl zur Entwurfszeit wie zur Laufzeit, Daten und Verhalten müssen in enger Beziehung betrachtet werden. Außerdem muss es möglich sein, den Prozess anzupassen, falls eine unvorhergesehene Situation eintreten. Etablierte Prozessmodellierungssprachen, wie z.B. BPMN, sind daher ungeeignet.

Deshalb werden neue Sprachen entwickelt, in denen sich generell zwei Tendenzen beobachten lassen: ein Wechseln von imperativer zu deklarativer Modellierung und eine zunehmende Integration von Daten. Im Fragment-Basierten-Case-Management können imperative Prozessfragmente zur Laufzeit flexibel kombiniert werden solange spezifizierten Datenanforderungen erfüllt sind.

In dieser Arbeit wird *Wickr* vorgestellt. Dabei handelt es sich um eine Modellierungssprache, die das Fragment-Basierte-Case-Management erweitert. *Wickr* kombiniert Prozessfragmente mit einem Datenmodell inklusive Assoziationen und zwei Arten an Multiplizitätseinschränkungen: Die erste Art muss immer gelten, wohingegen die Zweite nur am Ende eines Falls gelten muss. Zusätzlich unterstützt *Wickr* Stapelverarbeitung und Datenaustausch zwischen Fällen. Des Weiteren entwickeln wir eine translationale Semantik, die *Wickr* in gefärbte Petri-Netze übersetzt. Die Semantik berücksichtigt sowohl die Vorgaben des Prozessmodells wie auch die des Datenmodells. Die Semantik eignet sich nicht nur für die Beschreibung eines einzelnen Falls, sondern kann auch mehrere untereinander in Beziehung stehende Fälle abdecken. Durch Prototypen wird die Umsetzbarkeit von *Wickr* demonstriert und mittels bekannten Anforderungslisten die Einsatzmöglichkeit für wissensintensive und datengetriebene Prozesse evaluiert.

Publications

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

1. Stephan Haarmann, Marco Montali, and Mathias Weske. "Technical Report: Refining Case Models Using Cardinality Constraints". In: *CoRR* abs/2012.02245 (2020). arXiv: 2012.02245. URL: https://arxiv.org/abs/2012.02245.

2. Stephan Haarmann and Mathias Weske. "Correlating Data Objects in Fragment-Based Case Management". In: *Business Information Systems - 23rd International Conference, BIS 2020, Colorado Springs, CO, USA, June 8-10, 2020, Proceedings*. Ed. by Witold Abramowicz and Gary Klein. Vol. 389. Lecture Notes in Business Information Processing. Springer, 2020, pp. 197–209. ISBN: 978-3-030-53336-6. URL: https://doi.org/10.1007/978-3-030-53337-3_15.

3. Stephan Haarmann and Mathias Weske. "Cross-Case Data Objects in Business Processes: Semantics and Analysis". In: *Business Process Management Forum - BPM Forum 2020, Seville, Spain, September 13-18, 2020, Proceedings*. Ed. by Dirk Fahland, Chiara Ghidini, Jörg Becker, and Marlon Dumas. Vol. 392. Lecture Notes in Business Information Processing. Springer, 2020, pp. 3–17. ISBN: 978-3-030-58637-9. URL: https://doi.org/10.1007/978-3-030-58638-6_1.

4. Stephan Haarmann and Mathias Weske. "Data Object Cardinalities in Flexible Business Processes". In: *Business Process Management Workshops - BPM 2020 International Workshops, Seville, Spain, September 13-18, 2020, Revised Selected Papers*. Ed. by Adela del-Río-Ortega, Henrik Leopold, and Flávia Maria Santoro. Vol. 397. Lecture Notes in Business Information Processing. Springer, 2020, pp. 380–391. ISBN: 978-3-030-66497-8. URL: https://doi.org/10.1007/978-3-030-66498-5_28.

5. Stephan Haarmann. "Fragment-Based Case Management Models: Metamodel, Consistency, & Correctness". In: *Proceedings of the 13th European Workshop on Services and their Composition (ZEUS 2021), Bamberg, Germany, February 25-26, 2021*. Ed. by Johannes Manner, Stephan Haarmann, Stefan Kolb, Nico Herzberg, and Oliver Kopp. Vol. 2839. CEUR Workshop Proceedings. CEUR-WS.org, 2021, pp. 1–8. URL: http://ceur-ws.org/Vol-2839/paper1.pdf.

6. Stephan Haarmann, Adrian Holfter, Luise Pufahl, and Mathias Weske. "Formal Framework for Checking Compliance of Data-Driven Case Management". In: *J. Data Semant.* 10.1 (2021), pp. 143–163. URL: https://doi.org/10.1007/s13740-021-00120-3.

7. Stephan Haarmann, Marco Montali, and Mathias Weske. "Refining Case Models Using Cardinality Constraints". In: *Advanced Information Systems Engineering - 33rd International Conference, CAiSE 2021, Melbourne, VIC, Australia, June 28 - July 2, 2021, Proceedings*. Ed. by Marcello La Rosa, Shazia W. Sadiq, and Ernest Teniente. Vol. 12751. Lecture Notes in Computer Science. Springer, 2021, pp. 296–310. ISBN: 978-3-030-79381-4. URL: https://doi.org/10.1007/978-3-030-79382-1_18.

8. Stephan Haarmann, Anjo Seidel, and Mathias Weske. "Modeling Objectives of Knowledge Workers". In: *Business Process Management Workshops - BPM 2021 International Workshops, Rome, Italy, September 6-10, 2021, Accepted for Publication*. 2021.

In addition to the publications above, the author of this thesis was involved in the following research indirectly contributing to the thesis:

9. Stephan Haarmann, Nikolai Podlesny, Marcin Hewelt, Andreas Meyer, and Mathias Weske. "Production Case Management: A Prototypical Process Engine to Execute Flexible Business Processes". In: *Proceedings of the BPM Demo Session 2015 Co-located with the 13th International Conference on Business Process Management (BPM 2015), Innsbruck, Austria, September 2, 2015*. Ed. by Florian Daniel and Stefan Zugal. Vol. 1418. CEUR Workshop Proceedings. CEUR-WS.org, 2015, pp. 110–114. URL: http://ceur-ws.org/Vol-1418/paper23.pdf.

10. Kimon Batoulis, Stephan Haarmann, and Mathias Weske. "Various Notions of Soundness for Decision-Aware Business Processes". In: *Conceptual Modeling - 36th International Conference, ER 2017, Valencia, Spain, November 6-9, 2017, Proceedings*. Ed. by Heinrich C. Mayr, Giancarlo Guizzardi, Hui Ma, and Oscar Pastor. Vol. 10650. Lecture Notes in Computer Science. Springer, 2017, pp. 403–418. ISBN: 978-3-319-69903-5. URL: https://doi.org/10.1007/978-3-319-69904-2_31.

11. Stephan Haarmann, Kimon Batoulis, and Mathias Weske. "Compliance Checking for Decision-Aware Process Models". In: *Business Process Management Workshops - BPM 2018 International Workshops, Sydney, NSW, Australia, September 9-14, 2018, Revised Papers*. Ed. by Florian Daniel, Quan Z. Sheng, and Hamid Motahari. Vol. 342. Lecture Notes in Business Information Processing. Springer, 2018, pp. 494–506. ISBN: 978-3-030-11640-8. URL: https://doi.org/10.1007/978-3-030-11641-5_39.

12. Adrian Holfter, Stephan Haarmann, Luise Pufahl, and Mathias Weske. "Checking Compliance in Data-Driven Case Management". In: *Business Process Management Workshops - BPM 2019 International Workshops, Vienna, Austria, September 1-6, 2019, Revised Selected Papers*. Ed. by Chiara Di Francescomarino, Remco M. Dijkman, and Uwe Zdun. Vol. 362. Lecture Notes in Business Information Processing. Springer, 2019, pp. 400–411. ISBN: 978-3-030-37452-5. URL: https://doi.org/10.1007/978-3-030-37453-2_33.

Additionally, the author was involved in the following publications on topics not directly related to the PhD thesis:

13. Ekaterina Bazhenova, Stephan Haarmann, Sven Ihde, Andreas Solti, and Mathias Weske. "Discovery of Fuzzy DMN Decision Models from Event Logs". In: *Advanced Information Systems Engineering - 29th International Conference, CAiSE 2017, Essen, Germany, June 12-16, 2017, Proceedings*. Ed. by Eric Dubois and Klaus Pohl. Vol. 10253. Lecture Notes in Computer Science. Springer, 2017, pp. 629–647. ISBN: 978-3-319-59535-1. URL: https://doi.org/10.1007/978-3-319-59536-8_39.

14. Sebastian Serth, Stephan Haarmann, and Lukas Faber. "Serving Live Multimedia for the Linked Open Data Cloud". In: *47. Jahrestagung der Gesellschaft für Informatik, Digitale Kulturen, INFORMATIK 2017, Chemnitz, Germany, September 25-29, 2017*. Ed. by Maximilian Eibl and Martin Gaedke. Vol. P-275. LNI. GI, 2017, pp. 2487–2498. ISBN: 978-3-88579-669-5. URL: https://doi.org/10.18420/in2017_252.

15. Stephan Haarmann, Kimon Batoulis, Adriatik Nikaj, and Mathias Weske. "DMN Decision Execution on the Ethereum Blockchain". In: *Advanced Information Systems Engineering - 30th International Conference, CAiSE 2018, Tallinn, Estonia, June 11-15, 2018, Proceedings*. Ed. by John Krogstie and Hajo A. Reijers. Vol. 10816. Lecture Notes in Computer Science. Springer, 2018, pp. 327–341. ISBN: 978-3-319-91562-3. URL: https://doi.org/10.1007/978-3-319-91563-0_20.

16. Stephan Haarmann. "Estimating the Duration of Blockchain-Based Business Processes Using Simulation". In: *Proceedings of the 11th Central European Workshop on Services and their Composition, Bayreuth, Germany, February 14-15, 2019*. Ed. by Stefan Kolb and Christian Sturm. Vol. 2339. CEUR Workshop Proceedings. CEUR-WS.org, 2019, pp. 24–31. URL: http://ceur-ws.org/Vol-2339/paper5.pdf.

17. Stephan Haarmann, Kimon Batoulis, Adriatik Nikaj, and Mathias Weske. "Executing Collaborative Decisions Confidentially on Blockchains". In: *Business Process Management: Blockchain and Central and Eastern Europe Forum - BPM 2019 Blockchain and CEE Forum, Vienna, Austria, September 1-6, 2019, Proceedings*. Ed. by Claudio Di Ciccio, Renata Gabryelczyk, Luciano García-Bañuelos, Tomislav Hernaus, Rick Hull, Mojca Indihar Stemberger, Andrea Ko, and Mark Staples. Vol. 361. Lecture Notes in Business Information Processing. Springer, 2019, pp. 119–135. ISBN: 978-3-030-30428-7. URL: https://doi.org/10.1007/978-3-030-30429-4_9.

Acknowledgments

Contrary to romantic beliefs, research is rarely conducted by lonely geniuses that hide away in their isolated chambers. Research is a team effort that involves more people than authors listed on the publications. And I have to thank many.

Foremost, I thank my supervisor Mathias Weske for his continuous support, for creating a pleasant work environment, for teaching and mentoring me, for his countless advice, and much more. Without him, this work would not have been possible.

I thank my colleagues at TSPT for making work so enjoyable, for discussing ideas and non-work-related stuff. I thank Luise Pufahl, who

dragged me as a student assistent into TSPT; Marcin Hewelt for his inspiring work on VCM and his supervision in my Bachelor's Project; Kimon Batoulis and Adriatik Nikaj for helping me to gain initial momentum and answering my newbie questions. Furthermore, thank you, Jan Ladleif, Sven Uhde, and Simon Remy, for sharing the journey and supporting me continously. I can only hope to be as valuable as a colleague as these people have been to me. Also, thanks to all the proofreaders: Kerstin Andree, Leon Bein, Jonas Cremerius, Sven Uhde, Jan Ladleif, Tom Lichtenstein, Simon Remy, Anjo Seidel, and Maximilian Völker.

I'd also like to thank Marco Montali from the Free University of Bolzano. Our discussions have been profound, our collaboration fruitful. More researchers than I can list here deserve thanks for the inspiration they provided. I'm grateful for every review I've received, every discussion I've had.

I'd like to thank our students who showed interest in my work. Some contributed directly or indirectly to my research. I especially thank Adrian Holfter, Leon Bein, Anjo Seidel, and Kerstin Andree. Their thoughts, criticism, and work helped to shape this thesis.

Finally, I like to thank all my friends and my family for being there for me.

Axé

Stephan Haarmann
November 2021

Contents

Abstract iii

Zusammenfassung iv

Publications v

Acknowledgments viii

List of Figures xiv

List of Tables xvi

List of Definitions xvii

1 Introduction 1
- 1.1 Business Process Management . . . 1
  - 1.1.1 Business Process Modeling . . . 3
  - 1.1.2 Knowledge-Intensive Processes . . . 3
- 1.2 Research Objective . . . 5
- 1.3 Contribution . . . 6
- 1.4 Structure of the Thesis . . . 8

2 Preliminaries 11
- 2.1 Data Modeling . . . 12
  - 2.1.1 Data Structure . . . 12
  - 2.1.2 Data Behavior . . . 15
- 2.2 Business Process Modeling . . . 16
  - 2.2.1 Traditional Process Models . . . 16
  - 2.2.2 Fragment-Based Case Management . . . 22
- 2.3 Petri Nets & Formal Execution Semantics . . . 30
  - 2.3.1 Petri Nets . . . 30
  - 2.3.2 Colored Petri Nets . . . 33

3 Related Work 39
- 3.1 Data and Traditional Processes . . . 39
- 3.2 BPM for Knowledge-Intensive Processes . . . 40
  - 3.2.1 Characterizing Knowledge-Intensive Processes . . . 41
  - 3.2.2 Modeling Knowledge-Intensive Processes . . . 42
- 3.3 Formal Execution Semantics . . . 48
  - 3.3.1 Formalization of Traditional Processes . . . 48
  - 3.3.2 Formalization of KiPs . . . 49
- 3.4 Overview of the Most Influential Works . . . 51

4 Wickr: Improving fCM **53**
4.1 Domain Model . . . 53
4.2 Object Behavior . . . 57
4.3 Fragments . . . 58
4.4 Goal Specification . . . 66
4.5 Case Model . . . 66
4.5.1 Structural Satisfiability . . . 66
4.5.2 Object Behavior Conformance . . . 67
4.5.3 Contextual Object Creation . . . 67
4.5.4 Contextual Batch Processing . . . 68
4.5.5 Well-Formed Case Model . . . 69
4.6 Cases in Wickr . . . 70
4.7 Summary . . . 70

5 A Petri Net-Based Semantics for Wickr **73**
5.1 An Example Case . . . 73
5.2 The Case State . . . 74
5.2.1 Case Data . . . 74
5.2.2 Fragment Instances . . . 75
5.3 The Case Behavior . . . 75
5.3.1 Case Instantiation . . . 76
5.3.2 Case Execution . . . 76
5.3.3 Case Termination . . . 79
5.3.4 The Complete Case Model . . . 79
5.4 Translation to Classical Petri Nets . . . 81
5.5 Summary . . . 82

6 Associations and Multiplicity Constraints **85**
6.1 Object Identities . . . 85
6.2 Links . . . 87
6.3 Set Data Object Nodes . . . 91
6.4 Global Multiplicity Constraints . . . 93
6.5 Goal Multiplicity Constraints . . . 95
6.6 Extended Translation . . . 97
6.7 Summary . . . 103

7 Sharing Data Among Cases **105**
7.1 Case Identities . . . 105
7.2 Cross-Case Data Objects . . . 106
7.3 Allocating and Publishing Cross-Case Data . . . 108
7.4 Correlating Cross-Case Data to Cases . . . 110
7.4.1 Attribute-Based Correlation . . . 111
7.4.2 Links and Cross-Case Data . . . 113
7.5 The Case and Cross-Case Data . . . 116

8 Technical Evaluation **119**
8.1 Architectural Overview . . . 120
8.2 Modeling . . . 121

8.3 Compilation and Verification . . . 123
8.3.1 Structural Verification . . . 124
8.3.2 Behavioral Verification . . . 125
8.4 Execution . . . 131
8.4.1 Case Execution Engine . . . 131
8.4.2 Goal Modeling and Planning . . . 133
8.5 Runtime Extension . . . 138

9 Conceptual Evaluation **141**
9.1 Wickr for Knowledge-Intensive Processes . . . 141
9.1.1 Data . . . 141
9.1.2 Knowledge Actions . . . 142
9.1.3 Rules and Constraints . . . 143
9.1.4 Goals . . . 143
9.1.5 Processes . . . 144
9.1.6 Knowledge Workers . . . 145
9.1.7 Environment . . . 146
9.1.8 Comparing Wickr to fCM and Others . . . 147
9.2 Wickr for Data-Centric Processes . . . 149
9.2.1 Design . . . 149
9.2.2 Implementation and Execution . . . 150
9.2.3 Diagnosis and Optimization . . . 151
9.2.4 Tool Implementation and Practical Cases . . . 151
9.2.5 Wickr vs. Data-Centric Approaches . . . 152
9.3 Transferring Insights to BPMN & CMMN . . . 154
9.3.1 Domain Models and BPMN . . . 154
9.3.2 Domain Models and CMMN . . . 156

10 Conclusion **159**
10.1 Summary of the Contribution . . . 159
10.2 Limitations and Future Work . . . 160
10.2.1 Conceptual Extensions . . . 160
10.2.2 Application & Tooling . . . 162
10.2.3 Case Studies & Usability . . . 163
10.3 Final Remarks . . . 163

A Wickr is Turing Complete **167**

Bibliography **171**

Declaration **191**

List of Figures

1.1 The business process lifecycle 2
1.2 Overview of our research method 6
1.3 Schematic representation of our contribution 7

2.1 Modeling levels 11
2.2 A data model for an insurance 13
2.3 An object lifecycle 16
2.4 A process model 18
2.5 A data-aware process model 20
2.6 The fCM domain model of the example 23
2.7 The object lifecycles of the example 24
2.8 fCM fragments of the example 26
2.9 Example of activities' I/O behavior in fragment-based Case Management (fCM) 30
2.10 Petri net for the process model in Figure 2.4 32
2.11 Reachability graph for the Petri net in Figure 2.10 33
2.12 Colored Petri net of the process in Figure 2.4 35
2.13 Partial reachability graph of the CPN example 38

3.1 DECLARE model of the insurance example 43

4.1 A not well-formed Wickr domain model 55
4.2 Well-formed *Wickr* domain model 56
4.3 Object behavior of the Wickr example 57
4.4 Fragments with preconditions in fCM and Wickr 59
4.5 Example of initial fragments in fCM and Wickr 60
4.6 Advanced I/O behavior in Wickr 62
4.7 Wickr fragments for the insurance example 65
4.8 Metamodel of Wickr 71

5.1 Places for abstract case states 74
5.2 Places for data objects 74
5.3 The activity lifecycle 75
5.4 Places for control flow 75
5.5 Transition for the start event 76
5.6 Transitions for activity "assess claim" 77
5.7 The transition for activity "receive external review" . . 78
5.8 Transitions for the gateway $\times_1$ 79
5.9 Transition for the termination condition 80
5.10 Petri net for fragments 1 and 2 of the example 81

6.1 Places to count the number of object per class 86
6.2 Colored Petri net for fragment five 87

6.3 Colored Petri net for fragment six 88
6.4 Petri net for fragment three 90
6.5 Petri net for fragment five including links 91
6.6 Petri net for activity "decide on claim" & output set • . 92
6.7 Petri net for activity "decide on claim" & output set ▲ . 93
6.8 Petri net with multiplicity constraints for fragment five 94
6.9 Petri net example with object registry 95
6.10 Petri net for the termination condition including goal multiplicity constraints 96
6.11 Behavior of claims inclduing guard for goal multiplicities 97
6.12 Petri net for activity "request external review" and outputset □ including goal multiplicity constraints 98

7.1 Mapping of cross-case and local data object nodes . . . 106
7.2 Multi-case Petri net of the start event and the subsequent activity 107
7.3 Multi-case Petri net of activity fragment three 109
7.4 Fragment with two activities and its formalization . . . 110
7.5 Extended Wickr domain model 111
7.6 Extended version of Wickr fragment one 112
7.7 Formalization of correlation of cross-case data objects to cases 114
7.8 Fragment accessing linked cross-case data objects . . . 115
7.9 Petri net formalization of fragment three as depicted in Figure 7.8. 117

8.1 Block diagram of the Wickr architecture 121
8.2 Block diagram of the compiler's architecture 123
8.3 Schematic representation of model checking 125
8.4 The insurance domain model without unbounded multiplicity constraints 126
8.5 Block diagram of the execution engine's architecture . . 132
8.6 Screenshot of the execution engine's UI 132
8.7 Screenshot of the form-based goal modeling 137

9.1 BPMN claim handling process 155
9.2 BPMN suprocess for external reviews 155
9.3 Domain model for the Business Process Model and Notation (BPMN) process models in Figures 9.1 and 9.2. . 156
9.4 CMMN model for handling insurance claims 157

A.1 Domain model for simulating a 2-counter machine . . . 167
A.2 Initial fragment of the 2-counter machine 168
A.3 Fragment for an instruction q_i incrementing CounterA. 168
A.4 Fragment for an instruction q_i decrementing CounterA. 168
A.5 Fragments for an instruction q_i testing current value . . 169

List of Tables

2.1 Multiplicity constraints for Figure 2.2 14

3.1 Overview of important related work 52

4.1 Multiplicity constraints for Figure 4.2 56
4.2 Structural Correctness Criteria 72

6.1 Variables for counters and identities 86

8.1 Results of experiments 130

9.1 Requirements for knowledge-intensive processes 148
9.2 Requirements for data-centric processes 153

List of Definitions

1 Data Model 12
2 Finite State Transition System. 15
3 Object Lifecycle. 15
4 Phase 16
5 Business Process Model. 17
6 Data-Aware Process Model, Data Condition 18
7 Case 22
8 fCM Domain Model 23
9 Process Fragment. 24
10 Termination Condition 27
11 fCM Case Model 27
12 fCM Case 28
13 Petri Net 30
14 Reachability Graph 31
15 Multiset 34
16 Colored Petri Net [56] 34
17 Colored Petri Net Concepts [56] 36
18 Colored Petri Net Firing Rule 36
19 Reachability Graph of Colored Petri Nets 37

20 Wickr Domain Model 54
21 Wickr Data Object Node, Data Condition 62
22 Wickr Fragment 63
23 Well-Formed Wickr Fragment 63
24 Wickr Case Model 66
25 Structurally Satisfiable Case Model. 67
26 Object Behavior Conformance 67
27 Contextual Object Creation. 68
28 Contextual Batch Processing. 68
29 Well-Formed Wickr Case Model 69
30 Wickr Case 70

31 Lazy Relaxed Soundness 127

1 Introduction

Modern enterprises must survive in competitive and complex environments. To succeed, an enterprise must operate effectively and efficiently, while constantly coping with change. Thereby, information systems play a key role. Simply put, they ought to provide the right information at the right time. They organize data and orchestrate business processes. And information systems must be ready for change [20].

Consider an online retailer. It employs a customer relationship management system to store customer-related information, to provide customer support, and to conduct marketing campaigns. The system implements respective functionalities but must also adapt to changes in internal and external requirements. If the retailer decides to add an instant messenger as an additional support channel, the system must adapt. Similarly, when the *General Data Protection Regulation* was published in 2018 [109], customer relationship management systems were required to store consents, to delete personal data permanently if requested, to disclose related data to the customer, and much more. In both cases, the right system can be a competitive advantage.

Information systems engineering is the discipline concerned with the specification, design, analysis, implementation, maintenance, and more of information systems [34]. Research in the discipline is concerned with new information systems engineering methods and methodologies. A common methodology actively researched and successfully applied in industry is Business Process Management (BPM).

1.1 Business Process Management

"A business process consists of a set of activities that are performed in coordination in an organizational and technical environment. These activities jointly implement a business goal" [149]. BPM treats processes as first class citizens: Everything relevant happening in an organization or its environment is viewed from the process perspective. Processes are central and put activities, decisions, events, data, and more into relation. Information systems support and implement processes.

The business process lifecycle (Figure 1.1) describes the phases of a business process [149]. During *design/analysis*, organizations specify processes. In *configuration*, engineers implement the process, e.g., by configuring an information system according to the process specification. The configured information system orchestrates processes and supports process participants during *enactment*. Meanwhile, the system may log data, which business analysts can use to *evaluate* the process and to gain insights before the lifecycle is reiterated.

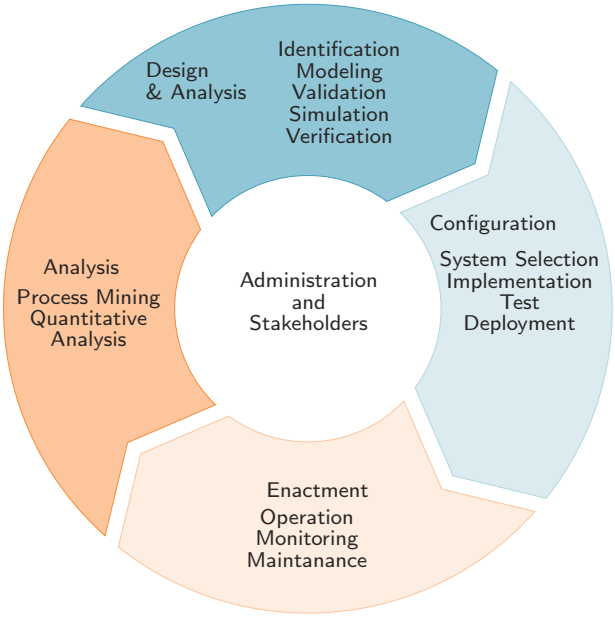


Figure 1.1: The business process lifecycle (cf. [149, p. 12]).

During design, stakeholders identify and specify a process. They determine its potential triggers and its goal. Furthermore, all activities that might be necessary to achieve the goal as well as dependencies among activities are specified. The outcome of the design phase is usually a process model. The model can be analyzed, validated, simulated, and verified as a surrogate for the real process.

The online retailer in our example plans to design its shipment process. Therefore, workshops with stakeholders are conducted, and the process is described: Every new order triggers the process which completes when the order has been shipped and the invoice has been paid. To handle the order, the retailer sends an invoice to the customer, and the warehouse retrieves, packs, and sends the ordered products. The process also waits for the customer to pay the invoice. To validate the process, the stakeholders may use simulation. They may recognize that an order cannot be canceled and extend the model accordingly.

A process model is a blueprint for many similar process instances. However, after the design phase, the process specification lacks details. Therefore, engineers implement the process during configuration, e.g., using information systems. Missing details are added; the implementation is tested; and eventually the process is deployed.

In the example, the retailer's process must handle different payment channels. Furthermore, it needs to access customer data. During implementation, the process is configured to support multiple payment providers and to request and store user data.

The deployed process can be enacted. When a trigger occurs, a new instance is started. By executing activities in a defined order, process participants and services move the process instance towards its goal. Once a goal has been accomplished, the process terminates.

During execution, data can be logged. Afterwards, business analysts can use process mining techniques to gain insights into the process. Do instances deviate from the model? Are there bottlenecks or other shortcomings? Answers to these and similar questions provide input for the next iteration of the lifecycle: The process can be redesigned and improved.

After receiving and handling many orders, the retailer in the example decides to evaluate the status quo. An analysis of the process execution logs shows that some orders do not complete because customers do not pay. In the next design iteration, the retailer decides to add an activity for sending a reminder, and another activity to cancel orders after three missed payments.

1.1.1 Business Process Modeling

Process models are central to BPM [128, 149]. At design-time, process models are created and analyzed. During implementation, processes act as a specification. At run-time, engines can interpret models. And after execution, models can be discovered from data and results can be visualized with models.

Many modeling languages for process exist; some are better suited for certain tasks than others. However, clear semantics are often required [15, 46]. If the semantics of a model is not clear, it is difficult to reach a common understanding; ambiguities must be resolved during implementation; and when interpreted, informal models support different interpretation which lead to differences during execution. Therefore, standardized modeling languages with formal semantics are employed.

The de facto industry standard is Business Process Model and Notation (BPMN) [92]. A BPMN process model consists of events, activities, and gateways that are connected by control flow. The control flow, describes causal dependencies between the elements. Furthermore, data objects and data flow can model inputs and outputs of activities and events. A BPMN model is imperative since every possible sequence of activities must be modeled explicitly. Therefore, BPMN is well suited to describe and standardize highly structured processes. This can be an advantage because structure reduces implementation effort, improves consistency, and comprehension [51, 69]. For these and other reasons, BPMN is widely used by industry, e.g., in e-commerce, legal tech, or supply chain management. Yet, limiting a process to a few standardized variants is not always desirable [51, 72].

1.1.2 Knowledge-Intensive Processes

Knowledge work is unpredictable and unrepeatable [82]. Its domains are often inherently complex. Knowledge workers use their expertise and experience to choose the right actions out of many alternatives. And while knowledge work is increasingly important to businesses [53], knowledge-intensive processes are not well-supported by traditional process modeling languages, such as BPMN [72, 82, 98].

Physicians are exemplary knowledge workers. They are trained extensively and often specialize before joining the workforce. Furthermore, they constantly need to keep up with advances in medicine. When they treat a patient, physicians apply their knowledge to decide on a highly personalized treatment. Therefore, they require detailed information about the patient, possible treatments, clinical guidelines, and regulations. At the same time, this knowledge changes: New information about the patient, such as an allergy, may be discovered; research and industry develops new treatments; clinical guidelines are updated; and regulations change. It is hard, if not impossible, to model all this explicitly and to keep the model up-to-date.

Highly structured process models that describe particularly flexible behavior are often perplexing [39, 51]. Therefore, the BPM community proposes novel methods, technologies, and modeling languages for knowledge-intensive processes.

- Knowledge workers constantly make decision that impact how the process' advances. Therefore, respective process models define what actions are possible instead of how activities should be ordered. Modeling languages become more declarative.
- Knowledge workers gather and consider information about a particular case. Therefore, data should be tightly integrated into processes and models.
- Knowledge-intensive processes are often long-running and fast-changing at the same time. Many approaches allow unspecified behavior or adapting models ad hoc.

Case management is a BPM paradigm tailored towards knowledge work [53, 82]. It is based on the idea that every process evolves around a case, i.e., a patient, an insurance claim, or a lawsuit. Changing the state of the case to a desired outcome is the primary concern of the process, and the case's current state essentially defines possible activities. Knowledge-intensive processes require ad hoc adaptation: If knowledge workers encounter unforeseen situations, they can deviate from the modeled behavior.

A patient comes with health problems. The goal is a successful treatment. Physicians perform multiple test to diagnose the cause, and every new result narrows down the treatment. However, if the patient has an allergy against a prescribed drug, the treatment may deviate from the projected path.

Case management can support this scenario: All relevant information is organized in the case. Furthermore, many process variants are supported, and the physicians choose the one that fits the specific patient best. If the existing variants are not enough, physicians can adapt the model ad hoc to their and the patient's needs.

Different modeling languages support case management. Among them are data-centric approaches, for example, Guard-Stage-Milestone (GSM) [60] and PHILharmonicFlows [66]. The enablement of activities,

as well as their effects, are defined by data conditions and data operations, respectively. Furthermore, declarative approaches exist such as Dynamic Condition Response Graphs (DCR Graphs) [59] and DECLARE [43]. These approaches define declarative constraints between activities, which must not be violated.

Our work is based on fragment-based Case Management (fCM) [112], which is a modeling language tailored towards case management. It combines ideas from activity-centric and data-centric process modeling. At design-time, the process is split into fragments. At runtime, knowledge workers can combine fragments almost freely. The only constraints are imposed by fragments' and activities' data requirements. Furthermore, the model can be adapted at runtime to account for unforeseen situations.

1.2 Research Objective

The central role of data is one factor that distinguishes knowledge-intensive from traditional processes [98]. For example, the treatment of a patient depends highly on the diagnosis, while the sequence of steps required to ship an order of books can be laid out without knowing any specificities of the order. A knowledge-intensive process creates and maintains data, and data in turn controls the process behavior. In case management, models are meant to describe these relationships.

Conceptually, the structure of data is often modeled by data models that include entities and relationships [4, 123], and the behavior of processes is modeled by process models that include activities and dependencies among them [24, 92]. Yet, in case management, where flexibility plays an important role, few approaches integrate a data-centric and an activity-centric view, and even fewer provide a joint formal semantics [98, 112, 147]. Some approaches focus on data but do not have an activity-centric view [60, 86]. Others focus on activities but lack the data-centric view [43, 59]. In their survey [89], Hauder et al. list *data integration* as one of five key challenges in case management. Another is *theoretical foundation*. These challenges lead directly to our research questions:

RQ1 How does the flexible behavior of a case model create and maintain data?

RQ2 How do the constraints of a data model affect the case execution?

To understand the relationship between process and data, we investigate how data-driven knowledge-intensive processes are modeled. Thereby, we especially focus on the relationship between data models and process models. To answer the research question, we develop a new case management approach, and we develop a formal execution semantics for respective case models. In this semantics, processes instantiate a data model and adhere to its constraints.

Our research was conducted iteratively (see Figure 1.2). We started with the existing case management approach fCM [112] and developed

a formal execution semantics. We evaluated the current state technically using prototypes and conceptually using lists of requirements and by comparing our approach against related work. Afterwards, we extended our approach with additional data modeling constructs. By refining the formal semantics, we entered the next iteration. Eventually, we reached the current state summarized in the novel case management approach *Wickr*.

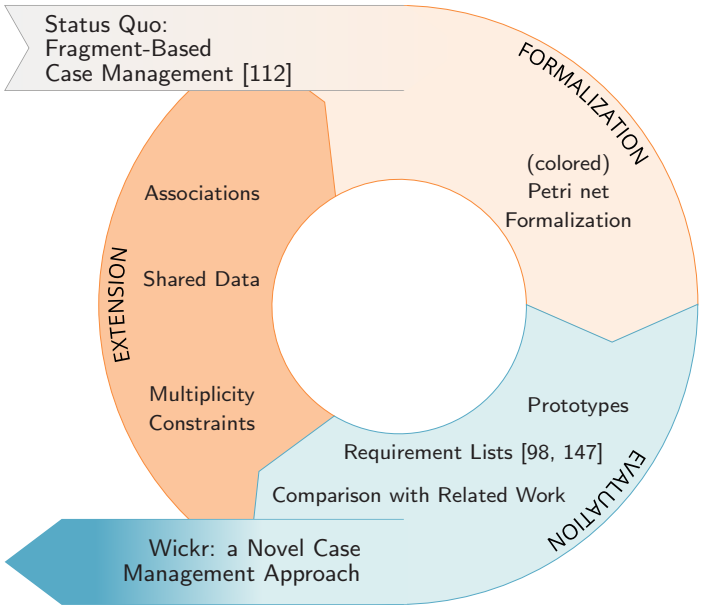


Figure 1.2: Overview of our research method (cf. [38, 171]). We incrementally i) defined formal semantics, ii) evaluated our approach, before iii) extending it.

In this thesis, we focus on the combined results of our research efforts rather than presenting each iteration.

1.3 Contribution

In this thesis, we present a joint semantics for case models and data models. The contribution is split into parts:

- We introduce *Wickr*—a novel case management approach, which extends fCM with full-fledged data models. *Wickr* supports data objects, links, multiplicity constraints, and shared data.
- We define *Wickr's* semantics by translating case models to Petri nets and colored Petri nets.
- We develop and present tools for using *Wickr* for different BPM tasks, such as modeling, analysis, execution, and planning.
- We evaluate *Wickr* conceptually using two respective frameworks introduced by literature. Using the results, we compare *Wickr* against other process modeling approaches.

- We transfer important insights to the two standard modeling languages BPMN and Case Management Model and Notation (CMMN).

Knowledge work is data-driven [98]. Data requirements and data acquisition influence how a case can evolve. Furthermore, in computer science, it is often assumed that data is structured. Naturally, if this structure is constrained and if the process is responsible for creating data, the process is constrained as well. Yet, activity-centric process modeling often ignores the impact data constraints have on the process execution.

In *Wickr*, we consider the relationship between data and processes (cf. Figure 1.3). A case model includes both a process model (i.e., fragments) and a data model (i.e., a class diagram). At runtime, a case has multiple activity instances that execute the modeled behavior. Activities may operate on data. They can create, read, and update data objects. However, the data model constrains data objects, and activities must adhere to these constraints.

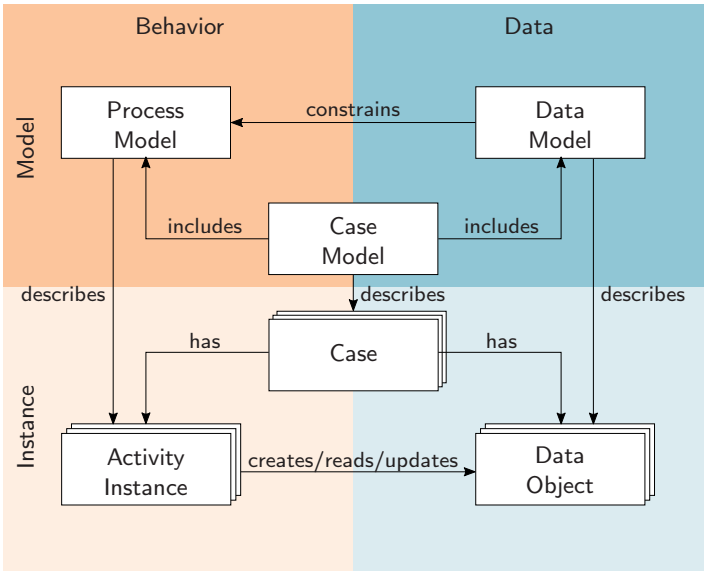


Figure 1.3: Schematic representation of our contribution. Case behavior and data are in a mutual relationship. The data model constrains the process model. Consequently, activities that create, read, and update data must adhere to data constraints.

We specify *Wickr's* semantics formally through a translation to (colored) Petri nets. Petri nets are formal behavioral models. They have clearly defined semantics. When translating case models to Petri nets, we must specify when classes get instantiated and when objects get linked. This enables us to capture the data state precisely and to check data constraints, e.g., on multiplicities. The mapping integrates both the data model and the process model into one formal representation. Based on the integrated semantics, we discuss structural and behavioral correctness criteria. In general, combining data and process modeling

allows capturing the case more precisely. However, it also introduces additional complexity.

We show *Wickr's* feasibility through prototypes. A compiler translates case models to CPNTools-compatible Petri nets. Using CPNTools [56], the nets can be analyzed and verified. An execution engine takes a case model and supports enactment: It tracks the state of the case and generates forms for user input. Furthermore, we present a framework for modeling goals at runtime and planning actions accordingly.

We also evaluate *Wickr* conceptually. Di Ciccio et al. describe characteristics and requirements for knowledge-intensive processes [98]. We use the requirements to evaluate *Wickr* and to compare it against other approaches. We also compare *Wickr* to data-driven approaches [147] and see that it provides comparable support, while not being fully data-centric. Finally, we apply results to the modeling languages BPMN and CMMN to show that of our approach can be generalized.

With *Wickr*, we present a novel case management approach that integrates data into flexible, activity-centric processes. Thereby, the processes are defined more precisely, which improves the explanatory power and helps to design correct systems.

1.4 Structure of the Thesis

In this thesis, we will provide background information, present related work, introduce *Wickr's* syntax and semantics, and evaluate our contribution. The rest of this thesis is structured as follows:

Chapter 2 (Preliminaries). First, we provide a brief introduction into process and data modeling. This includes class diagrams to model data structures, state transition systems to model object behaviors, highly structured process models with and without data, and fCM. We also introduce formal behavioral models, i.e., classical and colored Petri nets. These are the foundations for our contribution.

Chapter 3 (Related Work). We consider related work from different areas. First, we discuss works improving the data-awareness of traditional business process models. Next, we present knowledge-intensive processes and respective modeling approaches. The fCM approach gets dedicated attention because it is the groundwork for *Wickr*. Finally, we present works on formal execution semantics for both traditional, knowledge-intensive, and data-centric processes.

Chapter 4 (Wickr: Improving fCM). We improve fCM by elaborating data support and removing ambiguity. The new approach is named *Wickr*. It supports data models with associations and two types of multiplicity constraints. One of them holds globally; the other one holds eventually. *Wickr* supports batch processing and cross-case data objects. Furthermore, we define structural consistency criteria that describe when the parts of a case model are correctly integrated.

Chapter 5 (A Petri Net-Based Semantics for Wickr). We present a Petri-net-based semantics for *Wickr* that precisely describes the control flow within fragments. It also supports data flow. However, data identities, links, and multiplicity constraints are not covered. Consequently, batch processing (which relies on links) and multiple concurrent cases (which rely on identities) are also not supported.

Chapter 6 (Associations and Multiplicity Constraints). Object identities allow us to distinguish between different data objects and to model links. We extend the Petri net semantics of *Wickr* to create unique identities, whenever a new data object is created. Therefore, we use colored Petri nets, in which tokens can contain data objects. We also formalize links among data objects, multiplicity constraints, and their impact on activities.

Chapter 7 (Sharing Data Among Cases). We extend the mapping further. Besides object identities, we introduce identities for cases. Consequently, multiple concurrently running cases can be modeled. The cases do not interfere, except for cross-case data objects. Cross-case data objects are shared among multiple cases and allow synchronized data access. We present correlation mechanisms to connect cross-case data objects to cases. In this regard, we discuss the effect of links and multiplicity constraints.

Chapter 8 (Technical Evaluation.) We provide multiple prototypes for *Wickr*. Case models can be designed in standard process and data modeling tools. A dedicated compiler translates case models to colored Petri nets. A colored Petri net can be used for verification against generic and domain-specific rules. However, properties known from traditional business processes are not necessarily meaningful for knowledge-intensive ones. We discuss various properties, such as variants of soundness.

We also present tools assisting knowledge workers at runtime: An engine tracks the case state and generates forms to enter and view data. A planning component allows modeling goals and planning a case accordingly. We also discuss the challenges and limitations of modeling, verifying, and executing flexible processes.

Chapter 9 (Conceptual Evaluation). We complement *Wickr's* technical evaluation with a conceptual one. Using well-known frameworks for knowledge-intensive and data-centric process modeling approaches, we assess *Wickr's* capabilities to represent respective processes. We compare the results to those of other modeling languages. Not only the language but also other aspects, such as tooling and adaptability, are covered. The evaluation shows both strengths and weaknesses of *Wickr*. Furthermore, we take insights gained during our research on *Wickr* and apply them to BPMN and CMMN.

Chapter 10 (Conclusion). Finally, we conclude the thesis. We summarize our contribution. We also look at open questions and potential improvements: In the future, *Wickr's* support for data modeling constructs may be improved further, tooling may be elaborated, and additional use cases may be discussed. Yet, *Wickr* already shows the benefits and challenges of coupling data and flexible process models.

2 Preliminaries

Modeling is an important and common in information systems engineering [42, 94, 149] because models help to tame complexity by focusing on important aspects, while leaving irrelevant parts out [3]. Engineers use models to communicate with stakeholders and to foster a common understanding. Also, models can be verified, analyzed, and validated in place of the modeled original, i.e., to find errors [67].

Models can have multiple instances, and a model whose instances are models themselves is a metamodel (see Figure 2.1) [116]. Metamodels define elements (i.e., concepts and relationships) used in respective models. By assigning a visual representation for each element in the metamodel, a notation for models is defined. The semantics of a model describes the set of all possible instances. Some models have formal semantics—we can determine the set of instances unambiguously. Others have informal or semiformal where this is not possible.

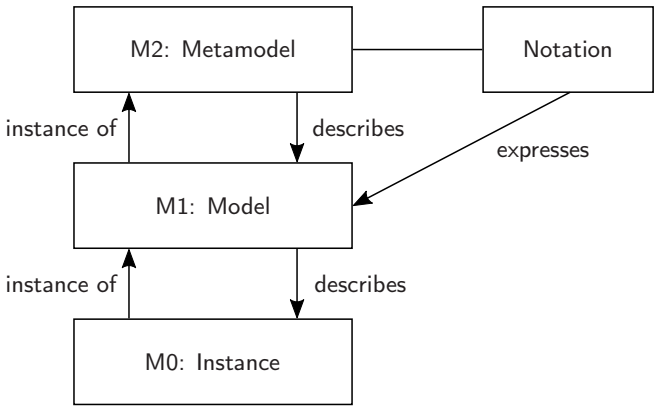


Figure 2.1: Modeling layers according to the meta object facility [116].

In information systems engineering, many models fall in one of three categories [42]: Structural, functional, and behavioral models. Structural models describe how a system is structured; functional models describe the provided functionality; and behavioral models define how systems behave. A single system is often modeled by multiple complementary models: When designing an information system, we may use a data model that describes the structure of information, a functional model that contains the provided functionality, and a set of process models that define the system's behavior.

In this thesis, we develop a novel case management approach. It combines aspects from structural and behavioral models and provides a joint semantics. In this chapter, we briefly introduce structural and behavioral data modeling (Section 2.1), process modeling and case management (Section 2.2), as well as Petri nets (Section 2.3).

2.1 Data Modeling

Information systems provide access to factual information called data.[1] In computer science, structured, semi-structured, and unstructured data are distinguished. We are interested in structured data, which can be described by structural data models. However, even structured data is not necessarily static: Data items may be created, updated, and deleted. Therefore, we also use behavioral models to describe how individual data artifacts may evolve. Furthermore, we are primarily concerned with conceptual modeling and not with implementation.

2.1.1 Data Structure

Following common approaches to conceptual modeling [94, 100, 123] and especially process modeling [66, 92], we consider data to be contained in a set of *objects*, which may be linked to one another. Structural data models describe the internals of objects as well as connections among them. In this thesis, we use the terminology of the Unified Modeling Language (UML) [123]. A *class* is a model for similar data objects and consists of a name and typed attributes. *Associations* connect classes and model possible *links* between data objects. Yet, there can be many objects for one class and many links for one association. *Multiplicity constraints* are defined for each association and each associated class. They define how many respective links a data object may have at least and at most. Thus, a data model describes possible data structures.

An instance of a structural data model consists of a set of data objects and links, so that each data object is an instance of a class, and each link is an instance of an association, and the entirety adheres to the multiplicity constraints. While a data model is fixed, the instance may change. Over time, objects can be updated, new objects can be created, and additional links can be established. However, the instance must always comply to the model.

We consider classes and associations, while abstracting data attributes (see Definition 1). Each association has multiple ends that connect to classes. Each end has a role and a multiplicity constraint.

Definition 1 (Data Model)**.** A data model d is a tuple

$$d = (C, R, A, \ell, u), \text{ where}$$

1. C is a finite non-empty set of classes.
2. R is a finite set of roles.
3. A is a finite set of associations.
4. $\ell : (C \times R \times A) \to \mathbb{N}_0$ assigns each class for a given role and association a lower bound.

[1] `https://www.merriam-webster.com/dictionary/data` (2021/11/06)

5. $u : (C \times R \times A) \to (\mathbb{N}_0 \cup \{*\})$ assigns each class for a given role and association an upper bound. It holds that $*$ is greater than any number: $\forall n \in \mathbb{N}_0 : n < *$. A class $c \in C$ takes part in an association $a \in A$ as role $r \in R$ iff $u(c, r, a) > 0$.

Furthermore, the following properties must hold:

1. Each association has at least two ends:

$$\forall a \in A : |\{(c, r) \in (C \times R) | u(c, r, a) > 0\}| \geq 2$$

2. Upper bounds must be greater than or equal to the respective lower bound:

$$\forall c \in C, a \in A, r \in R : l(c, r, a) \leq u(c, r, a)$$

◇

Two functions l and u define multiplicity constraints' upper and lower bounds, respectively. for an association a that connects to a class c with role r, the value $u(c, r, a)$ defines the multiplicity constraint's upper bound. If $u(c, r, a) = 0$, no such association end exists.

We visualize data models as UML class diagrams [123]. Classes are represented by rectangles, binary associations by undirected edges, and n-ary association by a set of edges and a rhombus. Roles and multiplicity constraints are annotated textually. Furthermore, we omit roles that equal the corresponding class name in lower case.

Figure 2.2 depicts a data model for an insurance. It describes policies, clients, claims, and reviews, as well as their relationships. Clients can be linked as insurers (a_1) or co-insurers (a_2) to policies. Policies have exactly one insurer and arbitrarily many co-insurers. Every client can be linked to a set of claims. Claims can be linked to arbitrarily many reviews, and for each review, there is exactly one claim.

Formally, the example is described by $d_i = (C_i, R_i, A_i, l_i, u_i)$, where

$C_i = \{\textit{Client}, \textit{Policy}, \textit{Claim}, \textit{Review}\}$
$R_i = \{\textit{insurer}, \textit{co-insurer}, \textit{policy}, \textit{claim}, \textit{review}, \textit{client}\}$
$A_i = \{a_1, a_2, a_3, a_4\}$

The multiplicity constraints are defined by Table 2.1

Objects for clients, policies, claims, and reviews with corresponding links compose the instances of the data model d_i. Any valid instance

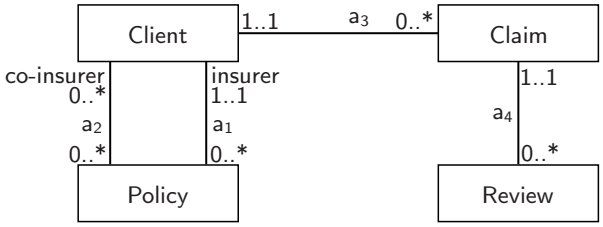


Figure 2.2: A data model for an insurance.

Table 2.1: Formal definition of the multiplicity constraints in Figure 2.2.

Class	Role	Association	ℓ_i	u_i
Client	insurer	a_1	1	1
Client	co-insurer	a_2	0	*
Client	client	a_3	1	1
Policy	policy	a_1	0	*
Policy	policy	a_2	0	*
Claim	claim	a_3	0	*
Claim	claim	a_4	1	1
Review	review	a_4	0	*
all other combinations			0	0

must adhere to the multiplicity constraints. Let O_i be a set of objects partitioned into clients, policies, claims, and reviews.

$$O_i = O_{Client} \cup O_{Policy} \cup O_{Claim} \cup O_{Review}$$

Furthermore, let L_i be a set of links partitioned according to the associations, where a link is a set of pairs, which consist of a role and a corresponding object.

$$L_i = L_1 \cup L_2 \cup L_3 \cup L_4$$

The tuple (O_i, L_i) is a valid instance of d_i if it satisfies the multiplicity constraints. For association, a_1 the following must hold:

$$\begin{aligned} \forall o_p \in O_{Policy} : & \ell_i(Client, insurer, a_1) \leq \\ & |\{o_c \in O_{Client} | \{(policy, o_p), (insurer, o_c)\} \in L_1\}| \leq \\ & u_i(Client, insurer, a_1) \end{aligned}$$

The following instance satisfies all the multiplicity constraints and is therefore a valid instance of the model:

$$\begin{aligned} O_{Client} =& \{Miller, Smiths, Adams\} \\ O_{Policy} =& \{Policy_1, Policy_2\} \\ O_{Claim} =& \{Claim_1\} \\ O_{Review} =& \emptyset \\ L_1 =& \{\{(insurer, Miller), (policy, Policy_1)\}, \\ & \{(insurer, Smith), (policy, Policy_2)\}\} \\ L_2 =& \{\{(co\text{-}insurer, Adams), (policy, Policy_2)\}\} \\ L_3 =& \{\{(client, Smiths), (claim, Claim_1)\}\} \\ L_4 =& \emptyset \end{aligned}$$

The instance has to comply with the data model at all times: When, for example, a review is created, it must be linked to a claim. When a new claim is made, it must be linked to one of the clients.

Definition 1 has its limitations. In practice, some data models distinguish between different types of associations, most commonly whole-part relationships. Furthermore, inheritance can be used to construct class hierarchies (subtypes). However, we do not consider specialized associations and hierarchies in this thesis.

2.1.2 Data Behavior

Data objects have states, and states can change over time. The state of an object comprises all internal information and the links to other objects. In behavioral models, however, states are often reduced to a finite, non-empty set of abstract values. Each of these values is meaningful to the domain: For example, the state of insurance claims may be reduced to a set of important milestones.

Once relevant states have been identified, the object behavior can be described by finite state transition systems. A state transition system (Definition 2) consists of a set of states and a set of valid state transitions. It can be used to describe the state space of objects. Given an object in a state, it can only change to one of the defined successor states.

Definition 2 (Finite State Transition System.)**.** A finite state transition system b is a tuple $b = (Q, \delta)$, where

1. Q is a finite non-empty set of states;
2. $\delta \subseteq Q \times Q$ is the set of possible state transitions.

◇

Often, the object behavior is enriched with additional information. A common variant is the object lifecycle (Definition 3). It is a state transition system with a dedicated initial state and a set of final states. When a class is instantiated, the new object is in its initial state. Eventually, the object ought to reach a final state.

Definition 3 (Object Lifecycle.)**.** An object lifecycle λ is a tuple

$$\lambda = (Q, q_0, Q_f, \delta), \text{ where}$$

1. Q is a finite non-empty set of states.
2. $\delta \subset Q \times Q$ is the set of possible transition relation.
3. $q_0 \in Q$ is the initial state.
4. $Q_f \subseteq Q$ is the set of final states.

◇

Figure 2.3 depicts the object lifecycle for class *claim*. A claim starts in state *received*. It can change to state *assessed*. In *assessed*, the object can

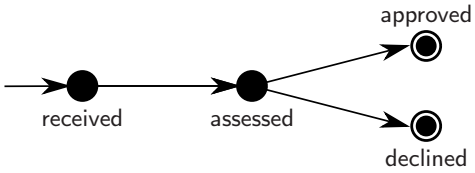


Figure 2.3: Object lifecycle for claim objects. The initial state is *received*. The final states are *declined* and *approved*.

transition to *declined* or *approved*. Formally, the object lifecycle λ_{Claim} is specified as follows: $\lambda_{Claim} = (Q_{Claim}, q_{0;Claim}, Q_{f;Claim}, \delta_{Claim})$, where

$$
\begin{aligned}
Q_{Claim} &= \{received, assessed, approved, declined\} \\
q_{0;Claim} &= \{received\} \\
Q_{f;Claim} &= \{approved, declined\} \\
\delta_{Claim} &= \{(received, assessed), (assessed, approved), (assessed, declined)\}
\end{aligned}
$$

We call a combination of a class and a state a phase (Definition 4) [100].

Definition 4 (Phase)**.** Let c be a class and $b_c = (Q_c, \delta_c)$ the state transition system for objects of c. $PH_c = \{c\} \times Q_c$ is the set of phases for class c. For a phase (c, q), we also write $c[q]$. ◇

2.2 Business Process Modeling

Business processes group activities that, when executed in coordination, contribute to a common goal [149]. An activity is a unit of work. When performed, activities may create, read, and update data objects. A process model defines activities and their possible execution orders.

Traditional process models are well-defined and highly structured. Well-defined means that all internal and external constraints are specified. Highly structured requires that all variants are defined explicitly. These process models are suited for standardization of repeatable processes [51]. Yet, they do not fit knowledge-intensive processes, which are flexible and often not fully known at design-time [72]. Therefore, case management has been proposed.

This section is dedicated to process modeling. We present traditional business process models with and without data, as well as fragment-based Case Management (fCM), which is the outset of our work.

2.2.1 Traditional Process Models

Traditional process models are imperative and activity-centric. They limit a process to a few fully defined variants, which reduces the effort spend on implementation and quality assurance. Furthermore, when executed frequently, small improvements may have significant effects. The most common imperative and activity-centric process modeling language is BPMN [92, 107].

Activity-centric process models focus on activities and causal dependencies among them. On this basis, the state of a process instance depends on the activities that have been executed already, and the state determines the activities that can be executed next. The process model represents this causal dependency by control flow. When an activity is executed, its outgoing control flow is triggered, which enables the next activity. Additionally, gateways represent decisions and concurrency: An exclusive gateway triggers only one of multiple alternative control flows; a parallel gateway triggers all outgoing control flows.

Definition 5 (Business Process Model.)**.** A business process model p is a tuple $p = (N, \xrightarrow{N})$, where

1. $N = \{s\} \uplus N_A \uplus N_\times \uplus N_+ \uplus N_E$ is a set of control flow nodes partitioned into
   a) a single start event s,
   b) a finite non-empty set of activities N_A,
   c) a finite set of exclusive gateways $N_\times$,
   d) a finite set of parallel gateways N_+, and
   e) a finite non-empty set of end events N_E.
2. $\xrightarrow{N} \subseteq (N \setminus N_E) \times (N \setminus \{s\})$ is the control flow relation.

Furthermore, the control flow must satisfy the following properties: Each node $n \in N$ is on a path from s to an end event $e \in N_E$. The start event has one outgoing and no incoming control flow. Each activity has exactly one incoming and one outgoing control flow, and each end event has exactly one incoming and no outgoing control flow. Gateways have at least one incoming and one outgoing control flow. Each gateway should have multiple incoming or multiple outgoing control flows. $\diamond$

Figure 2.4 depicts a process model for handling an insurance claim. The process begins when the insurance receives a new claim from a client (s_1). A worker assesses the claim (a_1) and decides $(\times_1)$ whether a review is necessary. If the worker decides for a review, they subsequently request (a_2) and receive (a_3) it. Eventually, the worker makes a decision (a_4): If the claim is declined, the worker informs the client (a_5) and the process terminates (e_1). In case of approval, the worker concurrently $(+_1)$ informs the client (a_6) and pays the reimbursement (a_7).

While data is not modeled, it is still present in the process: The claim, the assessment, the review, and the decision are data objects.

Data plays a minor role in traditional business process modeling. The BPMN standard version 1.0 states

> [...] the behavior of the Process can be modeled without Data Objects for modelers who want to reduce clutter. The same Process can be modeled with Data Objects for modelers who want to include more information without changing the basic behavior of the Process. [41, p. 92]

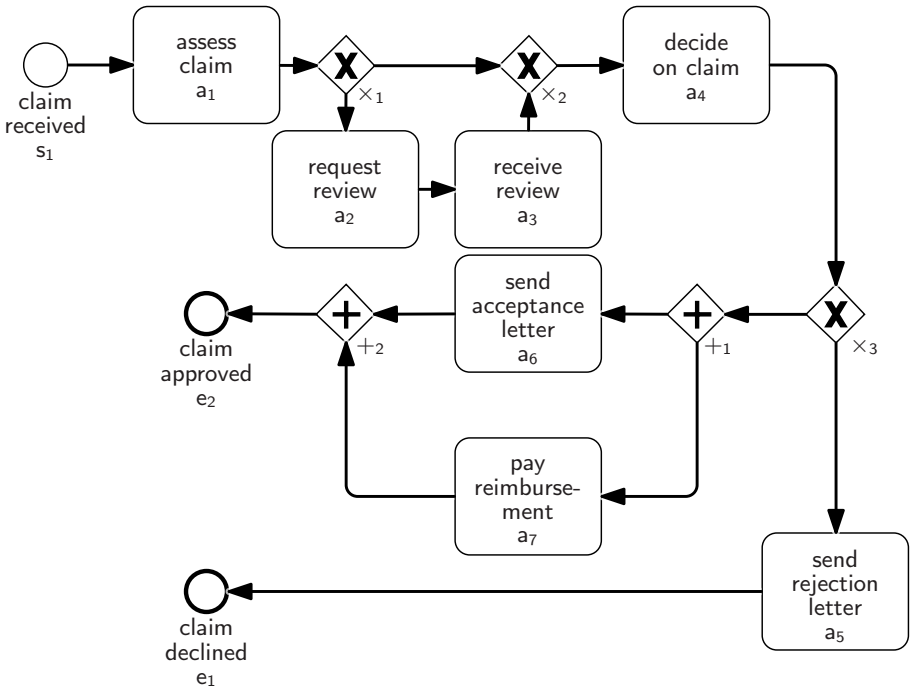


Figure 2.4: Process model for the claim handling process at an insurance.

Data can be modeled by data objects that consist of a name (often the class name) and a state. Activities create, read, and write data objects (in specific states). Yet, data only augments the process without changing the process execution.

Since BPMN version 2.0 [92], additional data-related information can be specified, but they are not represented visually:

- Data objects may reference a data structure [92, p. 90].
- The inputs and outputs of activities are grouped into alternative input sets and output sets, respectively [92, p. 217 f.]. An additional relation defines possible input-output-set combinations.

BPMN 2 also acknowledges the significance of data on process execution: To execute an activity, at least one of its inputs sets is required. Furthermore, data conditions may define how the process branches. We define data-aware processes accordingly (Definition 6).

Definition 6 (Data-Aware Process Model, Data Condition)**.** A data-aware process model p_{da} is a tuple $p_{da} = (N, \xrightarrow{N}, DO, i, o, io, con)$, where

1. $N = \{s\} \cup N_A \cup N_\times \cup N_+ \cup N_E$ is a set of control flow nodes partitioned into
   a) a single start event s,
   b) a finite non-empty set of activities N_A,
   c) a finite set of exclusive gateways $N_\times$,

d) a finite set of parallel gateways N_+, and

e) a finite non-empty set of end events N_E.

2. $\xrightarrow{N} \subseteq (N \setminus N_E) \cup (N \setminus \{s\})$ is the control flow relation.

3. DO is a set of data object nodes. Each data object node $(c, q) \in DO$ is a phase (Definition 4).

4. $i : N_A \to \mathcal{P}(\mathcal{P}(DO))$ assigns each activity a non-empty set of (possibly empty) input sets.

5. $o : (N_A \cup \{s\}) \to \mathcal{P}(\mathcal{P}(DO))$ assigns each activity and the start event a non-empty set of (possibly empty) output sets.

6. $io : N_A \to \mathcal{P}(\mathcal{P}(DO) \times \mathcal{P}(DO))$ is a function assigning each activity a set of valid input-output-set combinations. It holds that $io(a) \subseteq i(a) \times o(a)$ for all activities $a \in N_A$.

7. $con : \xrightarrow{N} \to \mathcal{P}(\mathcal{P}(DO))$ assigns each control flow a data condition (see below). Only control flow starting in an exclusive gateway can be conditional:

$$\forall n_1 \xrightarrow{N} n_2 : n_1 \notin N_\times \Rightarrow con(n_1, n_2) = \{\emptyset\}$$

A *data condition* is a set of sets that contain data object nodes. A data object node is used as a propositional variable. Its value is *true* if a respective object exists. Otherwise, it is *false*. This means the empty set is equivalent to *true*. Let $\mathcal{CON}$ be a data condition, it is interpreted as a logical formula in disjunctive normal form, where a set of data object nodes is a *product term*:

$$\mathcal{CON} \equiv \bigvee_{\Theta \in \mathcal{CON}} \left(\bigwedge_{(c,q) \in \Theta} (c, q) \right)$$

$\diamond$

Data object nodes can connect a process model to a data model, yet the semantics of such models do not consider details of the data structure, such as associations and multiplicity constraints.

Figure 2.5 models a data-aware version of the insurance process. When the start event occurs, a *claim* object is created in state *received*. It is input to activity a_1, which has two output sets: The claim changes to state *assessed* and a risk object is created either in state *regular* or in state *high*. In case a review is required, a_2 is executed, and a respective object is created in state *required*. Activity a_3 updates the review to state *received*. Activity a_4 models a decision. It has three input sets, consisting of a claim in state *assessed* and either

- a risk in state *regular*; or
- a risk in state *regular* and a review in state *received*; or

- a risk in state *high* and a review in state *received*.

Activity a_4 also has two alternative output sets, it either *approves* or *declines* the claim. The state of the claim is used for the conditional branching of the following exclusive gateway ($\times_3$): In the model, we represent control flow conditions by associating data object nodes to the control flow. The rest of the process resembles the model Figure 2.4 but includes data.

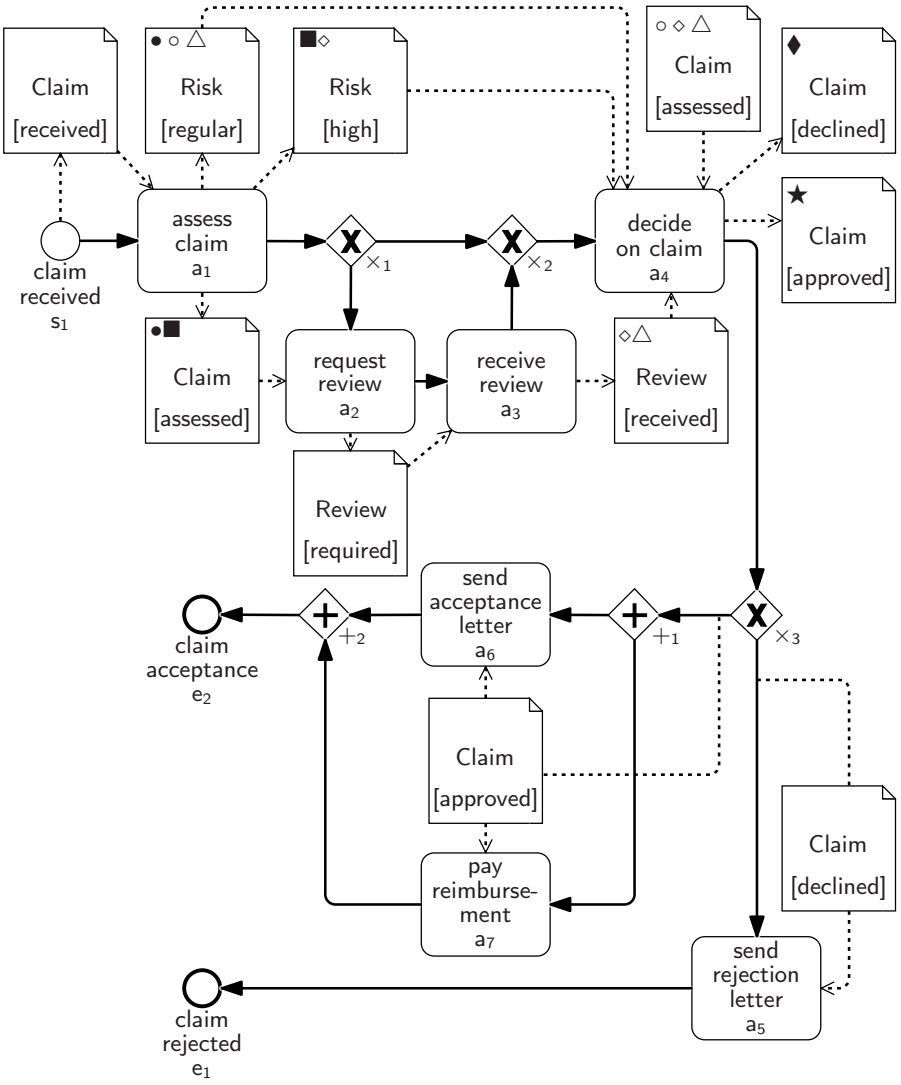


Figure 2.5: Process model diagram representing the example process of handling a claim at an insurance, including data objects, data flow, and markers for input and output sets.

The set of data object nodes DO_i in the insurance example contains all phases occurring in the process model:

$$\begin{aligned} DO_i = \{&(\{\textit{Claim}\} \times \{\textit{received, assessed, declined, approved}\}) \cup \\ &(\{\textit{Risk}\} \times \{\textit{regular, high}\}) \cup \\ &(\{\textit{Review}\} \times \{\textit{requested, received}\})\} \end{aligned}$$

In general, input and output sets are not part of the visual representation. However, we mark them with icons on the data object nodes. The function $\imath_i$ defines the input sets for each activity. Some activities, such as a_1, have a single input set, others, such as a_3, have multiple:

$$\begin{aligned} \imath_i(a_1) &= \{\{(\textit{Claim}, \textit{received})\}\} \\ \imath_i(a_4) &= \{\{(\textit{Claim}, \textit{assessed}), (\textit{Risk}, \textit{regular})\}, \\ &\quad \{(\textit{Claim}, \textit{assessed}), (\textit{Risk}, \textit{regular}), (\textit{Review}, \textit{received})\}, \\ &\quad \{(\textit{Claim}, \textit{assessed}), (\textit{Risk}, \textit{high}), (\textit{Review}, \textit{received}))\}\} \end{aligned}$$

Similarly, the start event and all activities have output sets. Some have only one, such as s_1, others have multiple, such as a_1 and a_3:

$$\begin{aligned} \mathit{o}_i(s_1) &= \{\{(\textit{Claim}, \textit{received})\}\} \\ \mathit{o}_i(a_1) &= \{\{(\textit{Claim}, \textit{assessed}), (\textit{Risk}, \textit{regular})\}, \\ &\quad \{(\textit{Claim}, \textit{assessed}), (\textit{Risk}, \textit{high})\}\} \\ \mathit{o}_i(a_4) &= \{\{(\textit{Claim}, \textit{declined})\}, \\ &\quad \{(\textit{Claim}, \textit{approved})\}\} \end{aligned}$$

The function $\imath\mathit{o}_i$ may limit input-output-set combinations. Often, however, all possible combinations are valid: Activity a_4 has multiple input sets and multiple output sets, and all combinations are allowed.

$$\imath\mathit{o}_i(a_4) = \imath_i(a_4) \times \mathit{o}_i(a_4)$$

Data-aware BPMN models define highly structured processes with simple data integration. Yet, there are limitations. First, BPMN does not include a combined semantics for process and data models, albeit possible conflicts between them. For example, a process may create objects in an order contradicting the multiplicity constraints in the data model [94]. Second, when representing a flexible process, highly structured models become complex [51].

The current model of the insurance example (Figure 2.5) is oversimplified. It assumes that all claims are complete, that at most one review is requested, and that external reviewers always respond. Furthermore, clients cannot revoke their claims. Certainly, the simplicity of the model is one of its strengths, but it is unsuited to describe all variants. Some variants may even be unknown during design. Therefore, the insurance requires more flexibility than traditional business process models offer.

2.2.2 Fragment-Based Case Management

Case management addresses the flexibility and uncertainty of knowledge-intensive processes [82]. Activities and data are organized in cases. However, definitions of cases differ [60, 115, 148]. In this thesis, we adapt the notion coined by CMMN [115, p. 5]: A case is a process instance including actions regarding a subject in a particular situation to achieve a goal. Actions include activity instances. Examples for subjects are a patient, a claim, or a citizen. The subject is in a situation such as a patient's hospital visit or a claim's pending state. The goal is to resolve the situation for the subject, e.g., to treat the patient or to handle the claim. To describe the situation accurately, a case also includes a history of past actions. We define a case formally in Definition 7.

Definition 7 (Case)**.** A case is a tuple $case = (E, Subject, H, G)$, where

1. E is the set of *enabled* actions.
2. *Subject* is a representation of the subject.
3. H is the case history.
4. G is the case goal.

The situation is encoded in the representation of the subject, the enabled actions, and the case history. ◇

The definition of a case is vague because details depend on the case management approach. For this reason, we refine the definition once we have introduced case models.

Some case management approaches are purely data-centric [60]. They describe the subject and situation by data artifacts and the goal by data conditions. In the case model, the subject's data structure is described by a data model which is sometimes called domain model or information model [115]. Furthermore, activities are defined with their data requirements and data operations.

But not all knowledge-intensive processes are purely data-driven. The observation that knowledge-intensive processes may contain both highly structured and loosely coupled parts led to the development of fCM [90, 112]. It combines techniques from data-centric and activity-centric process modeling: Highly structured control-flow-based fragments can be combined dynamically based on data conditions. An fCM model combines a data model, object behavior models, and process models. The combination of different modeling paradigms allows representing a variety of constraints and rich domain knowledge. Furthermore, the fCM approach is adaptive: If the model lacks desired behavior, new fragments can be added at runtime.

Domain Models in fCM

In fCM, a data model that describes the data objects of a case is called a domain model [112]. A domain model is a full-fledged data model with a dedicated class for the subject, which is called case object. The model

contains classes with typed attributes and associations with multiplicity constraints. However, associations and attributes are not part of the model's execution semantics [112, 117, 143]. Since we are primarily interested in the semantics, we define a simplified version of the case model without changing the behavior. An fCM domain model is a set of classes with a dedicated class for the case object which represents the subject of the case (Definition 8).

Definition 8 (fCM Domain Model)**.** An fCM domain model is a tuple $d_{fcm} = (C, c_{co})$, where

1. C is a finite non-empty set of classes.
2. $c_{co} \in C$ is the case class describing the central case object/subject.

◇

In the insurance example, the *claim* is the case object (see domain model in Figure 2.6). There is exactly one claim for each case. Furthermore, there are the classes *Review*, *Risk*, and *Decision*.

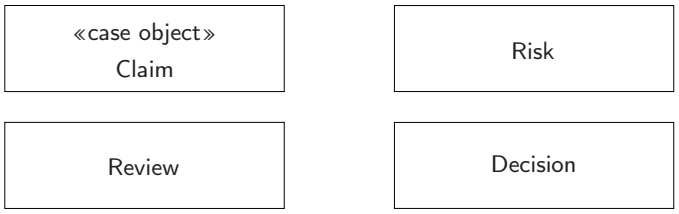


Figure 2.6: An fCM domain model for the insurance example.

The model is formally defined by the tuple $d_{\text{fcm};i} = (C_i, c_{co;i})$, where

- $C_i = \{$*Claim, Risk, Review, Decision*$\}$ is the set of classes, and
- $c_{co;i} =$ *Claim* is the case class.

Object Behavior in fCM

In fCM, the object behavior is defined by object lifecycles.[2] They specify data operations that can be performed by activities. Therefore, the object behavior constrains the process. Every fCM model contains object lifecycles that are assigned to classes.

Figure 2.7 depicts the object lifecycles for the classes in Figure 2.6. A claim starts in state *received*. It can transition to either *complete* or *incomplete*. It is possible to update a claim that is in state *incomplete* to state *update pending*. From *update pending*, the claim can change back to *incomplete* or to *complete*. A complete claim continues to be *assessed*, *in review*, and *reviewed* before reaching its final state, *archived*. The behavior of other objects can be inferred from the lifecycles.

[2]While not included here, [112] allows labeled state transitions. However, labels do not influence the case execution.

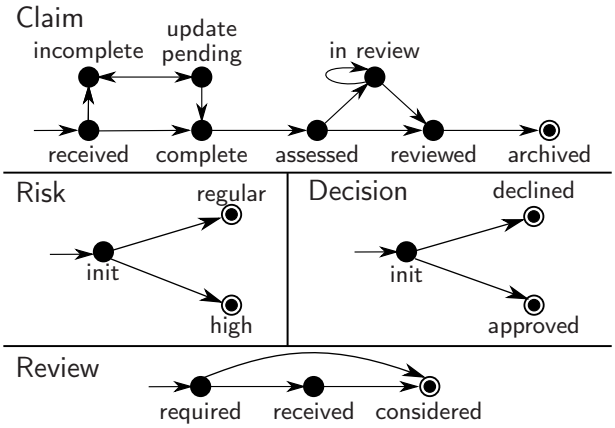


Figure 2.7: Object lifecycles for the insurance example (see classes in Figure 2.6).

Process Fragments in fCM

The case behavior is primarily defined by a set of process fragments (Definition 9). Each fragment is a highly structured control flow graph including data objects and data flow. It is similar to a data-aware process model (cf. Definition 6). However, the inputs and outputs of activities are not grouped into input sets and output sets, and fCM does not define conditional control flow.

Nevertheless, fragments are inherently data-driven. They have data-based preconditions. Furthermore, all fragments of one case operate on shared data, which allows synchronization: A fragment or one of its activities may require data that is provided by another fragment. Yet, except for the data requirements, fragments can run concurrently and repeatedly, which results in flexible behavior.

Definition 9 (Process Fragment.)**.** Let $d_{fcm} = (C, c_{co})$ be a domain model, Λ a set of object lifecycles, and $\mathit{b} : C \to \Lambda$ a function assigning each class a behavior. An fCM process fragment f is a tuple

$$f = (N, \xrightarrow{N}, DO, \mathit{read}, \mathit{write}, \mathcal{CON}_{pre}), \text{ where}$$

- $N = \{s\} \dot\cup N_A \dot\cup N_\times \dot\cup N_+ \dot\cup N_E$ is the set of control flow nodes partitioned into a start event, activities, exclusive gateways, parallel gateways, and end events, respectively.[3]
- $\xrightarrow{N} \subset N \times N$ is the control flow relation.
- DO is a set of data object nodes. Each data object $(c, q) \in DO$ is a phase (Definition 4).
- $\mathit{read} : N_A \to \mathcal{P}(DO)$ assigns each activity data object nodes that it may read.
- $\mathit{write} : N_A \to \mathcal{P}(DO)$ assigns each activity data object nodes that it may write.

[3] Hewelt and Weske [112] define start events implicitly through the precondition.

- $\mathcal{CON}_{pre} \subseteq \mathcal{P}(DO)$ is the fragment's precondition. If the precondition is not a tautology, the fragment has a *conditional start event*.

$\diamond$

Control flow defines causal dependencies among activities imperatively. Preconditions and incoming data flow define requirements of fragments and activities declaratively. By combining imperative and declarative process modeling, fCM can express loosely structured behavior more concisely than traditional process models.

Figure 2.8 depicts fragments for the insurance's claim handling process. In comparison to the process model in Figure 2.5, more variants are supported: A claim is not necessarily complete, so updates may be requested and received (fragment 2). Furthermore, an internal (fragment 3) or multiple external reviews (fragment 4) can be created.

Fragment 1 includes the core activities of the process. It can be augmented by executing fragments 2–4. When the case starts, the knowledge worker checks whether the claim is complete. If it is incomplete, fragment 3 may be executed once or multiple times (sequentially). Once the claim is complete, fragment 1 continues by assessing the risk. If the risk is *regular*, one internal review is created and completed (fragment 3). If the risk is high, one or more external reviews may be requested, received, and considered (fragment 4). External reviews can be handled concurrently. Once the knowledge worker decides that sufficiently many reviews have been considered, the claim is changed to state *reviewed* and fragment 1 continues: The knowledge worker decides whether to approve or decline the case and continues the process, respectively.

To demonstrate the formalization, consider fragment 3. The fragment is formally defined by the tuple f_3:

$$f_3 = (N_3, \xrightarrow{N_3}, DO_3, \mathit{read}_3, \mathit{write}_3, \mathcal{CON}_{pre;3}), \text{ where}$$

- $N_3 = \{s_3, \mathit{create\ review}, \mathit{complete\ review}, e_3\}$ is the set of control flow nodes with start event s_3 and end event e_3.
- $\xrightarrow{N_3}$ is the control flow relation.

$$\begin{aligned}\xrightarrow{N_3} = \{&(s_3, \mathit{create\ review}), (\mathit{create\ review}, \mathit{complete\ review}),\\ &(\mathit{complete\ review}, e_3)\}\end{aligned}$$

- DO_3 is the set of data object nodes.

$$\begin{aligned}DO_3 = &(\{\mathit{Claim}\} \times \{\mathit{assessed}, \mathit{in\ review}, \mathit{reviewed}\}) \cup\\ &(\{\mathit{Review}\} \times \{\mathit{required}, \mathit{completed}\})\end{aligned}$$

- read_3 is a function assigning activities their inputs.

$$\begin{aligned}\mathit{read}_3(\mathit{create\ review}) &= \{(\mathit{Claim}, \mathit{assessed})\}\\ \mathit{read}_3(\mathit{complete\ review}) &= \{(\mathit{Claim}, \mathit{in\ review}), (\mathit{Review}, \mathit{required})\}\end{aligned}$$

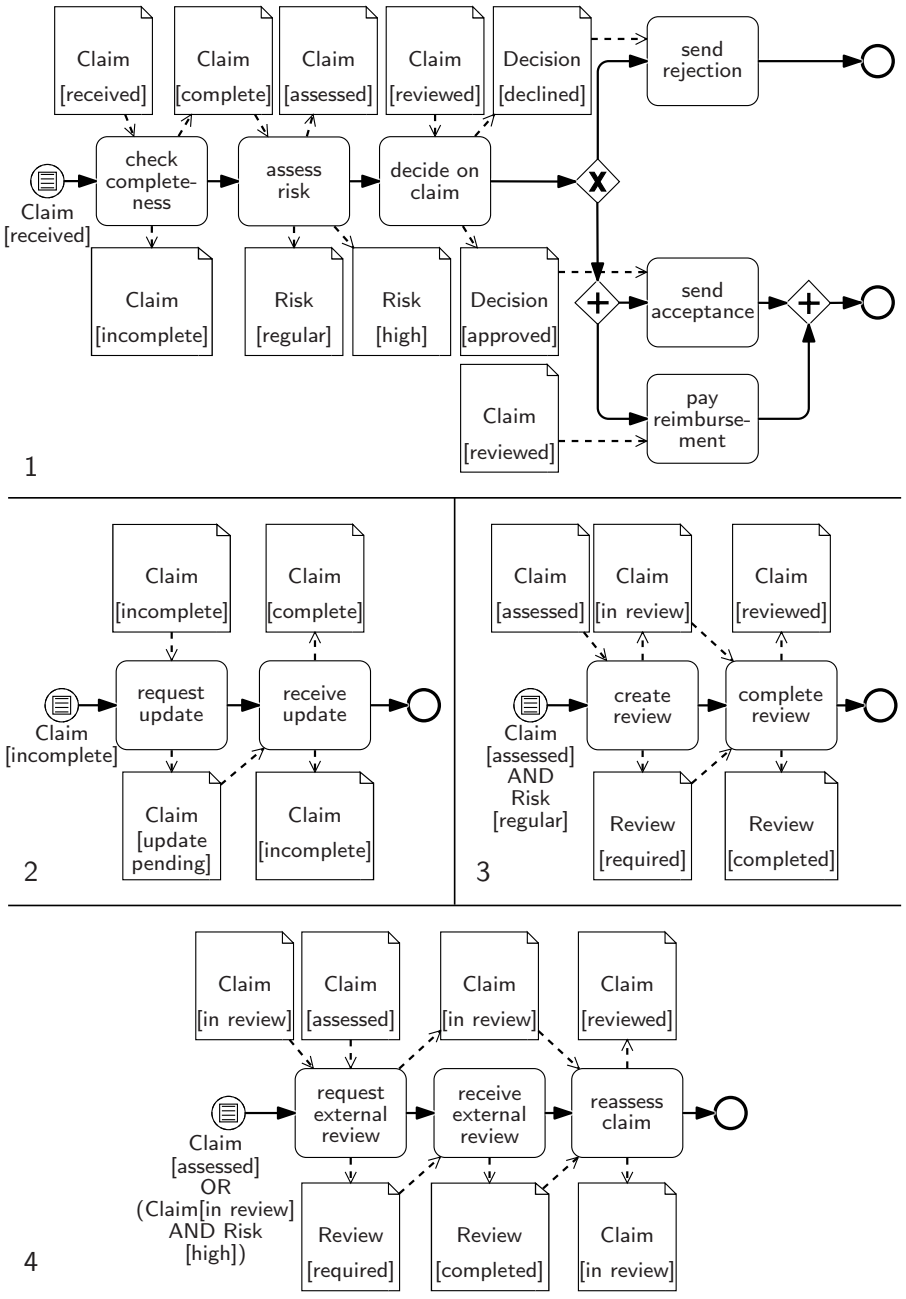


Figure 2.8: Process fragments for the insurance example. Fragment 1 handles the main steps, fragments 2 incomplete claims and fragments 3 and 4 reviewing.

- $write_3$ assigns each activity their outputs.

$$write_3(\textit{create review}) = \{(\textit{Review}, \textit{required}), (\textit{Claim}, \textit{in review})\}$$
$$write_3(\textit{complete review}) = \{(\textit{Claim}, \textit{reviewed}), (\textit{Review}, \textit{completed})\}$$

- $\mathcal{CON}_{pre;3} = \{\{(\textit{Claim}, \textit{assessed}), (\textit{Risk}, \textit{low})\}\}$ is the fragment's precondition.

Goal Specifications in fCM

In traditional process models, goals are represented by end events. In fCM each fragment has end events, but reaching one is not directly related to the goal of the case. Instead, a case model's goal is defined by a termination condition (Definition 10). Knowledge workers may close the case if the condition evaluates to true.

Definition 10 (Termination Condition)**.** Let $d_{fcm} = (C, c_{co})$ be a domain model, Λ a set of object lifecycles, and $\mathfrak{b} : C \to \Lambda$ a function assigning each class a behavior. The set $DO = \bigcup_{c \in C} (\{c\} \times \mathfrak{b}(c).Q)$ contains all phases. The termination condition $\mathcal{CON}_{term} \in \mathcal{P}(\mathcal{P}(DO))$ is a data condition (see Definition 6). ◇

The insurance example has two possible outcomes. The claim is approved or declined. In the model, these outcomes are represented by the state of the decision. The termination condition is defined accordingly:

$$\mathcal{CON}_{term;i} = \{\{(\textit{Decision, declined})\}, \{(\textit{Decision, approved})\}\}$$

It can be interpreted as a predicate logic formula:

$$\mathcal{CON}_{term;i} \equiv \textit{Decision}[\textit{approved}] \vee \textit{Decision}[\textit{declined}]$$

If the case contains a decision in state *approved* or *declined*, knowledge workers *can* close the case.

The fCM Case Model

An fCM case model (Definition 11) combines the elements specified before. In addition, it contains a selection of classes that are instantiated when a new case begins. All parts of the case model are connected. Conditions and data object nodes use classes and states defined in the domain model and the object lifecycles, respectively. Constraints of the domain model and lifecycles apply to the case behavior: Each case must instantiate the case class exactly once, and activities can only perform state changes that are specified in the object lifecycles.

Definition 11 (fCM Case Model)**.** An fCM case model *fcm* is a tuple $\textit{fcm} = (d_{fcm}, \Lambda, \mathfrak{b}, C_0, F, \mathcal{CON}_{term})$, where

1. $d_{fcm} = (C, c_{co})$ is a domain model.
2. Λ is a set of object lifecycles.
3. $\mathfrak{b} : C \to \Lambda$ is a function assigning each class c its behavior.

4. $C_0 \subseteq C$ is a set of classes that are instantiated during case initialization.
5. F is a set of process fragments.
6. $\mathcal{CON}_{term}$ is the termination condition.

All data object nodes and all conditions contain only phases specified by the domain model's classes and their object behavior. ◇

The case model for the insurance example consists of the domain model in Figure 2.6, the object lifecycles in Figure 2.7, the set $C_{0;i}$ containing *Claim*, *Risk*, and *Decision* of classes that are instantiated during case initialization, the fragments in Figure 2.8, and the termination condition $\mathcal{COND}_{term;i}$.

Cases in fCM

With fCM's definition of a case model, we can refine our definition of a case. A case comprises a subject, a situation, actions, and a goal. In fCM, the subject is described by the case object and other auxiliary objects. Each belongs to one class and has one state. The actions include starting fragment instances and executing activity instances that operate on data objects. The goal is specified by the termination condition. Definition 12 defines this formally. Knowledge workers have different options to evolve a case: They can perform enabled actions to change the current situation, or they can add new fragments to the model.

Definition 12 (fCM Case)**.** Let $fcm = (d_{fcm}, \Lambda, \flat, C_0, F, \mathcal{CON}_{term})$ be a case model. A corresponding case is defined as follows:

1. The *Subject* described by a set of data objects O, where[4]
   a) $o.class$ is the class of object $o \in O$.
   b) $o.state$ is the state of object $o \in O$.
2. A set $E \subseteq (F \cup (\mathcal{P}(O) \times N_{A;F} \times \mathcal{P}(\mathcal{O})))$ of enabled actions, where $\mathcal{O}$ is the universe of possible objects and each action is either a fragment $f \in F$ that can be instantiated or a tuple (O_r, n_a, O_w) consisting of
   a) a set O_r containing data objects that are read and
   b) a set O_w of data objects that are written
   c) by an instance of activity $n_a \in N_{A;F} = \left(\bigcup_{f \in F} f.N_A\right)$.
3. A history H of past actions.
4. The goal $G = \mathcal{CON}_{term}$.

◇

[4] We use the period to access members of a tuple. Given a collection T of tuples, where each tuple $t \in T$ has the structured (a, b, c), let t_1 be a tuple $t_1 = (a_1, b_1, c_1) \in T$. Instead of a_1 we also write $t_1.a$.

A Sketch of fCM's Semantics

A case is instantiated, advanced, and eventually closed. Meanwhile, all parts of the case model combined define the space of actions for the knowledge workers. To illustrate this, we follow the knowledge workers of the insurance through a case of the claim handling example.

Mr. Miller submits a new claim to his insurance. Upon receiving the claim, a respective case is instantiated. One object for each of the classes claim, risk, and decision is created. Each of these object is in its initial state, and no activities have been executed yet.

Only the precondition of fragment 1 is enabled. A knowledge worker can start a respective fragment instance. Since Mr. Miller has provided all the necessary information, the knowledge worker marks the claim as complete. Next, the worker assesses the risk. Due to the history of Mr. Miller and the comparably low amount of the claim, the worker considers the risk to be regular. The claim is changed to state *assessed*.

The next activity of fragment 1 is not yet enabled, but the precondition of fragment 3 is satisfied. The knowledge worker instantiates fragment 3, and an internal reviewer creates a review. Together with the knowledge worker, the review is completed—the claim changes to state reviewed. This enables activity "decide on claim" in fragment 1. It changes the decision's state to either *declined* or *approved*. The claim of Mr. Miller is approved, and the knowledge worker completes the case by informing Mr. Miller and reimbursing him. Afterwards, she closes the case.

The fCM case model does not contain explicit input sets, output sets, and input-output-set combinations. Yet, the sets and their combinations exist implicitly. An activity can access at most one object for a certain class. Activity "decide on claim", for example, reads and writes one decision object, although two respective data object nodes are connected via outgoing data flow. Activities that have multiple data object nodes for one class connected by incoming (or outgoing) data flow may read (or write) only one respective object. Input sets and output sets result from these alternatives. Subsequently, "decide on paper" has one input set and two output sets. Furthermore, the combination of input and output sets are limited by the object behavior.

Figure 2.9 shows an example of this. Activity *a* may read a *C1* object in state *s1* or *s2* and a *C2* object in state *z1*. This results in the two input sets $\{C1[s1], C2[z2]\}$ and $\{C1[s2], C2[z2]\}$. The activity may write the *C1* object in either state *s2* or *s3*.This results in the two output sets$\{C1[s2]\}$ and $\{C1[s3]\}$. Based on *C1*'s object lifecycle, we know that its state can only change from *s1* to *s2* and from *s2* to *s3*. Thus, we have the following input-output-set combinations:

- input set $\{C1[s1], C2[z2]\}$ with output set $\{C1[s2]\}$ and
- input set $\{C1[s2], C2[z2]\}$ with output set $\{C1[s3]\}$.

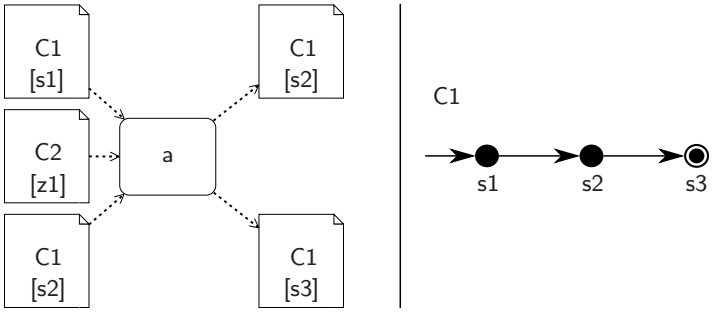


Figure 2.9: An example of implicitly defined input sets, output sets, and input-output-set combinations.

2.3 Petri Nets & Formal Execution Semantics

The specification and implementation of information systems requires both domain knowledge and IT expertise. Models can aid the communication among respective experts. Furthermore, models can be verified, analyzed, and implemented if they have a precise semantics. But common process modeling languages, such as BPMN, define informal or semiformal semantics, e.g., by using textual descriptions. In some cases, this can be insufficient. Formal models, by contrast, have mathematically defined execution semantics, which makes them suitable for verification, analysis, and implementation. Arguably, the most common formal models for business processes are Petri nets.

2.3.1 Petri Nets

A Petri net [2] is a directed bipartite graph consisting of places, transitions, and arcs (Definition 13). Places may hold tokens. The distribution of tokens on places is called *marking*. It is the state of a Petri net. The marking changes by *firing* a transition, which removes tokens from and produce tokens into places.

Definition 13 (Petri Net)**.** A Petri net pn is defined by a tuple

$$pn = (S, T, \xrightarrow{ST}, m_0), \text{ where}$$

1. S is a finite set of places.
2. T is a finite set of transitions-
3. $\xrightarrow{ST} \subseteq ((S \times T) \cup (T \times S))$ is a set of arcs connecting places and transitions.
4. $m_0 : S \to \mathbb{N}_0$ is the initial marking. It is a function assigning each place a number of initial tokens.

Furthermore, we define the concepts preset and postset:

1. The preset $\bullet t$ of a transition $t \in T$ is the set of all places that have an outgoing edge leading to t:

$$\bullet t = \{s \in S | s \xrightarrow{ST} t\}$$

2. The postset $t\bullet$ of a transition $t \in T$ is the set of all places that have an incoming edge starting in t:

$$\bullet t = \{s \in S | t \xrightarrow{ST} s\}$$

$\diamond$

The Petri net's structure does not define its semantics. Instead, different semantics can be assigned to the same model. We employ additional rules to specify the semantics clearly. The rules must define valid markings and the enablement and firing of transitions.

In the semantics that we use, places can hold any number of tokens. A transition consumes one token from each place in its preset and produces one token into each place in its postset:

1. A transition is enabled if there is at least one token in each place of its preset:

$$t\ is\ enabled\ in\ m \Leftrightarrow \forall s \in \bullet t : m(s) > 0$$

2. When a transition fires, it removes a token from each place in its preset and produces a token into each place in its postset. Let t be a transition enabled in m, the marking m' is reached by firing t (written as $m \xrightarrow{t} m'$):

$$m'(s) = \begin{cases} m(s) - 1 & \text{if } s \in \bullet t \setminus t\bullet \\ m(s) + 1 & \text{if } s \in t\bullet \setminus \bullet t \\ m(s) & \text{otherwise} \end{cases}$$

By firing a transition, the marking may change and so may the set of enabled transitions. Thus, a sequence of transitions leads to a sequence of markings. All markings that can be reached from a Petri net's initial marking and the transitions leading from one marking to another, describe the net's state space which is called *reachability graph* (Definition 14). The nodes of the graph are markings, and the edges represent the firing of transitions.

Definition 14 (Reachability Graph)**.** Let $pn = (S, T, \xrightarrow{ST}, m_0)$ be a Petri net. Let m and m_i be two markings of pn. Marking m_i is reachable from m (written as $m \xrightarrow{*} m_i$) if

- $m = m_i$ or
- $\exists t_1, \ldots, t_n \in T, n \geq 1 : m \xrightarrow{t_1} m_1 \ldots \xrightarrow{t_n} m_i$.

The reachability graph $rg(pn) = (M, \xrightarrow{M})$ is the Petri net's state space, where

1. $M = \{m | m_0 \xrightarrow{*} m\}$ is the set of markings reachable from the initial marking m_0.

2. $\xrightarrow{M} \subseteq M \times T \times M$ is the set of state transitions:

$$(m, t, m') \in \xrightarrow{M} \Leftrightarrow m \xrightarrow{t} m'$$

$\diamond$

We can model business processes as Petri nets. Transitions represent events, activities, and gateways. Places model data object nodes and control flow. Tokens represent that a respective data objects exist or that the corresponding control flow has been triggered. Just as activities, events, and gateways require and update data objects and just as they are enabled by control flow, transitions consume tokens from and produce tokens into respective places.

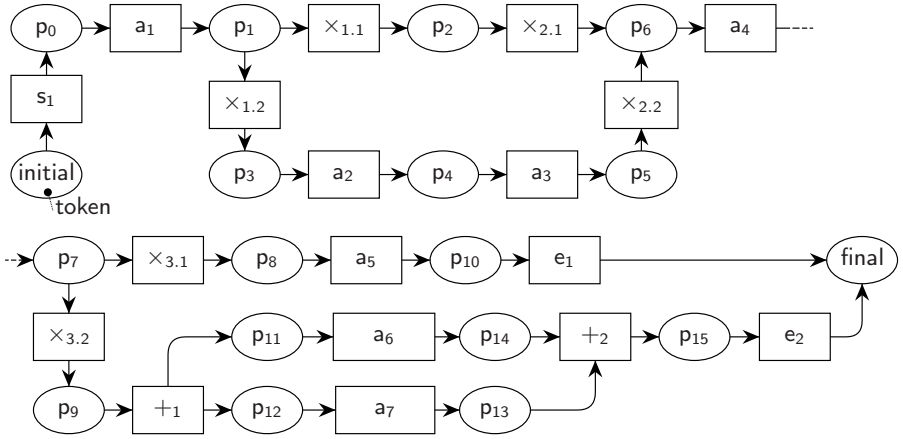


Figure 2.10: Petri net for the process modeled in Figure 2.4. Places are represented by ovals, transitions by rectangles.

The Petri net in Figure 2.10 models the insurance process without data (cf. BPMN, Figure 2.4). A token in place *initial* represents a process instance that has not started. A token in *final* represents a terminated instance. The place $p_0, p_1, \ldots p_{15}$ represent control flow arcs. If a token is in such a place, the corresponding control flow has been triggered, but the subsequent control flow node has not yet terminated.

Transitions represent activities, gateways, or events. However, a transition fires instantaneously, while an activity takes time: Its execution begins at one point, and it terminates at another. Thus, the Petri net in Figure 2.10 abstracts from the *running* state of activities. The firing of a transition represents the activity's (gateway's, or event's) termination.

The Petri net describes the process unambiguously [15]. The initial marking has one token on place *initial*. In this marking, transition s_1—the start event—is enabled. When an enabled event, activity, or gateway terminates, the outgoing control flow is triggered. Accordingly, firing transition s_1 removes a token from place *initial* and produces one into p_0. Now, transition a_1 is enabled. Eventually, one of the end events e_1 or e_2 is enabled, fires, and the process completes. In the final marking, there is one token on the place *final* and no transitions are enabled. We

can consider all possible sequences of transitions and the sequence of markings they produce. This results in the Petri nets reachability graph (see Figure 2.11), which is the state space of the modeled process.

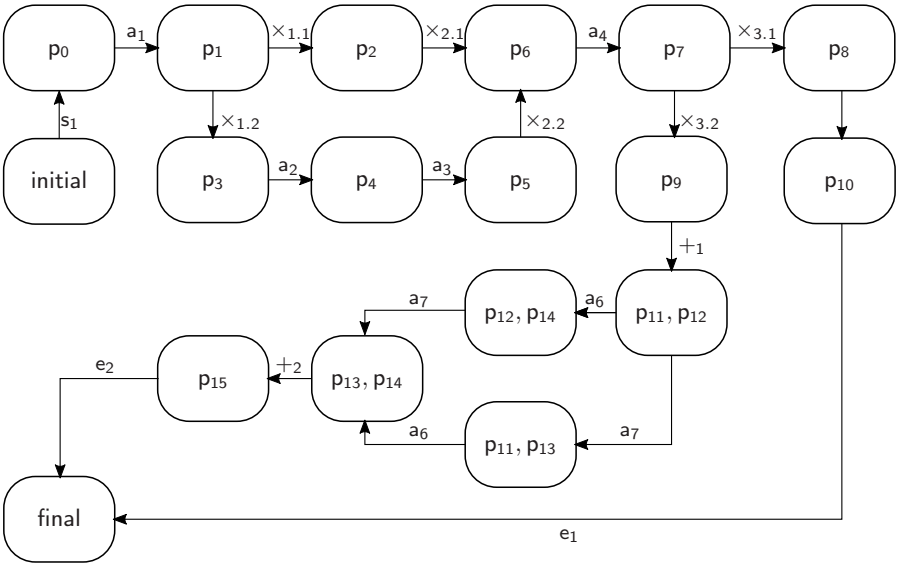


Figure 2.11: Reachability graph for the Petri net in Figure 2.10. The label of the place describes the distribution of tokens in the net.

Tokens in a Petri net may represent different concepts, such as a triggered control flow or a data object. The place holding a token determines its domain-specific semantics, but tokens in the same place cannot be distinguished. This limits the use of classical Petri nets.

The Petri net in Figure 2.10 cannot represent multiple concurrent process instances accurately. Given two process instances, we can reach a marking with one token in each of the places p_{11}, p_{12}, p_{13}, and p_{14}. However, we cannot know whether the tokens in p_{13} and p_{14} belong to the same process instances, and whether gateway $+_2$ should be enabled.

Nevertheless, Petri nets are often sufficient to model a single instance of a highly structured process: These processes usually instantiate classes at most once and iterate loops sequentially. In other words, each place of the corresponding Petri net holds at most one token. However, these assumptions do not hold for flexible data-centric processes.

2.3.2 Colored Petri Nets

Colored Petri nets are data-aware. Tokens are data items, and places are typed, so they can only hold tokens of a particular type. When a transition fires, it does not merely consume and produce tokens. Instead, tokens and values derived from tokens are bound to variables. Variables can be used in arc expressions to calculate other tokens that are consumed or produced. Furthermore, variables may be used in a guard condition that determines whether the transition is enabled.

In colored Petri net jargon, a data type is called *colorset*, and a value is a *color*. Colors can be atomic, structured (e.g., functions or tuples), or collections (e.g., sets, multisets (Definition 15), or lists). Furthermore, colored Petri nets use an expression language for arc expressions and guards. Guards evaluate to boolean and arc expressions to multisets or colors that belong to the colorset of the connected place.

To summarize, a colored Petri net (Definition 16) is a Petri net extended with a set of colorsets. A set of variables. A typing function for places. A guard for each transition. An arc expression for each arc. And an initialization expression for each place. The initialization expressions define the initial marking of the net and must not use variables. The marking of a colored Petri net assigns each place a multiset of colors.

Definition 15 (Multiset)**.** A multiset (cf. [56, pp. 82–83, 137, p. 15]) is an unordered collection that may contain the same value multiple times. Formally, a multiset X_{ms} over a non-empty set Z is a function mapping each value in Z to a natural number $X_{ms} : Z \to \mathbb{N}_0$. Given a value $v \in Z$, $X_{ms}(v)$ denotes how often v occurs in X_{ms}. We also write $\{a, a, b, c\}_{ms}$ for the multiset containing a twice and b and c once.

Given two multisets X_{ms} and Y_{ms} defined over the same set Z, we define the following operations:

1. $z \in X_{ms} \Leftrightarrow X_{ms}(s) > 0$
2. $X_{ms} \subseteq Y_{ms} \Leftrightarrow \forall z \in Z : X_{ms}(z) \leq Y_{ms}(z)$
3. $X_{ms} \cup X_{ms} = \{(z, \mathit{max}(X_{ms}(z), Z_{ms}(z)))|z \in Z\}$
4. $X_{ms} \cap X_{ms} = \{(z, \mathit{min}(X_{ms}(z), Z_{ms}(z)))|z \in Z\}$
5. $X_{ms} \setminus X_{ms} = \{(z, \mathit{max}(0, X_{ms}(z) - Z_{ms}(z)))|z \in Z\}$
6. $X_{ms} + X_{ms} = \{(z, X_{ms}(z) + Z_{ms})|z \in Z\}$
7. $|X_{ms}| = \sum_{z \in Z} X_{ms}(z)$

◇

Definition 16 (Colored Petri Net [56])**.** A colored Petri net *CPN* is a tuple $\mathit{CPN} = (S, T, \xrightarrow{ST}, \Sigma, V, c, g, e, \mathit{init})$, where

1. S is a finite set of places.
2. T is a finite set of transitions.
3. $\xrightarrow{ST} \subseteq ((S \times T) \cup (T \times S))$ is a set of arcs.
4. Σ is a finite set of non-empty colorsets.
5. V is a finite set of typed variables so that $\mathit{type}(v) \in \Sigma$ for all variables $v \in V$.
6. $c : S \to \Sigma$ is a function assigning each place a colorset (type).

7. $g : T \to EXPR_V$ is a function assigning each transition a guard expression, which may use variables in V and evaluates to boolean ($type(g(t)) = \{true, false\}$ for all transitions $t \in T$).

8. $e : \xrightarrow{ST} \to EXPR_V$ is a function assigning each arc an expression, which may use variables in V and evaluates to a multiset of the colorset of the arc's place: $type(e(a)) = c(s)_{MS}$ for all arcs $a \in A$ where s is the place connected to a, and $c(s)_{MS}$ is the set of multisets over the colorset $c(s)$.

9. $init : S \to EXPR_\emptyset$ is a function assigning an expression that uses no variables to each place to describe the initial marking (multiset of initial tokens): $type(init(s)) = c(s)_{MS}$ for all places $s \in S$.

◇

In colored Petri nets, tokens with a different color can be distinguished. By extending the Petri net in Figure 2.10 to a colored Petri net, it can support multiple process instances. In the colored Petri net in Figure 2.12, all places have the colorset $\mathbb{N}_0$. Each process instance is identified by a number. Initially, no running instances exist and the place *initial* has a token with value 0. When the start event s_1 occurs, the token in *initial* is consumed, its value binds to the variables cnt (arc expression) and i (guard). Two tokens are produced: A token with the value of variable i is produced into p_0, and a token with the value $cnt + 1$ is produced into *initial*. All other transitions require that all consumed and produced tokens have the same value. Therefore, a firing of a transition affects only a single process instance. Transition $+_2$, for example, is only enabled if there is at least one token in p_{13}, one token in p_{14}, and both tokens have the same color. Yet, the second term of the guard condition $cnt < 2$ limits the model to two instances.

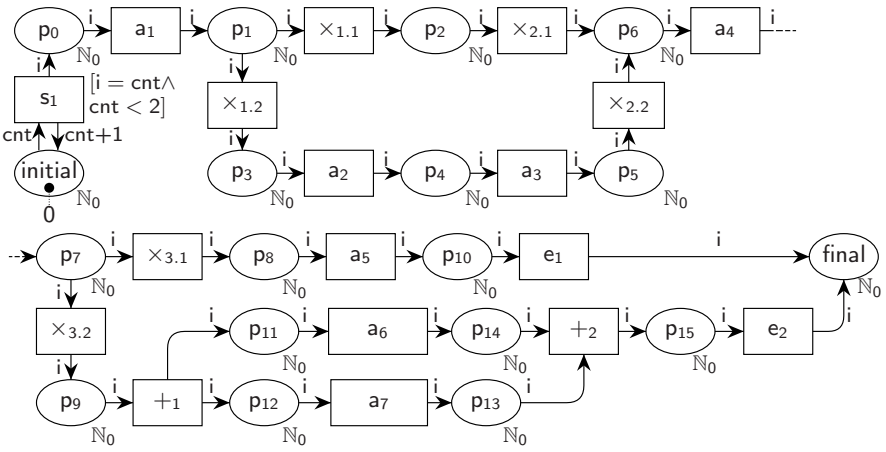


Figure 2.12: Colored Petri net of the insurance example. Process instances are represented by integers, allowing multiple instances without mixing their execution.

To describe the semantics of a colored Petri net formally, we must define additional concepts (see Definition 17). A *marking* assigns each place a multiset of tokens (colors). When a transition fires, arc expressions and guards determine which tokens are consumed. Thereby, values are bound to variables. Given such a *binding*, the guard must evaluate to true. A *binding element* combines the transition with one corresponding binding. Yet, a marking may support multiple valid bindings for one transition.

Definition 17 (Colored Petri Net Concepts [56])**.** Given a colored Petri net $CPN = (S, T, \xrightarrow{ST}, \Sigma, V, c, g, e, init)$, we define

1. A *marking* is a function m that assigns each place $s \in S$ a multiset of tokens $m(s) \in c(s)_{MS}$.
2. For a transition $t \in T$, $var(t)$ is the set of *variables* used in the guard of t and in the expressions of arcs that are connected to t.
3. A *binding* of a transition t is a valuation function $binding$ that assigns each variable $v \in var(t)$ a value in $type(v)$. Furthermore, $\mathcal{B}(t)$ is the set of all bindings for a transition t.
4. A *binding element* is a pair $(t, binding)$ with a transition $t \in T$ and a binding $binding \in \mathcal{B}(t)$. Furthermore, $BE(t)$ is the set of all binding elements of t and BE the set of all binding elements.

$\diamond$

The behavior of a colored Petri net depends on the binding elements (see Definition 18).

Definition 18 (Colored Petri Net Firing Rule)**.** Given a colored Petri net $CPN = (S, T, \xrightarrow{ST}, \Sigma, V, c, g, e, init)$ and a marking m. A transition $t \in T$ is enabled in m if there exist a $binding \in \mathcal{B}(t)$, that satisfies the guard and arc expressions:

1. The guard of transition t is true for the $binding$.
$$g(t)(binding) = true$$
2. The expressions $e(s,t)$ for places $s \in {\bullet}t$ evaluate for $binding$ to a multiset of tokens. This multiset must be contained in the place s.
$$\forall s \in {\bullet}t : e(s,t)(binding) \subseteq m(s)$$

$\diamond$

Consider transition s_1 of the example in Figure 2.12. Corresponding bindings assign a value to the variables cnt and i. When s_1 fires for the first time, the binding assigns the value 0 to both variables. The second time, the value is 1. Afterwards, there exist no binding for which the two requirements of the firing rule hold: If we choose value 2 for i and cnt, the second term of the guard is false. If we choose values 0 or 1, the guard is true, but no respective token exists in place *initial*.

The reachability graph of a colored Petri net (Definition 19) consists of markings and binding elements that lead from one marking to another. The firing of a transition for different bindings may lead to different successor states and are treated separately. In the example (Figure 2.12), a marking captures the state of all running process instance. A binding element marks the firing of a transition for one instance.

Definition 19 (Reachability Graph of Colored Petri Nets)**.** Given a colored Petri net $CPN = (S, T, \xrightarrow{ST}, \Sigma, V, c, g, e, init)$, its reachability graph $rg(CPN)$ is a tuple $rg(CPN) = (M, \xrightarrow{M})$, where

1. M is the set of markings reachable from $init$.
2. $\xrightarrow{M} \subseteq M \times \text{BE} \times M$ is the set of state transitions. A transition is a tuple $(m, (t, binding), m') \in \xrightarrow{M}$ that states, the marking changes from m to m' by firing t with $binding$.

$\diamond$

In our colored Petri net example, all process instances are fully independent of one another. The net's state space is exponentially larger than the one of the classical Petri net. The reachability graph of the classical Petri net has 18 states. The reachability graph of the colored Petri net that supports two process instances has 323 $(= 18^2 - 1)$. Therefore, we refrain from depicting the full reachability graph. Figure 2.13 shows the partial reachability graph. Each state describes the token in each place. Each state transition is a binding element.

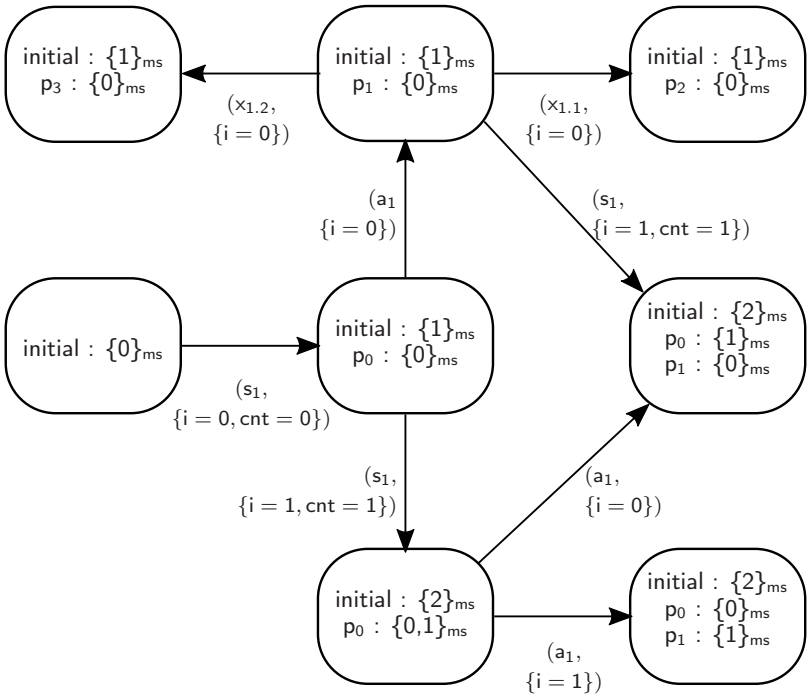


Figure 2.13: The first couple of states of the reachability graph of the colored Petri net in Figure 2.12. Empty places are omitted from the states.

3 Related Work

In knowledge work, both data and processes play an important role. Based on data, knowledge workers perform activities and make decisions that shape the process. Therefore, our research questions focus on the mutual relationship between flexible processes and data. To answer these questions, we develop a novel case management approach with a formal semantics to support knowledge-intensive processes.

Yet, the role of data in business processes has already been researched in the past, and modeling approaches for knowledge-intensive processes have been proposed: There exist works that extend existing modeling languages with data support [76, 127], that develop novel languages [147], and that formalize existing ones [81, 84]. Many of the results of such works have affected our work directly or indirectly.

For this reason, we summarize related work and describe its relationship to this thesis. We present the role of data in highly structured and well-defined process models (Section 3.1). Next, we focus on knowledge-intensive processes, their characteristics, and respective modeling approaches, which are further divided into declarative, data-centric, and hybrid languages (Section 3.2). Then, we present various works on formalizing data-aware processes—both traditional ones and knowledge-intensive ones (Section 3.3). We conclude this chapter with an overview on the most influential works (Section 3.4).

3.1 Data and Traditional Processes

Processes contain activities that are executed in coordination to achieve a business goal [149]. During execution, data is read, created, and updated. However, early process modeling approaches, such as Event-Driven Process Chain (EPC) [16] and early versions of Yet Another Workflow Language (YAWL) [28], focus solely on control flow. Nowadays, most languages also support data to different degrees.

BPMN version 1.0 [41] supports reading and writing data objects, but this does not influence the execution semantics. In reality, data requirements and data operations may shape the process behavior significantly. Various languages support variables to control the ordering of activities [24, 27, 40]. Furthermore, activities may read and update the value of such variables. ADEPT [13], for example, augments the control flow perspective with data requirements and data outputs of activities. On an implementation layer, YAWL can use variables to determine the number of instances for an activity [27]. Since version 2.0, BPMN [92] contains similar mechanisms. The data flow is refined through input and output sets, and the importance of data on the process-level is acknowledged [92, p. 202]. Furthermore, there can be

collections of objects that can be processed by multi-instance activities or multi-instances subprocesses.

In other approaches, data is considered in even greater detail and plays a central role. Hull et al. present Vortex [18], a declarative and data-driven process modeling approach. Activities have data preconditions and data outputs. They can be executed, if their preconditions are satisfied. MERODE [94, 170] is an integrated approach to enterprise modeling. A so called existence-graph is a data model that uses associations with multiplicity constraints to define an order in which objects may be created. However, MERODE's multiplicity constraints are limited. The lower bounds are 0 or 1 and upper bounds are 1 or unbounded. Furthermore, an object-event table defines events that are passed to objects and whether they create, update, or delete data objects. Activities of the process model may trigger these events and thereby affect objects. Meyer et al. [76] extend BPMN process models by adding primary and foreign key information to data object nodes. These implementation details are used to derive database queries and to represent links between objects explicitly. Queries are also used by Calvanese et al. [139], who present an approach for modeling and analyzing data-aware processes using only relational databases and SQL queries. The full state of a process instance, including all data objects, is represented by tuples in the database. In [91], Meyer et al. investigate the relationship between business processes and object lifecycles (data behavior). The authors propose a method to detect conflicts between the two. Combi et al. [127] integrate BPMN process models with UML class diagrams. They define activity views, which specify the classes and associations which are inserted, deleted, or updated by the activity. Based on the activity views, conflicts—such as data that is read after it has been deleted—can be detected.

In this thesis, we integrate data models describing structure and behavior with process models. Thereby, we leverage results from related work: We use activities with input- and output-sets [92] and domain models with existential associations [94], but multiplicity constraints can have lower and upper bounds greater than one. We integrate the process model with the domain model [94, 127] and with the object behavior [76]. However, we focus on knowledge-intensive processes and consider the integration essential for the process and not just an implementation detail.

3.2 BPM for Knowledge-Intensive Processes

Knowledge workers are primarily concerned with non-routine work that is dominated by activities for creating, maintaining, and applying knowledge to identify and solve problems [31]. Typical tasks include planning and decision-making. According to [53], knowledge workers made up 25–40% of the Canadian and US-American workforce in 2005.

3.2.1 Characterizing Knowledge-Intensive Processes

Business processes for knowledge work are called knowledge-intensive processes. They differ substantially from traditional business processes. Most prominently, they are described as unpredictable and unrepeatable [72, 98]. Both properties stem from the fact that instances emerge gradually based on knowledge workers' decisions [103]. Furthermore, involved knowledge may be too large and may change too quickly to be modeled fully. We define knowledge-intensive processes as multi-variant human-centered processes that are primarily shaped during execution by informed decisions of knowledge workers. Knowledge-intensive processes may differ from case to case.

Di Ciccio et al. [98] describe characteristics of knowledge-intensive processes and derive requirements for respective BPM approaches. Accordingly, knowledge-intensive processes have the following characteristics:

C1—Knowledge-Driven: Knowledge-intensive processes are driven by available data and data requirements.

C2—Collaboration Oriented: They are executed by humans with different roles and responsibilities.

C3—Unpredictable: The context and environment of the process impact the process execution but cannot be known beforehand.

C4—Emergent: The exact sequence of actions is unknown at design-time and emerges gradually during execution.

C5—Goal Oriented: Every process serves a goal, which may be decomposed into partial and/or intermediate goals.

C6—Event-Driven: Knowledge-intensive processes depend on the process environment and may need to react to events.

C7—Constraint and Rule-Driven: Knowledge-intensive processes may be regulated, i.e., by laws and guidelines.

C8—Non-repeatable: Cases differ from each other since the process execution depends highly on the subject and situation.

Supporting knowledge-intensive processes requires a paradigm shift. Case management is a BPM paradigm aiming at knowledge-intensive processes [82]. At its core, is the notion of a case, e.g., a claim at an insurance, a lawsuit, or the treatment of a single patient at a hospital [147]. The process evolves around, and all data is related to the case. Executing an activity means advancing the case. Case management is usually adaptive, i.e., it is possible to deviate from a model. While some approaches focus only on case management [60, 112], others [66, 129] support it without being limited to it.

3.2.2 Modeling Knowledge-Intensive Processes

Activity-centric control-flow-based modeling languages are considered unsuited for knowledge-intensive processes [72, 98]. Purely imperative models cannot capture the flexibility of knowledge work. Novel approaches promise to overcome this challenge by taking on new perspectives. Generally, there are two main developments: Modeling i) shifts from imperative to declarative and ii) becomes more data-centric.

Imperative process models define valid sequences of activities procedurally, i.e., using control flow. Declarative models define constraints and requirements for valid sequences. Activity-centric process models define activities and their effects: The next action depends primarily on the activities that have been executed in the past. In data-centric process models, the states and state changes of data artifacts dictate the process execution. Yet, models are not necessarily purely imperative or declarative, nor are they either data-centric or activity-centric. In this section, we present declarative, data-centric, and hybrid approaches. For each of these categories, we summarize the influence on our work.

Declarative Process Modeling

Purely imperative process models limit processes to a few fully defined variants. Declarative approaches are less structured and favor flexibility [51]. The most common declarative activity-centric languages are DECLARE [43] and DCR Graphs [59]. They define temporal constraints between activities using Linear Temporal Logic (LTL) on finite traces [5, 68], which extends traditional logic with temporal operators such as *always*, *eventually*, *until*, and *next*. All traces that comply with all modeled constraints are valid. Processes that have only a few constraints, many variants, or those that are kept purposely underspecified and rely on human decisions can be modeled concisely.

Both DECLARE and DCR Graphs models consist of annotated activities and connectors. An annotation can, for example, identify the initial activity of a case. Connectors may constrain the ordering of multiple activities, for example, A has to happen before B.

The DECLARE model in Figure 3.1 describes a flexible version of the insurances' claim handling. It has the following constraints (cf. to numbers in the model):

1. First, the claim must be assessed (a_1).
2. Reviews can be created (a_2) and must be followed by a decision (a_3). However, a single decision can follow multiple reviews.
3. Decisions must eventually be followed by a notification (a_4 or a_5).
4. Decisions can only be made before a notification (a_4 or a_5) is sent.
5. Either an acceptance (a_5) or rejection letter (a_4) is sent—never both.

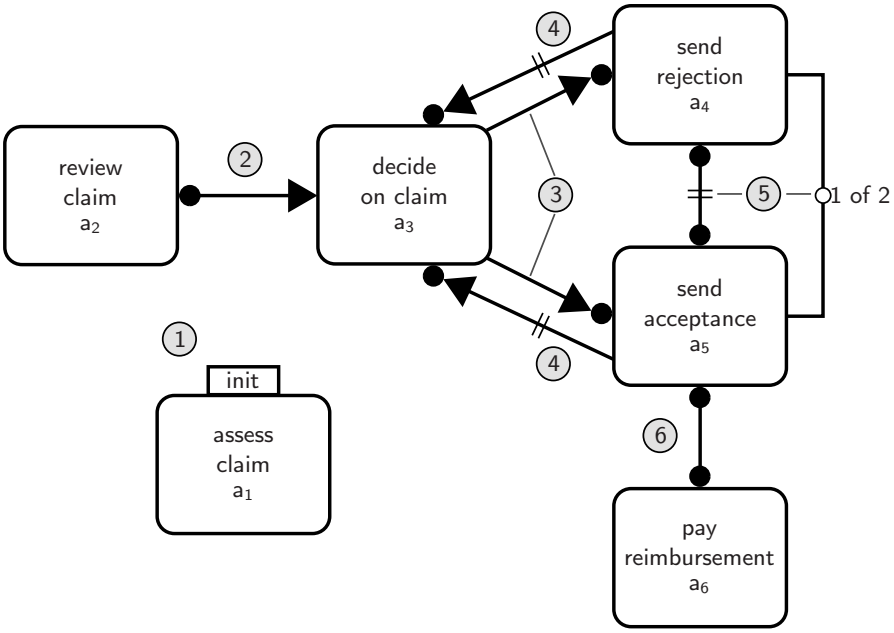


Figure 3.1: A declarative model of the insurance example using the DECLARE [43] language.

6. If an acceptance letter is sent (a_5), reimbursement must be paid (a_6) and vice versa.

The model leaves the precise ordering of activities and how often each activity is executed open. In the example, multiple reviews can be created before and after a decision. The latter, however, requires an additional decision that revises the outcome of the previous one. By defining only mandatory constraints, the declarative model is more concise than its control flow-based counterpart would be. However, it is also underspecified on purpose and allows undesired traces (e.g., multiple decisions following each other immediately).

DECLARE is not limited to the constraints used in our example. It supports both optional and mandatory rules [43]. Furthermore, various extensions to the core language have been proposed. Montali et al. [78] define metric temporal constraints, to express deadlines, delays, and more. In [77], Montali et al. extend DECLARE with data conditions: Constraints are mandatory if the corresponding data condition holds.

A common critique of declarative models is understandability [69]. Fahland et al. [55] propose that sequential aspects can be better understood in imperative models, while for circumstantial aspects declarative approaches are superior. Pichler et al. [69] investigate this claim empirically and find that imperative process models are generally easier to comprehend.[1] For DCR Graphs, tools have been developed to im-

[1]The authors of [69] point out that the participants of their study were more familiar with the imperative paradigm.

prove comprehensibility. Models are complemented with an equivalent textual description and an interactive simulator [134, 135].

Both DECLARE and DCR Graphs have limited support for data, which is central to knowledge-intensive processes. There exist process modeling approaches that are declarative and data-centric at the same time, though. The recently proposed Object-Centric Behavioral Constraints (OCBC) [118] is similar to DECLARE, but constraints are quantified via classes and associations. GSM [47] sorts activities into stages. Each stage has a data-based pre- and post-condition: If the precondition is satisfied, the contained activities can be executed. Thus, data-based conditions define declaratively how the process may be composed. We discuss such approaches in the next section.

Wickr, the contribution of this thesis, has declarative data dependencies and imperative control flow. However, declarative temporal rules, similar to DECLARE constraints, are used for verification of case models (see Chapter 8).

Data-Centric Process Modeling

Many processes are data-driven. Activities read, create, and update data, and their data-requirements constrain the process execution. Furthermore, processes may branch based on data. Thus, which variant of a process is executed depends highly on the available data.

It is important, to model this clearly. Therefore, data-centric process modeling languages emphasize the role of data during process execution. Data may be represented by business artifacts or objects, which have a structure and a behavior [47, 54, 147]. Data-centric process models are often declarative, as data conditions define whether an activity or state change can be triggered. In the remainder of this section, we present different data-centric process modeling approaches.

The case handling [29] approach addresses data-centric user-driven processes. A model consists of activities—often implemented by forms—and data objects. The execution order of activities is only constrained by data requirements. Furthermore, some data objects can be altered without executing an activity. At runtime, a query mechanism allows users to find relevant data and activities.

The GSM approach [60] is an artifact-centric process modeling language. Every model consists of an information model and a behavioral model. The information model describes the data structure. The behavioral model consists of stages with guards and milestones. Guards are attached to stages and define preconditions, i.e., a guard must evaluate to *true* to start a stage. Milestone are postconditions of stages, i.e., when the milestone has been reached, the stage can be left. Multiple stages can be active at the same time. When the artifact is in a certain stage, the aim is to satisfy a milestone.

PHILharmonicFlows [66] is an object-centric process modeling approach. A model consists of a set of classes, with a process model for each class. A class-specific process is a micro process, which describes

the state transition system of the respective objects. Furthermore, there is a macro process, which takes the interactions between different objects into account. One of the biggest strengths of PHILharmonicFlows is its holistic view on data-centric processes. It supports modeling data, behavior, as well as users' roles and access rights. Therefore, it can be used for various processes.

BAUML [86, 129] is an artifact-centric process modeling approach. Each process model consists of multiple UML models: A detailed class diagram defines the structure of artifacts. It supports associations, multiplicity constraints, and inheritance. State machines describe the lifecycles of artifacts, whose transitions can optionally be refined using activity diagrams. Furthermore, activities and additional constraints can be defined in the Object Constraint Language [93].

Proclets [23] are a Petri-net-based modeling language that supports artifact-centric processes. A model is split into multiple modules, each modeled by a workflow net. Through ports, different modules can communicate synchronously and asynchronously. Artifact-centric Proclets have one module for each artifact type. The module describes a type-specific lifecycle and may interact with others to realize the business process. Proclets inherit the formal precision of Petri nets.

Many data-centric modeling approaches modularize processes. Multiple instances of these modules come together at runtime to compose the process. State transitions of artifacts may need to be executed synchronously. However, synchronization may only occur between linked artifacts. Especially, many-to-many-associations (and synchronization) can be challenging. Such interactions, therefore, have been a research topic addressed by multiple works.

Fahland [141] presents a solution for Proclets. When instances of modules interact for the first time, so-called *synchronization sets* are initialized. Each module instance remembers instances of other modules it interacted with. This memory is used to synchronize future interactions. However, many-to-many synchronization requires global knowledge, e.g., the capability to query all synchronization sets to determine indirect relationships.

Similarly, relational process structures [133] describe the interactions between linked objects. They support both many-to-many interactions and multiplicity constraints. Furthermore, implicit transitive relations can be used for synchronization. At runtime, the graph of objects and their relations can be queried. The concepts have been applied to the PHILharmonicFlows language. Yet, [133] remains vague when and how relations are established on the instance level.

OCBC [118] is another object-centric process modeling approach. A model consists of a declarative activity-centric process model and a data model. The process model includes activities and temporal constraints similar to DECLARE. The data model has data classes and associations. Each association has two kinds of multiplicity constraints: Some always need to hold, and others need to hold eventually. The peculiarity of the approach is that activities and temporal constraints are quantified with regard to classes and associations. OCBC models have been successfully

discovered from event logs [120], but they have not yet been used to implement processes.

To keep the discussion brief, we discuss only some approaches. For a broader overview on data-centric process modeling, the DALEC framework [147] is a good starting point.

We provide a joint semantics for flexible process models and data models. Thereby, we respect declarative data-based constraints, i.e., multiplicity constraints. Similar to [118], we use two types of multiplicity constraints. Furthermore, we synchronize the processing of data objects based on links between the objects similar to [94, 133, 141]. Yet, our approach is not fully data-centric and rather shows the impact of data-centric constraints on activities.

Hybrid Process Modeling

Data-centric and activity-centric modeling have both their merits and demerits and so do imperative and declarative languages. Therefore, so call *hybrid approaches* combine ideas to get the best of both worlds [152]. We illustrate hybrid process modeling based on two representatives—CMMN [115] and fCM [112].

CMMN [115] is a case management standard by the Object Management Group (OMG). It is heavily influenced by GSM [60]. A case model consists of nested stages, which can contain activities. Each stage has entry and exit criteria, which are conditions that must be fulfilled to enter or exit a stage. As long as the process is in a certain stage, the activities in the stage can be executed—possibly repeatedly. An activity may refer to an atomic task or a process, e.g., a BPMN model. Additionally, rules may determine whether an activity or stage is mandatory and whether it must be repeated.

CMMN is a declarative approach combining ideas of activity-centric and data-centric process modeling. It includes data-centric stages as well as activities, and rules can be based on both available data and executed activities. The standard assumes an information model without specifying its nature or language [115, p. 39]. Based on the information model, data conditions and outputs of activities can be modeled, but this information is usually not part of the visual model.

In knowledge work, planning and ad-hoc adaptations to case-specific circumstances play an important role [31, 82]. CMMN supports planning with *discretionary tasks* [115, p. 6]. Each stage can be seen as a plan that can be adapted. Non-discretionary tasks must be included, while discretionary tasks may be included. Yet, CMMN does not consider adapting the model itself.

Adaptation is important to most case management use cases. Yet, not all use cases require the same degree of adaptability. Meyer et al. [90] distinguish between two types of case management: adaptive case management and production case management. Adaptive case management merges design and execution. Knowledge workers create

the process model on the go by continuously adapting it. In production case management, well-structured process fragments are created at design-time. At runtime, knowledge workers can compose these fragments and add new ones if necessary. In [90], the authors define a novel implementation framework for production case management. This framework is the first description of fCM.

Subsequently, the first formal definition of fCM is provided by Hewelt and Weske [112]. The paper introduces the fundamental concepts: domain models, object lifecycles, fragments, and termination conditions. Furthermore, the execution of cases and adding fragments at runtime is discussed. Yet, the presented semantics focuses on fragments, so some questions remain open: The authors state "data classes can be associated with each other" [112, p. 7] but do not discuss associations and corresponding multiplicities any further. However, already the example in [112] uses associations and multiplicity constraints, which clearly indicates the relevance of these concepts.

A first execution engine[2] for fCM is presented in [101]. It has been extended to accept external events [106] by integrating a complex event processing engine [96]. However, the execution engine supports neither data associations nor multiplicities.

Besides the implementation, conceptual extensions to fCM have been researched and proposed. Beck et al. [105] extend fCM to restore past states of data objects. Therefore, the state of a data object can be saved and later on restored. Andree et al. [153], introduce exception handling to fCM. The authors adapt exception handling patterns from traditional processes for case management. In general, exceptions may need to be handled on the level of the case, an activity, and/or a fragment.

One challenge that fCM faces is the complexity of case models with many fragments and inter-fragment dependencies. Therefore, Gonzalez-Lopez et al.[142, 163] propose landscape models for fCM. A landscape model is an abstract depiction of the dependencies among fragments.

In additional to the foundational and technical works on fCM, [131, 161] compare different methods for eliciting fCM models: starting with either the object lifecycles, the fragments, or the goals. In general, the fragment-first approach leads to less flexibility, while the goal-first and object-lifecycle-first approaches generally result in more flexible models, but conversely introduce additional complexity.

Our case management approach, *Wickr*, is based on fCM. We, for example, add support for associations, multiplicity constraints, and shared data objects. Thereby, the role of data is emphasized. Furthermore, we develop a translational semantics for case models, that targets Petri nets. The semantics combines both process and data-related aspects.

[2] `github.com/bptlab/chimera` (2021/11/06)

3.3 Formal Execution Semantics

Many BPM related activities require models with precise semantics. Yet, many languages, such as BPMN [92], have only textual, semiformal semantics, which leave room for interpretation. Consider BPMN's exclusive gateways: The subsequent control flow can be conditional. Ideally, the conditions are exclusive and complete, so that one and only one flow is chosen when the gateway is triggered. However, in BPMN conditions are evaluated in order until one evaluates to true [92, p 465]. Overlapping conditions are supported, but the evaluation order is not specified.

Formalizations resolve such inaccuracies. They define the semantics of a given model, sometimes tailored to a specific purpose. Hofstede et al. [14] present different kinds of semantics. *Translational semantics* map models of one language to another, usually a formal one. BPMN processes, for example, can be mapped to Petri nets [46]. *Operational semantics* are like interpreters. They specify the effects of modeled elements. For BPMN 2.0, Dijkman and Van Gorp [58, 74] define semantics based on graph rewriting rules, which operate directly on the process model. *Denotational semantics* assign a mathematical meaning to elements of the model. *Axiomatic semantics* go one step further: the model itself is treated as a logical theory which allows reasoning.

We specify the formal semantics of our case management approach by translating case models to Petri nets. The semantics adhere to constraints in the data model. In the remainder of this section, we discuss some examples of formalizations for business process models.

3.3.1 Formalization of Traditional Business Processes

Traditional business processes are highly structured and often expressed by control-flow-oriented process models. Their formal execution semantics are widely used to execute, verify, analyze, and simulate business processes. Dijkman et al. [46] present a translational semantics that maps BPMN models to classical Petri nets. Each activity is mapped to a single transition, which represents its termination while abstracting from the running state. Furthermore, there is no mapping for data objects and data flow. In [74], Van Gorp and Dijkman present a semantics based on in-place graph transformations of BPMN models. The transformation rules define token-based semantics for BPMN without translating the model into a different language (e.g., Petri nets). Van Gorp and Dijkman consider elements not supported in the Petri net mapping, such as inclusive gateways and multi-instance activities. Still, data objects and data flow are not considered.

Awad et al. [52] map data-aware BPMN models to Petri nets. Data object nodes are represented by places and data objects by tokens. Transitions that model activities can consume and produce tokens to represent reading and writing data objects. If an activity writes or reads

an object in one of several states, the activity is mapped to multiple alternative transitions, respectively.

The mentioned semantics do not consider data objects' attributes and links. However, in some cases, the internal structure of objects is used, e.g., in arc conditions [61]. Recently, such conditions and data-based rules are externalized to decision models [95], i.e., using Decision Model and Notation [169]. Batoulis et al. [119, 130] formalize these decision-aware process models using colored Petri nets. The semantics include data objects with abstract states and attributes as well as data-based conditions and rules that impact the process execution. Similarly, De Leoni et al. [168] map detailed decision and data-aware process models to *Data Petri nets* [70, 75].

Not all translational semantics target Petri nets. Process algebras, such as the π-calculus [19], can describe concurrent systems and provide methods for manipulating and analyzing them. Process algebras build complex processes from smaller ones using inter-process communication. As such, they are particularly useful for interorganizational processes [30]. BPMN models have been formalized using process algebras, such as Communication Sequential Processes [50], Calculus of Orchestration of Webservices [48], and π-calculus [97]. However, only [48] considers data and only in conditional control flow.

Process algebras can be used to modularize and compose processes, while classical Petri nets cannot. However, nowadays Petri nets are widely adopted by the BPM community, while process algebras are rarely used. This may be due to available tools [33]: While many tools exist for Petri nets, few support process algebras. Furthermore, representing Petri nets graphically eases human comprehension compared to the textual representation of process algebras. Finally, many extensions of Petri nets, such as hierarchical Petri nets [7], colored Petri nets [56], timed Petri nets [8], and the box calculus [9], have made the formalism more and more powerful and have resolved some disadvantages when compared to process algebras.

3.3.2 Formalization for Knowledge-Intensive Processes

Models and formalizations of knowledge-intensive process must capture flexible and data-driven behavior. In this section, we summarize works on formal semantics for knowledge-intensive and/or data-centric processes as well as respective formal languages.

Activity-centric declarative process modeling languages are based on temporal logics. The semantics of DECLARE [43] and DCR Graphs [59] are specified in LTL on finite traces [5, 68]. At any point, the enabled activities depend solely on the sequence of past activities.

This is insufficient for OCBC [118] which integrates data. OCBC's semantics is defined in [136] using temporal description logics [22]. The semantics support infinite time, because OCBC has no case notion.

Data-centric approaches constrain the behavior of a process based on fine-grained data-conditions. GSM's [60] semantics are based on event-condition-action rules. Analyzing and reasoning about such for-

malizations requires first-order logic and is, in general, undecidable [81]. However, it becomes decidable for state-bounded GSM models [81].

Data-aware processes including control flow have also been formalized as dynamic data-centric systems [114]. The process operates on a relational schema that includes additional relations to capture the control flow state. Montali and Calvanese [114] describe how processes can be encoded and how soundness can be verified. Verification against soundness is only decidable for state-bounded processes.

Another challenge is the presence of a database, whose contents are unknown during verification but influence the process execution [71]. Calvanese et al. [138, 155] use Satisfiability-Modulo-Theories to encode artifact-centric processes with read-only databases and verify safety properties independently of the database content.

For some data-centric approaches, translational semantics targeting Petri nets exist [140, 145]. Kang et al. [145] map synchronized object lifecycles to Petri nets with additional state transition rules. The traces of the Petri net can be filtered using the transition rules. The result represents the behavior of the synchronized object lifecycles. However, Petri nets may have infinitely many and infinitely long traces. Therefore, the approach is undecidable. Estanol et al. [140] present a Petri net mapping for BAUML models. The Petri nets uses inhibitor arcs, which causes common verification tasks, such as soundness checks, to become undecidable [64]. Furthermore, BAUML is not fully supported: Attribute-level information are missing, and multiplicity constraints can only be tested for 0 and 1. Another formalization of BAUML is presented by Calvanese et al. [84]. They show that termination of unrestricted BAUML models is undecidable, but it is decidable for a clearly defined subgroup. We discuss similar results for our approach.

Sporleder [117] presents a first mapping of fCM models to Petri nets, which is inspired by [46, 52]. The mapping uncovers some weaknesses of fCM, for example, once the precondition of a fragment has been satisfied, arbitrarily many fragment instances can be created. Holfter et al. [143, 144] use and adapt the Petri net mapping for model checking. Most notably, the number of concurrent fragment instances is limited by making a respective assumption. Furthermore, activities' begin and end are captured, which allows the concurrent execution of activities as long as they operate on different data objects.

Besides formalizations of higher-level languages, there exist low-level languages with precise semantics that are designed for flexible and/or data-centric processes. The already present Proclets [23] approach is a Petri-net-based language. However, some extensions, such as the mentioned many-to-many synchronization [141], cannot be mapped to classical Petri nets. Yet, the extension is mathematically defined.

DB-nets [122] can model processes that access relational databases. The database is persistent, and the schema may include mandatory primary and foreign key constraints. It is embedded into a colored Petri net via view places. A view place contains "tokens" that correspond to tuples in a database view. Each transition that is connected to a view place can delete, add, and consequently update tuples in the database.

The database must comply with the schema, but rollback arcs can be used to restore a compliant state if a transition caused a violation.

Catalog nets [156] also integrate relational databases and processes. In contrast to DB-nets, catalog nets only support read-only access to databases. In catalog nets, guards can query databases i) to check conditions and constraints, ii) to inject values from the database into the model, or iii) to inject new unique values (e.g., new primary keys). Limiting databases to read-only brings some advantages: Some properties can be verified independent of the database's contents.

In summary, formalizing and verifying data-aware processes is actively researched. However, verification is often undecidable, and only the state-bounded subclass can be verified [81, 84, 114].

In this thesis, we present *Wickr*, a case management approach combining data and process modeling. It supports associations, multiplicity constraints, and shared data objects. We present a formal semantics, which can be used for execution, verification, and planning, by translating models to classical and colored Petri nets. The formalism is based on fCM's Petri net mappings [117, 143, 144], but takes inspiration from other more data-centric approaches [114, 141, 155] as well.

3.4 Overview of the Most Influential Works

All the presented works are related to our research, yet some influenced us more than others. Table 3.1 provides an overview on the works that influenced us the most. The table lists the approaches; whether they focus on data, activities, or both; whether they are imperative, declarative, or combine both paradigms; and whether they are formal.

The table contains publications on high-level languages, such as the imperative activity-centric BPMN and the declarative data-centric OCBC, and publications on formalizations, such as BPMN's Petri net semantics [46], as well as works on formal analysis, such as [81, 84].

Table 3.1: Overview of important related work and their strongest influence on our research.

Approach	Activity-/ Data-Centric	Imperative/ Declarative	Formal Semantics?	Effect/Relation
fCM [112]	Hybrid	Hybrid	Partially [117, 143]	The starting point for our work
BPMN [92]	Activity	Imperative	No	Activities with explicit input and output sets
YAWL [28]	Imperative	Activity	Partially	Data controls the number of activity instances [27]
MERODE [94]	Data	Imperative	No	Existential associations and their effect on the order of object creation
Meyer et al. [76]	Activity	Imperative	No	Dependencies among data objects
Combi et al. [127]	Activity	Imperative	No	
Relational Process Structures [133]	Data	Declarative	No	Interactions among objects following their links
Proclets [141]	Either/Or	Imperative	Yes	Linking objects based on usage
PHILharmonicFlows [66]	Data	Imperative	No	Modularization of processes
BAUML [129]	Data	Imperative	Partially	
OCBC [118]	Data	Declarative	Yes [136]	Two types of multiplicity constraints
Dijkman et al. [46]	Activity	Imperative	Yes	Petri net formalization of activities and control flow
Awad et al. [52]	Activity	Imperative	Yes	Petri net formalization of data objects and data flow
Sporleder [117]	Hybrid	Hybrid	Yes	Petri net formalization of fCM
Holfter [143]	Hybrid	Hybrid	Yes	
DB-Nets [122]	Hybrid	Hybrid	Yes	Processes must adhere to integrity constraints of the data model
Solomakhin et al. [81]	Data	Declarative	Yes	Decidability of verification of data-centric processes
Montali and Calvanese [114]	Data	Imperative	Yes	
Calvanese et al. [84]	Data	Imperative	Yes	

4 Wickr: Improving fCM

Designed to model multi-variant data-driven processes concisely, fCM combines activity-centric and data-centric as well as imperative and declarative process modeling. Yet, there are limitations: i) Data associations and multiplicity constraints are not supported; ii) As a language, fCM is not orthogonal since there are multiple ways to model same behavior [14, 57]; iii) The formal semantics [117, 143] consider only fragments and the termination condition; iv) Every fCM model allows unwanted behavior (i.e., instantiating fragments arbitrarily often) [143].

Considering these limitations, we adapt fCM. We introduce associations and two sets of multiplicity constraints to domain models. We remove initial and final states from the object behavior, and we tidy up and extend fragments to reduce ambiguity and to make good use of links between data objects. Finally, we present a formal semantics that considers the full case model. We call the new approach *Wickr*.[1,2]

This chapter focuses on *Wickr's* syntax and is based on various publications [144, 157, 158, 160, 164–166]. Tim Sporleder's and Adrian Holfter's master theses [117, 143] reveal some shortcomings of fCM. Holfter's thesis was the foundation for two joint publications [144, 165]. Later works [157, 166] were written together with Marco Montali.

4.1 Domain Model

Data is essential to knowledge-intensive processes [98]. While data objects are containers for information, relationships among objects are also important: If we, for example, want to assess the performance of a reviewer, we need to consider all the claims they reviewed and all the reviews they created. Yet, fCM does not acknowledge the role of associations [112]. *Wickr's* domain models and semantics, on the other hand, include associations and multiplicity constraints.

Yet, there is a conceptual mismatch between structural data models and process models. Data models do not describe how the data changes over time, while process models define behavior, i.e., how the processes transition from one state to another. This includes changes to data. When combined, the process either has to adhere to the data model continuously, or the process is treated as a transaction: Its data must comply to the data model after a process instance has terminated.

[1] In previous publications, we call our approach fCM despite the changes. To prevent confusion, we refer to our extension of fCM as *Wickr* and to the work of Hewelt and Weske [112] as fCM.

[2] Wicker is a technique for weaving twigs or branches to create objects, i.e., furniture. In our approach, a single fragment is inflexible, but by *interweaving* multiple fragments at runtime, many variants can be realized—hence, the name *Wickr*.

For data structures, *Wickr* accounts for both their intermediate states, which may change, and goal states, that ought to be reached eventually. For this purpose, domain models have two types of multiplicity constraints: *Global multiplicity constraints* must never be violated. *Goal multiplicity constraints* can be violated during execution but must be fulfilled when the case is closed. In *Wickr*, objects and links cannot be deleted. Therefore, goal multiplicity constraints may be more restrictive than global multiplicity constraints but not vice versa. In the model, we precede goal multiplicity constraints with $\diamond$.

To avoid ambiguity during execution, we furthermore constrain *Wickr's* domain models:

1. All associations are binary and existential (at least one of the lower bounds is positive).
2. Every two classes are connected by at most one association.
3. Associations multiplicity constraints allow 1-to-1 or 1-to-many relationships but not many-to-many.

These constraints are necessary to determine when two objects get linked [94]. We elaborate this reasoning in context of *Wickr's* execution semantics in Chapter 6. A domain model that satisfies all these constraints is called *well-formed*.

As a consequence of these constraints, we can employ a simplified definition of domain models for *Wickr* (Definition 20). The multiplicity constraints are defined by three functions: global lower bound, global upper bound, and goal lower bound. There is no explicit goal upper bound because links cannot be deleted, so that goal upper bounds always equal their global counterpart. From the multiplicity constraints, we can infer the associations among classes. As a result, roles and explicit associations become obsolete (cf. Definition 1).

Definition 20 (Wickr Domain Model)**.** A *Wickr* domain model d is a tuple $d = (C, c_{co}, \ell, u, \diamond\ell)$, where

1. C is a finite non-empty set of classes.
2. $c_{co} \in C$ is the dedicated case class.
3. Associations and multiplicity constraints are defined by three functions:
   a) $\ell : (C \times C) \to \mathbb{N}_0$ is a function that defines the lower bounds of the global multiplicity constraints.
   b) $u : (C \times C) \to (\mathbb{N}_0 \cup \{*\})$ is a function that defines the upper bounds of global and goal multiplicity constraints. It holds that $*$ is greater than any number: $\forall n \in \mathbb{N}_0 : n < *$.
   c) $\diamond\ell : (C \times C) \to \mathbb{N}_0$ is a function that defines the lower bounds of goal multiplicity constraints.

Two classes $c_1, c_2 \in C$ are associated iff $u(c_1, c_2) > 0$ and $u(c_2, c_1) > 0$. Furthermore, lower bounds must not be bigger than upper bounds:

$$\ell(c_1, c_2) \leq \diamond\ell(c_1, c_2) \leq u(c_1, c_2)$$

If there is no association, all multiplicity constraints must equal zero:

$$(u(c_1, c_2) = 0) \Leftrightarrow (u(c_2, c_1) = 0)$$

This definition only supports binary associations and at most one association between two classes. Nevertheless, *well-formed* domain models must also satisfy the following properties:

1. Associations are existential.

$$\forall c_1, c_2 \in C : u(c_1, c_2) > 0 \Rightarrow (\ell(c_1, c_2) \geq 1 \vee \ell(c_2, c_1) \geq 1)$$

2. There are no many-to-many associations.

$$\forall c_1, c_2 \in C : u(c_1, c_2) > 1 \Rightarrow u(c_2, c_1) = 1$$

◇

While *Wickr* domain models are more restricted than data models (Definition 1), they are equally expressive: Any data model can be transformed into a well-formed *Wickr* domain model through reification [42, pp. 123]. Reification replaces associations with a novel class and new associations. A many-to-many association is reified into a class and two 1-to-many associations. A non-existential association is reified into a class and two existential associations. In some cases, such as reflexive associations, a single association needs to be reified into multiple classes—in other words, reification must be repeated.

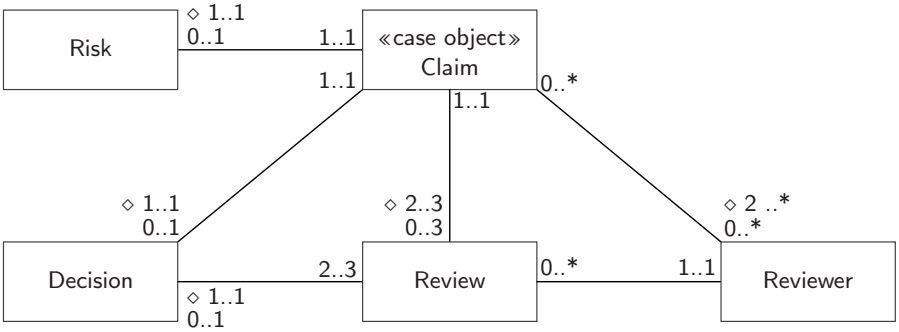


Figure 4.1: Not well-formed *Wickr* domain model for the insurance example. Goal multiplicity constraints that do not refine the corresponding global multiplicity constraint are omitted.

Figure 4.1 depicts the domain model for insurance claims. The case object is the claim. During the case, the risk is assessed, reviewers are assigned, reviews are created, and a decision is made. The goal multiplicities require that every claim eventually has at least two reviewers

and two reviews as well as exactly one risk and one decision. Also, every review must eventually be associated to one decision. However, the domain model is not well-formed. One reviewer may be assigned to different claims, and one claim can have multiple reviewers. This is modeled with a non-existential many-to-many association.

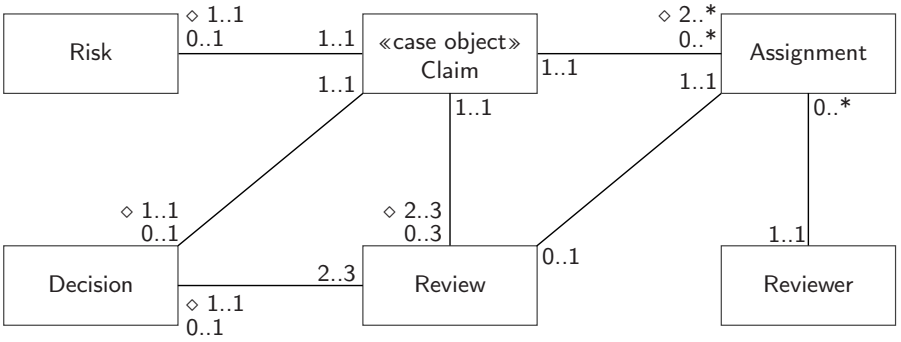


Figure 4.2: Well-formed domain model of the insurance example. The model was derived from Figure 4.1 through reification.

We can reify the association between claim and reviewer to make the domain model well-formed. Therefore, we introduce a class *Assignment* between *Claim* and *Reviewer*. Each assignment is linked to one reviewer and one claim. Furthermore, we *redirect* the association between reviewer and review to lead from review to assignment. Figure 4.2 depicts the result, and Table 4.1 specifies the functions ℓ, u, and $\diamond\ell$ fully.

Table 4.1: Function ℓ,$\diamond\ell$, and u for the domain model in Figure 4.2.

c_1	c_2	$\ell(c_1, c_2)$	$\diamond\ell(c_1, c_2)$	$u(c_1, c_2)$
Claim	Risk	1	1	1
Claim	Assignment	1	1	1
Claim	Review	1	1	1
Claim	Decision	1	1	1
Risk	Claim	0	1	1
Assignment	Claim	0	2	*
Review	Claim	0	2	3
Decision	Claim	0	1	1
Decision	Review	0	1	1
Review	Decision	2	2	3
Review	Assignment	0	1	1
Assignment	Review	1	1	1
Assignment	Reviewer	0	1	*
Reviewer	Assignment	1	1	1
all other combination		0	0	0

The example shows reification requires domain knowledge. In general, a domain expert has to provide meaningful names for newly added classes. However, if we have a data model with named associations and roles, some new class names may be derived automatically.

On another note, the term *domain model* often refers to a data model that comprises all relevant concepts and relationships of the domain

(i.e., an enterprise) and not only those relevant to one case model. While not considered here, *Wickr* can be adapted accordingly: Given such an organizational domain model, case objects and the goal multiplicity constraints must still be defined for each case model. Furthermore, goal multiplicity constraints may define upper bounds that overwrite the global ones for respective cases.

4.2 Object Behavior

The object behavior describes how the state of an object may change. In fCM, object behaviors are modeled by class-specific lifecycles. However, real-world processes may create an object in one of several states to encode a decision outcome [169], and processes may cover object lifecycles only partially [112], e.g., when objects are passed from one process instance to another. Therefore, initial and final states may be irrelevant to a case. Furthermore, objects' final states are ignored by fCM's execution semantics [117, 143].

For these reasons, we decided to omit initial and final states by using simple state transitions systems (see Definition 2). An object behavior may even include disconnected parts if the object can be created in different states. The states in which a data object is created, is solely determined by the I/O-behavior of activities. This eases the integration of object behaviors and process fragments.

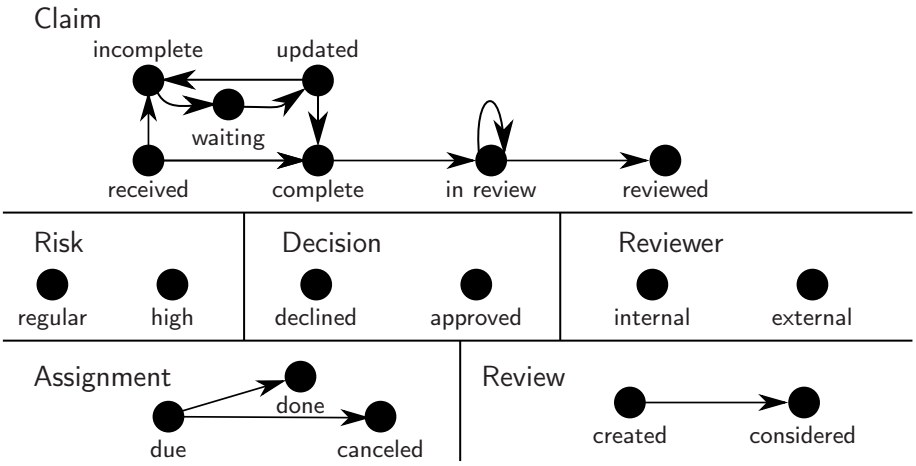


Figure 4.3: State transition systems describing the object behavior of the insurance example (see classes in Figure 4.2).

Figure 4.3 depicts the state transition systems for the insurance example. The state of class *Risk* represents the knowledge worker's assessment. It is *regular* or *high*; hence, the state transition system has two disconnected states. Similarly, a *decision* is either in state *declined* or *approved*.

4.3 Fragments

In both fCM and *Wickr*, process fragments are the building blocks for the case behavior. If all fragments consist of only one activity, the case is fully data-driven. If all activities are included in a single fragment, the case is activity-centric. Thus, the degree of flexibility depends on the design of fragments.

In *Wickr*, we redefine fragments to resolve design flaws in fCM:

- To limit the number of fragment-instances, fragments' explicit preconditions are removed.
- To clearly identify the entry point for a new process, initial fragments are marked.
- To guarantee that exactly one case object exists, it must be created during case instantiation, i.e., by a start event.
- To clearly separate the termination of a fragment from the goal of a case, end events are removed.
- To improve orthogonality, parallel gateways are not supported.
- Furthermore, data integration is improved:
  - Activities have explicit input and output sets.
  - Activities can process data objects in batches.
  - Data objects can be shared among cases.

Fragments Without Explicit Precondition. In fCM, each fragment has a data-based precondition modeled as a conditional start event. A new fragment can be instantiated if its precondition is satisfied [112]. Yet, BPMN's conditional start events have a different semantics. In BPMN, the conditional event is triggered *once* whenever its evaluation changes from false to true [92, p. 342]. On the other hand, fCM allows instantiating a fragment arbitrarily often if its precondition holds [117, 143]. Both semantics are problematic as BPMN cannot handle multiple data objects of the same class, and fCM's fragments have livelocks.

When the first activity is executed, it may invalidate its own precondition and prevent further instances. Therefore, we want to prevent that a fragment can be instantiated without executing its first activity. Therefore, *Wickr* fragments have no explicit precondition. Instead, the precondition is merged with the input sets of the fragment's first activity. In *Wickr*, instantiating the fragment is equal to starting this activity.

Figure 4.4 depicts both the fCM and the *Wickr* version of a process fragment for internal reviews. The fCM fragment has a precondition, which requires a claim object in state *complete* and a risk object in state *regular*. The *Wickr* version does not have a dedicated precondition, instead the fragment's first activity—"assign internal review"—reads both a claim in state *complete* and a risk object in state *regular*.

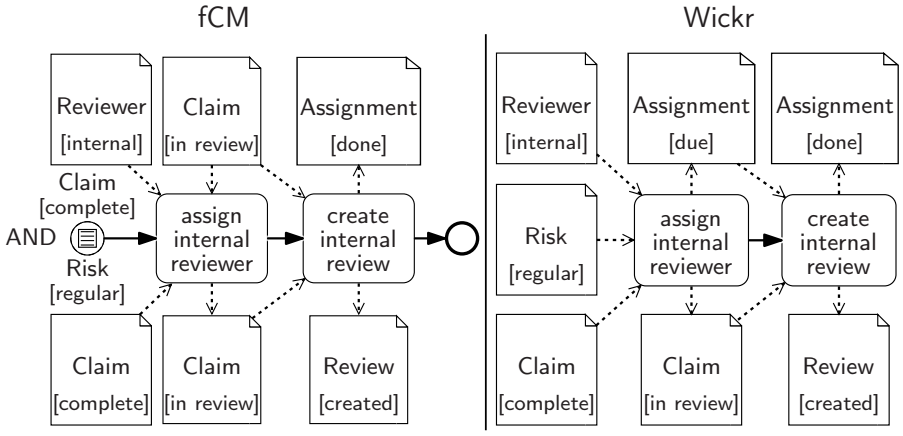


Figure 4.4: Fragments in fCM have a precondition, depicted by a conditional start event. In *Wickr*, the precondition is merged with the data-requirements of the first activity.

Initial Fragments. In fCM, case models have no dedicated beginning: A case can be started manually, and any fragment whose precondition is satisfied in the initial state can be instantiated. *Wickr* supports explicit initial fragments that begin a new case and are executed at most once. Opposed to all other fragments, initial ones begin with a start event.

Explicit entry points have an additional advantage. Start events can have output sets. When a new case begins, one of the output sets is selected, and the case is populated with the respective objects. This mechanism replaces fCM's set of initially instantiated classes (cf. Definition 8, p. 8). However, *Wickr's* solution is more expressive: Alternative start events can be modeled—each has a set of alternative output sets. Also, we require that the output sets include at least the case object, as it is the subject of the case, and a case without a subject cannot exist.

The models in Figure 4.5 define the fCM and *Wickr* version of the insurance example's initial fragment. The fCM fragment has a conditional start event. Without looking at other parts of the case model, it is impossible to tell if it is initially enabled. The *Wickr* fragment has a start event that creates a claim object. Since it is the only fragment with a start event, it clearly marks the start for every new case.

Fragments Without End Events. End events in fCM denote the end of a fragment. This is not necessarily a meaningful milestone, nor does it fit BPMN's semantics, where an end event denotes the completion of a process instance (i.e., a case) [92, p. 443]. Therefore, *Wickr* does not contain end events (see Figure 4.4 for an example). Instead, the goal of a case is specified using the termination condition (similar to fCM) and goal multiplicity constraints.

Data Integration. Within a fragment, data is modeled by data object nodes and data flow. An activity's incoming data flow represents that

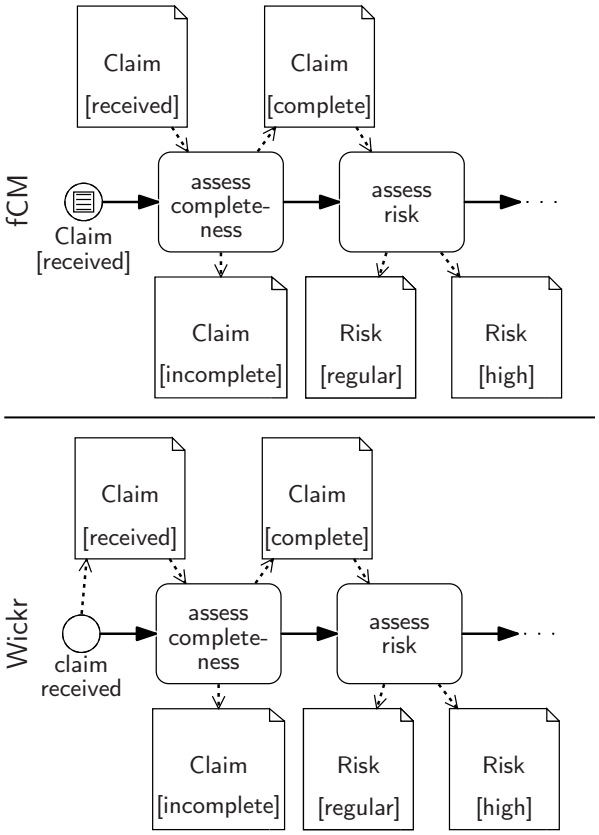


Figure 4.5: *Wickr* uses start events to model the possible beginnings of cases. Start events can introduce data to the case. Here, "claim received" creates a claim object in state *received*.

data objects are read. Outgoing data flow models that data objects are written. In fCM, input and output sets as well as combinations thereof are defined implicitly because an activity can access at most one object for each class, and activities that create or update objects must comply to the object lifecycles. Optional inputs or reading multiple objects of the same class are not supported. Furthermore, fCM assumes that each data object can be accessed by only one case.

Wickr defines input and output sets explicitly. It supports optional inputs, batch processing through *set data object nodes*, and data that is shared among cases. However, input-output-set combinations remain implicit. Valid combinations are determined by the object behavior and associations between classes. We detail these rules in Section 4.5.

Cases commonly involve multiple objects of *one* class. These objects are created one after another and can be updated independently. However, *Wickr* also supports batch processing: Multiple objects of the same class can be updated by executing an activity once. Therefore, data object nodes have a set indicator. Visually, data object nodes for sets include the icon "|||". In BPMN, the same icon represents a *collection* of data objects [92, p. 206]. An example is reviews for a claim. When the claim is approved or declined, all its reviews should be considered.

Finally, data objects in *Wickr* may be shared among cases. We call these objects *cross-case data objects*. They can exist before a case begins and after a case terminates. Furthermore, multiple cases can access them currently. Data object nodes have a boolean indicator that determines whether objects are shared. Nodes for cross-case data objects are visualized as data stores (cf. [92, p. 207]).

Figure 4.6 shows two *Wickr* fragments. Each has one activity. The first one assigns external reviewers to claims. Reviewers are not exclusive to one claim; hence, we model them as cross-case data objects. Also, more than one reviewer may be assigned: When the first reviewer is assigned, the claim changes from *complete* to *in review*. The second time, the claim is read in state *in review*. Activity "request external review" has two respective input sets, which are marked by □ and ○.

The second fragment captures a decision of the knowledge workers. The activity's input set consists of the claim, an external reviewer, and all the claim's reviews. Considering all the reviews for one claim, the knowledge workers may choose one of three outcomes:

Output set ■: The claim can be declined.

Output set •: The claim can be approved.

Output set ▲: No decision is created, but an additional external reviewer is assigned to the claim.

When the claim is declined or approved, all its reviews are changed to state *considered*. But when an additional review is required, an assignment is created, and no other objects are written.

Data objects are also used to evaluate conditional control flow. Such a flow must start in a gateway. Since *Wickr* captures details about data

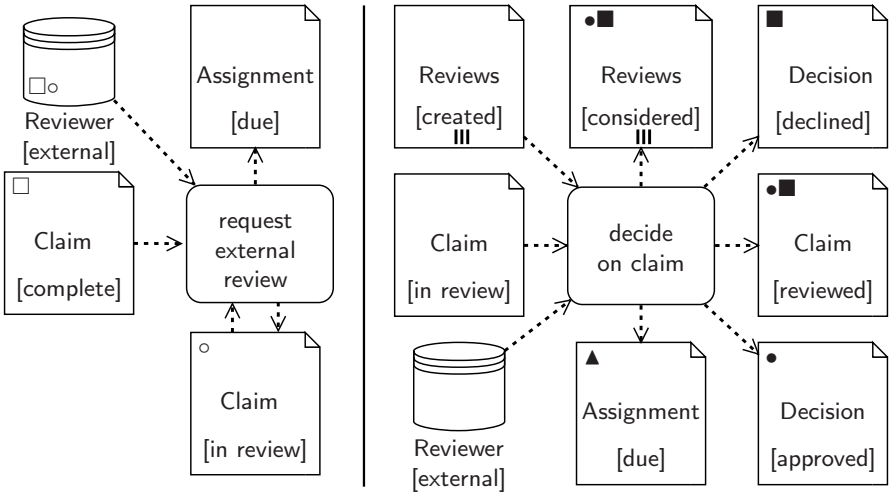


Figure 4.6: Two fragments demonstrating *Wickr's* explicit input and output sets, cross-case data objects, and batch processing. Activity "request external review" has two overlapping input sets (marked by □ and ○) which include a cross-case data object. Activity "decide on claim" reads a set of reviews and writes one of three output sets (marked by ■, ●, and ▲).

that are absent in fCM and data-aware process models, we revise the definition of a data condition (Definition 21). We model conditions using disjunctive normal form. As propositional variables, we use data object nodes.

Definition 21 (Wickr Data Object Node, Data Condition)**.** A data object node is a tuple $(c, q, \mathit{isSet}, \mathit{isShared}) \in DO$, where (c, q) is a phase (Definition 4), and $\mathit{isSet}, \mathit{isShared} \in \{\mathit{true}, \mathit{false}\}$ indicate whether the node represents a set and a cross-case data object, respectively.

Let DO be a set of data object nodes, where each data object node comprises a class, a state, a set indicator, and a cross-case indicator. A *Wickr* data condition $\mathcal{CON}$ is a set of sets of data object nodes:

$$\mathcal{CON} \subseteq \mathcal{P}(DO)$$

◇

The semantics of data conditions depends on the associations between classes and are elaborated in Chapter 6. Here, we limit the discussion to a few examples: The condition "a (local) claim must be in state reviewed" is formally expressed by

$$\{\{(\mathit{Claim}, \mathit{reviewed}, \mathit{false}, \mathit{false})\}\}$$

where the first element is a class, the second a state, the third a set indicator, and the last one a cross-case indicator. To express "all the (local) assignments of a (local) reviewed claim must be in state done," we use the following condition:

$$\{\{(\mathit{Claim}, \mathit{reviewed}, \mathit{false}, \mathit{false}), (\mathit{Assignment}, \mathit{done}, \mathit{true}, \mathit{false})\}\}$$

Wickr fragments are acyclic control flow graphs with i) an optional start event, and ii) data object nodes with a set indicator and an indicator for cross-cases data objects. Inputs and outputs of activities are grouped into alternative input and output sets, respectively. Fragments are formally defined in Definition 22.

Definition 22 (Wickr Fragment)**.** A *Wickr* process fragment is a tuple $f = (s, N_A, N_\times, \xrightarrow{N}, DO, i, o, con)$, where

1. $N = \{s | s \neq \bot\} \uplus N_A \uplus N_\times$ is a set of control flow nodes, where
   - s is a start event or $\bot$ if the fragment has none.
   - N_A is a finite non-empty set of activities.
   - $N_\times$ is a finite set of exclusive gateways.
2. $\xrightarrow{N} \subseteq (N \times (N \setminus \{s\}))$ is an acyclic control flow relation.
3. DO is a set of *Wickr* data object nodes.
4. $i : N_A \to \mathcal{P}(\mathcal{P}(DO))$ assigns each activity a non-empty finite set of potentially empty input sets that contain data object nodes.
5. $o : (N \setminus N_\times) \to \mathcal{P}(\mathcal{P}(DO))$ assigns activities and start events a non-empty finite set of potentially empty output sets, where each output set contains data object nodes.
6. $con : \xrightarrow{N} \to \mathcal{P}(\mathcal{P}(DO))$ assigns each control flow a *Wickr* data condition. Only control flow starting in an exclusive gateway may be conditional:

$$\forall n_1 \xrightarrow{N} n_2 : n_1 \notin N_\times \Rightarrow con(n_1, n_2) = \{\emptyset\}$$

◇

We define additional rules for well-formed fragments (see Definition 23) to prevent some structural errors.

Definition 23 (Well-Formed Wickr Fragment)**.** A *Wickr* fragment $f = (s, N_A, N_\times, \xrightarrow{N}, DO, i, o, con)$ is well-formed, if

1. Fragments start with a single activity or a start event.

$$\exists! n \in (N \setminus N_\times), \forall n' \in N : (n', n) \notin \xrightarrow{N}$$

2. The control flow graph is connected.

$$\exists! n \in (N \setminus N_\times), \forall n' \in N : n \neq n' \Rightarrow (n, n') \in \xrightarrow{N^*}$$

   where $\xrightarrow{N^*}$ is the transitive closure of $\xrightarrow{N}$.

3. There is no uncontrolled flow, i.e., only gateways may have more than one incoming or more than one outgoing control flow.

$$\forall n \in (N \setminus N_\times) : |\{n' \in N | n' \xrightarrow{N} n\}| \leq 1 \wedge |\{n' \in N | n \xrightarrow{N} n'\}| \leq 1$$

4. Gateways are either merges or splits.

$$\forall g \in N_\times : (|\{n \in N | n \xrightarrow{N} g\}| = 1 \Rightarrow |\{n \in N | g \xrightarrow{N} n\}| > 1) \wedge$$
$$(|\{n \in N | g \xrightarrow{N} n\}| = 1 \Rightarrow |\{n \in N | n \xrightarrow{N} g\}| > 1)$$

5. Input sets include at most one data object node for each class.

$$\forall n \in N_A, \forall R \in i(n), \forall do, do' \in R : do.c = do'.c \Rightarrow do = do'$$

6. Arc conditions' product terms refer at most once to a class.

$$\forall (n, n') \in \xrightarrow{N}, \forall \Theta \in con(n, n'), \forall do, do' \in \Theta :$$
$$do.c = do'.c \Rightarrow do = do'$$

7. For each control flow node, each output set has at most two data object nodes of the same class. If there are two, the set indicators must be different.

$$\forall W \in o(N \setminus N_\times), \forall do, do' \in W :$$
$$(do \neq do' \wedge do.c = do'.c) \Rightarrow do.isSet \neq do'.isSet$$

8. The output sets of start events must not contain set objects.

$$s \neq \perp \Rightarrow \forall W \in o(s), \forall do \in W : do.isSet = false$$

9. If an activity has an output set containing a set data object node, it must have an input set with a corresponding element.

$$\forall a \in N_A, \forall W \in o(a), \forall do' \in W, \exists R \in i(a), \exists do \in R :$$
$$do'.isSet = true \Rightarrow (do.isSet = true \wedge do.c = do'.c)$$

◇

Figure 4.7 depicts all *Wickr* fragments of the insurance example. Here, we cover each one briefly, while the next chapter provides a more detailed explanation. The first fragment is the initial one. It is triggered when the claim is received, and it completes by rejecting the claim or paying the reimbursement. We use undirected associations between a control flow and data object nodes to model conditional control flow. While executing the first fragment, other fragments may be necessary. The second fragment is executed once or multiple times sequentially if the claim is incomplete. The third fragment assigns an external reviewer; the fourth receives an external review; and the fifth cancels a requested review that has not yet been received. The sixth fragment is responsible for internal reviews. The final fragment includes the mentioned decision: A claim can be declined or approved, or an additional external reviewer may be assigned. All fragments are well-formed.

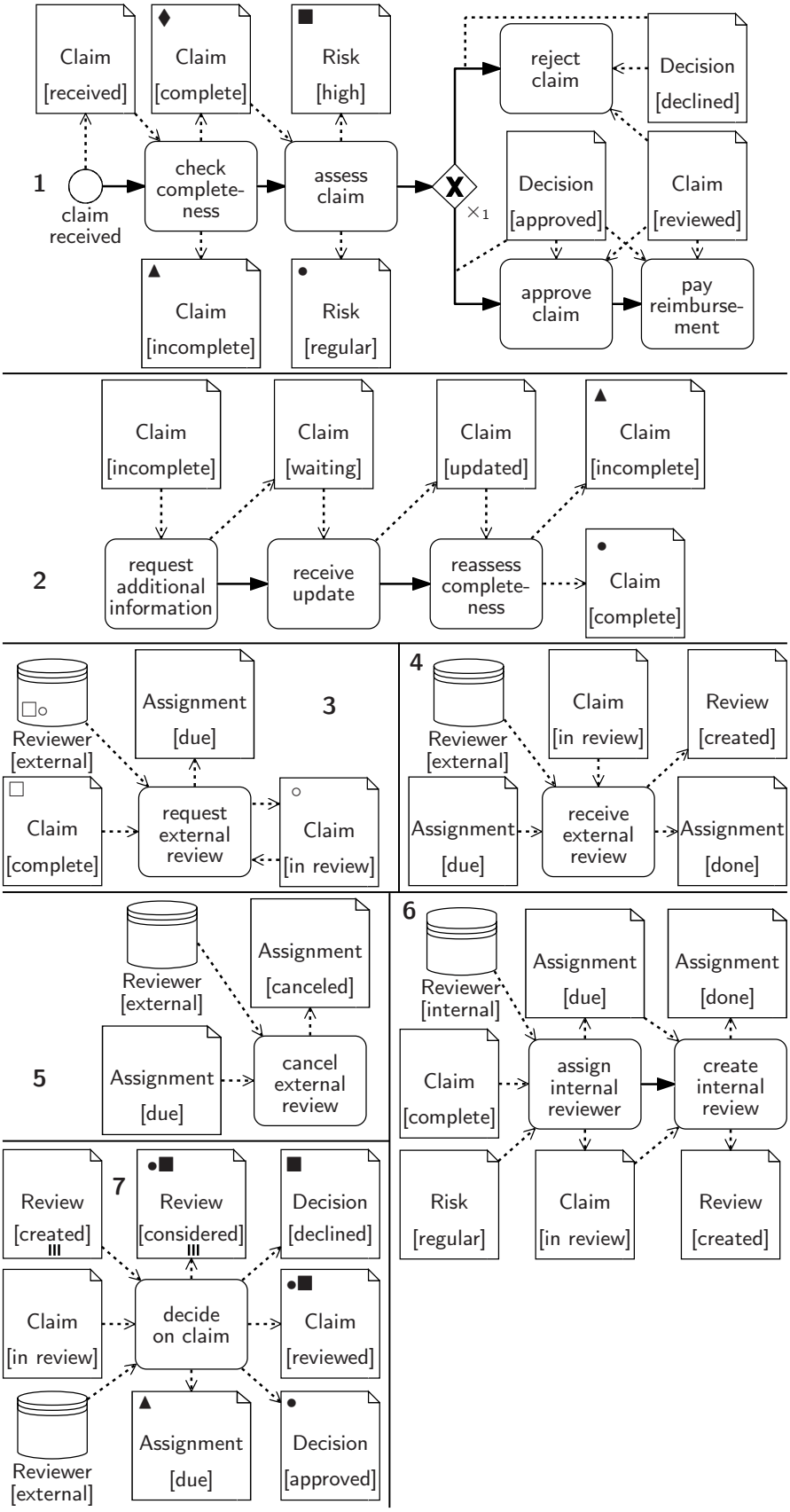


Figure 4.7: Wickr fragments for the insurance example.

4.4 Goal Specification

Wickr defines the case goal by the goal multiplicity constraints and the termination condition. If a case satisfies both, it can be closed by knowledge workers. The termination condition is a data condition.

In the insurance example, eventually the claim must be reviewed, and there must be a linked decision in state approved or declined. We can express this formally by the data condition:

$$\{\{(\mathit{Claim}, \mathit{reviewed}, \mathit{false}, \mathit{false}), (\mathit{Decision}, \mathit{approved}, \mathit{false}, \mathit{false})\},$$
$$\{(\mathit{Claim}, \mathit{reviewed}, \mathit{false}, \mathit{false}), (\mathit{Decision}, \mathit{declined}, \mathit{false}, \mathit{false})\}\}$$

4.5 Case Model

Wickr's case models (Definition 24) consist of a domain model, an object behavior for each class, a set of fragments, and a termination condition. *Wickr* and fCM differ in the definitions of their parts. Furthermore, unlike fCM, *Wickr* does not define a set of classes that gets instantiated with every new case. Instead, start events can have outputs.

Definition 24 (Wickr Case Model)**.** A *Wickr* case model *wickr* is a tuple $\mathit{wickr} = (d, B, \mathfrak{b}, F, \mathcal{CON}_{term})$, where

1. $d = (C, c_{co}, \mathfrak{l}, u, \diamond\mathfrak{l})$ is a *Wickr* domain model.
2. B is a set of state transition systems.
3. $\mathfrak{b} : C \to B$ is a function assigning each class c its behavior.
4. F is a set of fragments, where each data object node is defined based on the classes in $d.C$ and their states according to $\mathfrak{b}$.
5. $\mathcal{CON}_{term}$ is the termination condition.

$\diamond$

The parts of a case model depend on one another: For each class exists a behavior; each data object node references a class and a state; and the termination condition uses data object nodes. In the following, we present criteria that all case models must satisfy.

4.5.1 Structural Satisfiability

Activities and data conditions express data requirements using data object nodes. In *Wickr*, objects are created, linked, and updated by activities and events contained in fragments. Furthermore, a single fragment does not necessarily have a continuous data flow. Yet, despite the fragmented data flow, all data requirements must be satisfiable. A data object node is *structurally satisfiable* (see Definition 25) if the node is cross-case,[3] or if an output set contains a local data object node with the same class and the same state.

[3] We assume an open world, in which a cross-case data object node is always satisfiable because it may be in an output set in another (unknown) case model.

Definition 25 (Structurally Satisfiable Case Model.)**.** Given a *Wickr* case model $wickr = (d, B, \mathfrak{b}, F, \mathcal{CON}_{term})$, let DO_R be the set of data object nodes contained in input sets and data conditions. Furthermore, let DO_W be the set of data object nodes contained in output sets. The model *wickr* is structurally satisfiable, if

$$\forall do \in DO_R, do.isShared = false, \exists do' \in DO_W :$$
$$do'.isShared = false \wedge do.c = do'.c \wedge do.q = do'.q$$

◇

4.5.2 Object Behavior Conformance

Activities read, write, and create data objects. When the state of a data object is changed, the update must conform to the corresponding object behavior. This means every state transition performed by an activity must be modeled in the corresponding object behavior. However, an activity may have multiple input sets and multiple output sets, and not all possible input-output-set combinations need to be valid.

To check whether an input-output-set combination is *object behavior conform*, we determine all data objects that get updated. Therefore, we search for a node in the input set and a node in the output set which have the same class and the same set indicator.

Definition 26 (Object Behavior Conformance)**.** Given a case model $wickr = (d, B, \mathfrak{b}, F, \mathcal{CON}_{term})$, let R be an input set and W be an output set of an activity. The input-output set combination (R, W) is object behavior conform if every update complies to the respective object behavior:

$$\forall do \in R, do' \in W :$$
$$(do.c = do'.c \wedge do.isSet = do'.isSet) \Rightarrow ((do.q, do'.q) \in \mathfrak{b}(do.c).\delta)$$

◇

4.5.3 Contextual Object Creation

The I/O-behavior of activities clearly describes when data objects are created. Due to the existential associations in the domain model, it also describes when data objects are linked. An existential association describes that objects of one class depend on objects of another class. We call the objects *dependents* and *supporters*, respectively.[4] When a dependent is created, it is linked to all its supporters.

Therefore, when an input-output-set combination implies creating a new object, the object's supporters must either be read or co-created. Furthermore, a dependent may require multiple supporters of the same class if the respective global multiplicity has a lower bound greater than one. In this case, a set of corresponding data objects must be read.

[4] This is similar to MERODE's use of existential association [94], but the naming is different: In MERODE *supporters* are called *masters*, and *dependents* are called *slaves*.

Definition 27 (Contextual Object Creation.)**.** Given a *Wickr* case model $wickr = (d, B, \mathfrak{b}, F, \mathcal{CON}_{term})$ and an input set R and an output set W. The input-output set combination (R, W) satisfies contextual object creation if for each object that is created all supporters are accessed.

We define a set W_{new} of data object nodes that are instantiated:

$$W_{new} = \{do \in W | \{do' \in R | do'.c = do.c \wedge do'.isSet = do.isSet\} = \emptyset\}$$

Furthermore, we define a function *sup* that given a class returns the set of supporting classes:

$$\forall c \in d.C : sup(c) = \{c_s \in d.C | d.l(c_s, c) > 0\}$$

Then (R, W) satisfies contextual object creation if

$$\forall do_n \in W_{new}, \forall c_s \in sup(do_n.c), \exists\, do_s \in (R \cup W_{new}) : \\ do_s.c = c_s \wedge ((d.l(c_s, do_n.c) > 1) \Rightarrow do_s.isSet)$$

◇

4.5.4 Contextual Batch Processing

If an input set or condition contains a set data object node, multiple objects may be accessed. At runtime, it must be clear which objects are contained in the set. Therefore, the input set of the activity or the product term of the condition must contain a *reference object*. The set consists of all objects that belong to the class specified by the respective node and that are linked to the reference object.

Consider an input set that requires a set of reviews. The input set must also contain a data object node for a suited reference object. If the reference object is a claim, all reviews of that claim need to be read. If it is a reviewer, all reviews of the reviewer are read.

Contextual batch processing requires that whenever a set of objects is accessed, a *single* suited reference object must be read as well:

- The reference object must not be read as a part of a set.
- The reference object's class must be associated to the set's class.
- The multiplicity constraints must allow that multiple objects of the set's class can be linked to one reference object (i.e., it is a 1-to-many association).

A data object node for the reference object must be included in the input set, but its special role is not explicitly modeled.

Definition 28 (Contextual Batch Processing.)**.** Given a case model $wickr = (d, B, \mathfrak{b}, F, \mathcal{CON}_{term})$, let R be an input set of an activity or the product term of a condition. The set R satisfies contextual batch processing if for each set data object node in the input set, exists exactly one reference object.

$$\forall do_s \in R, do_s.isSet, \exists! do_r \in R, \neg do_r.isSet : d.u(do_s.c, do_r.c) > 1$$

◇

4.5.5 Well-Formed Case Model

A well-formed case model (Definition 29) consists of well-formed parts. Furthermore, it satisfies *Structural Satisfiability*, and for each input set must exist an output set and vice versa so that the combination satisfies *Object Behavior Conformance*, *Contextual Object Creation*, and does not create an instance of the case object. Also, each input set must satisfy *Contextual Batch Processing*. While these criteria do not guarantee correct behavior (i.e., that the case goal can be reached), they prevent some structural errors.

Definition 29 (Well-Formed Wickr Case Model)**.** Given a case model $wickr = (d, B, \mathfrak{b}, F, \mathcal{CON}_{term})$, let $N_{A;F} = \bigcup_{f \in F} f.N_A$ be the set of all activities, $i_F = \bigcup_{f \in F} f.i$ the function assigning each activity a set of input sets, and $o_F = \bigcup_{f \in F} f.o$ the function assigning each activity and start event a set of output sets. The case model is well-formed, if

1. The domain model d is well-formed.
2. All fragments $f \in F$ are well-formed.
3. There exists at least one fragment with a start event.

$$\exists f \in F : f.s \neq \bot$$

4. Every start event creates the case object.

$$\forall s \in \{f.s | f \in F \wedge f.s \neq \bot\}, \forall W \in o(s), \exists do \in W : do.c = d.c_{co}$$

5. The case object remains local.

$$\forall f \in F, \forall do \in f.DO : do.isShared \Rightarrow do.c \neq d.c_{co}$$

6. *Wickr* satisfies structural satisfiability.
7. Each input set and product term satisfies contextual batch processing.
8. For each activity in $a \in N_{A;F}$ and each input set $R \in i_F(a)$ exists an output set $W \in o_F(a)$ and vice versa so that the combination (R, W) is valid. A combination is valid if it satisfies
   a) *Object Behavior Conformance*,
   b) *Contextual Object Creation*, and
   c) no case object is created:

$$\forall do \in W : do.c = d.c_{co} \Rightarrow \exists do' \in R, do'.c = d.c_{co}$$

For each well-formed case model, we define a function io assigning each activity its valid input-output-set combinations. ◇

4.6 Cases in Wickr

The differences between *Wickr* and fCM case models also manifest in their respective cases. The case data that describes the subject and parts of the situation consists of objects and links among objects. Furthermore, instantiating a non-initial fragment is equal to executing its first activity. Therefore, the enabled actions are redefined. Finally, goal multiplicity constraints contribute to the goal definition.

Similarly to fCM, knowledge workers can change the case by performing an action or adapting the case model. In Chapter 8, we discuss how the model can be changed at run time.

Definition 30 (Wickr Case)**.** Let *wickr* $= (d, B, \mathfrak{b}, F, \mathcal{CON}_{term})$ be a case model. A corresponding case is defined by

1. the *Subject* described by a set of data objects O and a set L of links, where
   a) $L \subseteq \{\{o_1, o_2\} | o_1, o_2 \in O \wedge o_1 \neq o_2\}$
   b) *o.id* is the identity of object $o \in O$
   c) *o.class* is the class of object $o \in O$
   d) *o.state* is the state of object $o \in O$
   e) *o.isShared* denotes whether $o \in O$ is a cross-case object.
2. a set $E \subseteq (\mathcal{P}(O) \times N_{A;F} \times \mathcal{P}(\mathcal{O}))$ of enabled actions, where $\mathcal{O}$ is the universe of possible data objects and each action is a tuple (O_r, n_a, O_w) consisting of
   a) a set O_r containing data objects that are read and
   b) a set O_w of data objects that are written
   c) by an instance of activity $n_a \in N_{A;F} = \left(\bigcup_{f \in F} f.N_A\right)$
3. a history H of past actions
4. the current case goal G

◇

4.7 Summary

Wickr is a case management approach adapting and extending fCM to reduce ambiguity in the case definition and emphasize the role of data during case execution. A *Wickr* case model includes a domain model with associations and multiplicity constraints. At runtime, activities are executed to create, update, and link objects. Links are used to select objects that are processed by an activity and to evaluate conditions.

Figure 4.8 depicts the metamodel of *Wickr*. While it does not capture all the details, it shows clearly the dependencies between different parts: The data model is instantiated by executing activities in the fragments. The activities implement state transitions that are modeled in the object

behavior. Furthermore, classes and states are used to specify data conditions, such as the termination condition.

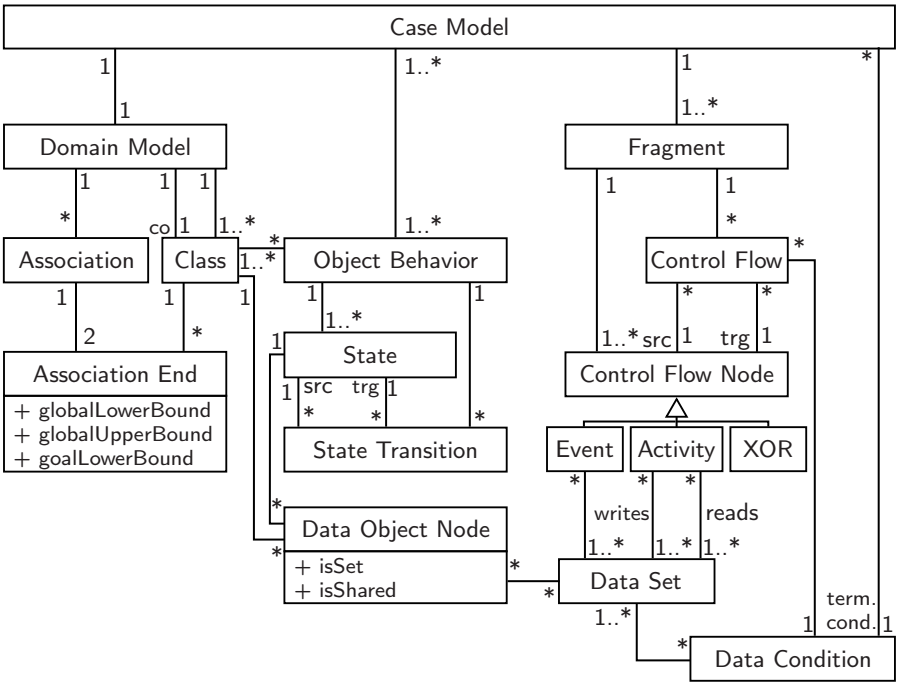


Figure 4.8: Metamodel of Wickr.

Dependencies that are not explicitly modeled are called *hidden dependencies* [12]. Due to hidden dependencies in *Wickr*, one part of the case model may impose constraints on another: Associations and multiplicity constraints impose requirements on object creation. The object behaviors impose requirements on updating objects. If these dependencies are not carefully considered during design-time, the model may contain inconsistencies that prevent proper execution. Therefore, we define some structural correctness criteria that must be satisfied by well-formed case models (Table 4.2) to circumvent some inconsistencies.

So far, we only mentioned how a case model defines the behavior of a case. Process fragments model knowledge workers activities and dependencies among them. The object behavior describes how data objects can be changed. And the data model specifies the structure of the information. However, during a case, all these parts come together. In the next chapters, we formally define the case behavior by translating the case models to Petri nets.

Table 4.2: Overview of the structural correctness criteria for well-formed *Wickr* case models.

Structural Catisfiability	Objects for every local data object node required by an input set or condition must be produced by the case, i.e., a respective data object node must be part of an output set.
Object Behavior Conformance	Only state transitions present in the object behavior are valid.
Contextual Object Creation	When a dependent is created, corresponding supporters must be provided.
Contextual Batch Processing	To determine the objects accessed for a set data object node, a reference object must be included in the input set or product term.

5 A Petri Net-Based Semantics for Wickr

The goal of a case is the resolution of its current situation. Knowledge workers execute actions to change the case and to move it toward the goal. In *Wickr*, actions are modeled as activities with input-output-set combinations. Actions must adhere to the object behavior, and the case data must comply with the domain model. In this chapter, we set out to specify *Wickr's* semantics formally. Focusing on the general data and control flow, we translate case models to classical Petri nets. This translation does not exploit associations and multiplicity constraints, but it is the foundation for respective extensions in Chapter 6.

The semantics are based on fCM's as specified in [117, 143, 144]. A preliminary version of the semantics presented in this chapter was published in a joint paper with Adrian Holfter and Luise Pufahl [165].

5.1 An Example Case

Before presenting the formal semantics, we briefly sketch the example's behavior in a fictive case. Consider a bicycle insurance that offers reimbursement in case a client's bicycle gets stolen.

Mrs. Starley's bicycle was stolen, and she reports the theft to her insurance. Upon receiving Mrs. Starley's claim, the insurance starts a new case (start event of fragment 1 in Figure 4.7 on p. 65). A claim object in state received is created, and an insurance worker checks the completeness of the claim. Unfortunately, Mrs. Starley forgot to add the reference to the police report. Hence, the claim is incomplete.

The insurance worker contacts Mrs. Starley and asks her to update the claim (fragment 2). She sends the missing reference, which is checked. Now, the claim is complete.

Next, the risk can be assessed (fragment 1). The insurance worker considers the risk to be regular. However, at least two reviews for the claim are required (goal multiplicity constraint).

A co-worker, who is eligible to act as an internal reviewer, is assigned and creates a review (fragment 6). Furthermore, an external review is requested (fragment 3). Two weeks pass, and the insurance has not received the external review yet. The insurance worker decides to cancel the assignment (fragment 5) and requests a review from another reviewer (fragment 3). Within a couple of days, the review arrives (fragment 4). Considering both reviews, the worker finally decides to approve the claim (fragment 7). Mrs. Starley receives notice about the approval, and the reimbursement is paid (fragment 1). The insurance worker closes the case (termination condition).

5.2 The Case State

Models for discrete behavior describe states and state transitions. In *Wickr*, a state includes i) data and ii) fragment instances. The state changes when instances of control flow nodes are executed. An activity instance, for example, may create, update, and link objects as well as start, advance, or terminate a fragment instance. To translate a case model to a Petri net, we have to create places for state information.

We consider a case that has not yet started to be in state *initial*. After a start event occurred, it is in state *running*. Eventually, the case is closed and changes to state *terminated*. The Petri net formalization of a case model has the places *initial*, *running*, and *terminated* (see Figure 5.1). In the initial state, one token is in place *initial*.

Figure 5.1: Every case is either in state initial, running, or terminated. This is represented by a token in the respective place. The place's label is written inside the place, following the notation used by CPNTools [56].

5.2.1 Case Data

Every case contains data to describe the subject and the current situation. In *Wickr*, the case data consists of multiple objects and links. Every object has a state, and it is either local to one case or a cross-case object. For now, we do not distinguish between local and cross-case data objects, and we do not support links.

The classical Petri net formalization has a place for each phase. A token in such a place represents a data object of the phase's class in the phase's state. We label the places *Class*[*state*], e.g., *Claim*[*received*]. Figure 5.2 shows the respective places for the insurance example.

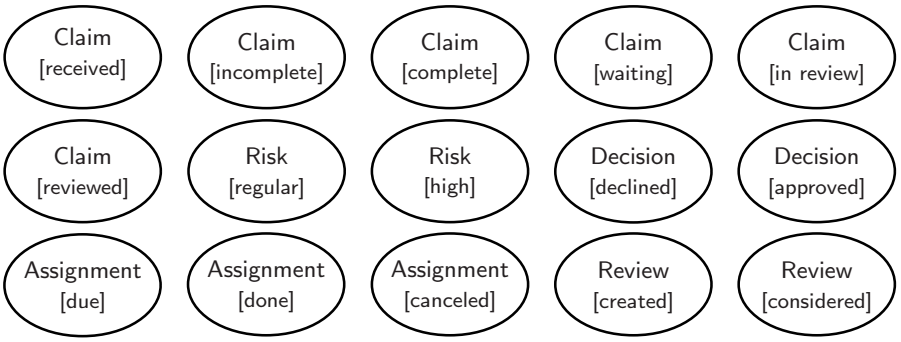


Figure 5.2: The Petri net has a place for each phase. A token on such a place represents a data object that belongs to the phase.

5.2.2 Fragment Instances

The state of a fragment instance consists of the state of its control flow nodes. However, following other formalisms [46, 52], we abstract from the *initial*, *running*, and *terminated* state of activity instances (cf. Figure 5.3). Instead, we formalize the control flow and that it has been triggered. The Petri net for a case model includes a place for each con-

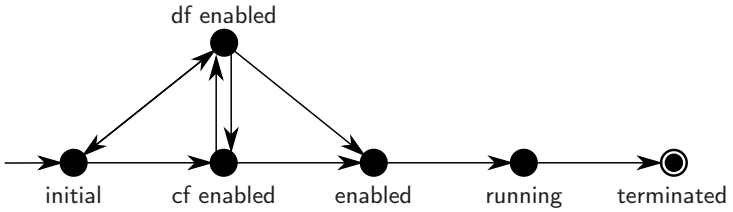


Figure 5.3: The lifecycle of an activity consists of multiple states (cf. [112]). Each activity starts in the initial state. It can be enabled by both control flow and data flow, executed, and terminated.

trol flow arc. A token in such a place represents that the control flow has been triggered, but the target node (activity or gateway) has not terminated yet. Triggering a control flow enables its target node. Since a single fragment may be instantiated multiple times concurrently, a control flow place may have multiple tokens.

For the insurance example (Figure 4.7, p. 65), we add places $p_{i;j}$ for each control flow j in fragment i (see Figure 5.4).[1] A token on the place $p_{1;1}$ represents that the start event has occurred, but activity "check completeness" has not been completed yet.

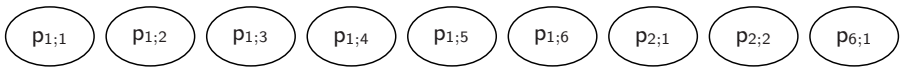


Figure 5.4: Places for the control flow of fragments. A place with label $p_{i;j}$ refers to the control flow j in fragment i.

5.3 The Case Behavior

A new case begins when a start event occurs. During a case, knowledge workers execute activities, which may start fragment instances, advance the control flow, and access data objects. The case can be closed if the goal—consisting of the termination condition and the goal multiplicity constraints—is satisfied. This summarizes the case behavior.

In a Petri net, behavior is defined by transitions, which consume tokens from and produce tokens into places. To capture the full case

[1] The place must identify the control flow clearly. Here, however, we use abstract labels to simplify the net. Later, the context allows identifying the represented control flow arc.

model, events, activities, gateways, and the case termination must be translated into transitions.

5.3.1 Case Instantiation

Each case model has at least one fragment with a start event. Each start event has one or multiple output sets. When a start event occurs, a new case begins, and data objects for one output set are created.

A start event with multiple output sets cannot be translated into a single Petri net transition because transitions are deterministic. They consume a token from each place in their preset and produce a token into each place of their postset. Start events' output sets are exclusive alternatives: Only one of them is chosen.

Given a start event, we create a transition for each of its output sets. Such a transition consumes a token from *initial* and produces a token into *running*. Furthermore, for each data object node in the output set, the transition produces a token into the respective place. The transition also produces a token into the control flow place for the start event's outgoing control flow.

The insurance example has a single start event—"claim received." It has a single output set, which contains a node for claim in state received. It translates to one transition (see Figure 5.5), which consumes a token from *initial* and produces a token into each of the places *running*, *Claim*[*received*], and $p_{1;1}$.

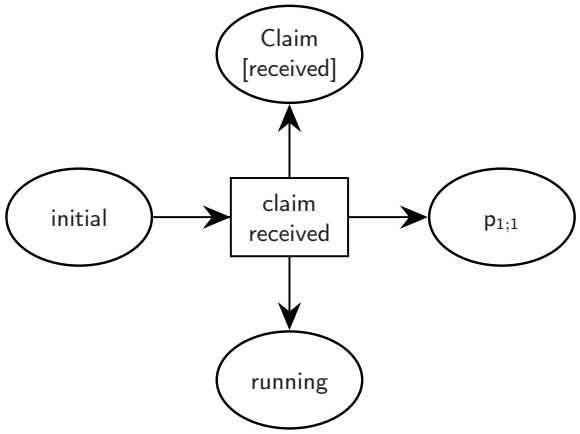


Figure 5.5: The start event of the insurance example is mapped to a single Petri net transition. It sets the state of the case from *initial* to *running*, advances the control flow, and creates a claim in state received.

5.3.2 Case Execution

Once the case has started, other control flow nodes can be executed to change the case state, and to make progress towards the case goal. Activities implement knowledge workers' actions, and gateways model conditional branching. In the Petri net, we represent both as transitions.

Activities

Each activity has a set of valid input-output-set combinations and optionally incoming and/or outgoing control flow. Activities without an incoming control flow start a new fragment instance. They can be executed as long as the case is running, and data objects for at least one input set exist.

An activity is mapped to a set of transitions—one for each valid input-output-set combination. For each data object that is read, a token is consumed and produced. If the state of an object remains the same, the transition reproduces the token into the same place, Otherwise, they produce the token into a different place. Also, when an object is created, a token is produced into the respective place. If the activity has an incoming control flow, a token is consumed from the corresponding place. The transitions for activities without incoming control flow consume a token from and produce a token into place *running*. If an outgoing control flow exists, a token is produced into the respective place.

Activity "assess claim" in the first fragment of the example (see Figure 4.7, p. 4.7) has one input set, two output sets, and two valid combinations. The input set contains a node for the claim in state *complete*. The output-sets contain the risk in state *regular* or *high*.

Activity "assess claim" is mapped to two transitions (see Figure 5.6). One represents outputset •, writing the risk in state *regular*. The transition's preset contains the place $p_{1;2}$ for the incoming control flow and the place *Claim*[*complete*] for the input set. The postset contains the place $p_{1;3}$ for the outgoing control flow, the places *Claim*[*complete*], and *Risk*[*regular*] for the output set •. Transition "assess claim ■" covers the output set ■. The transition has place *Risk*[*high*] instead of *Risk*[*regular*] in its postset and is otherwise the same as "assess claim •".

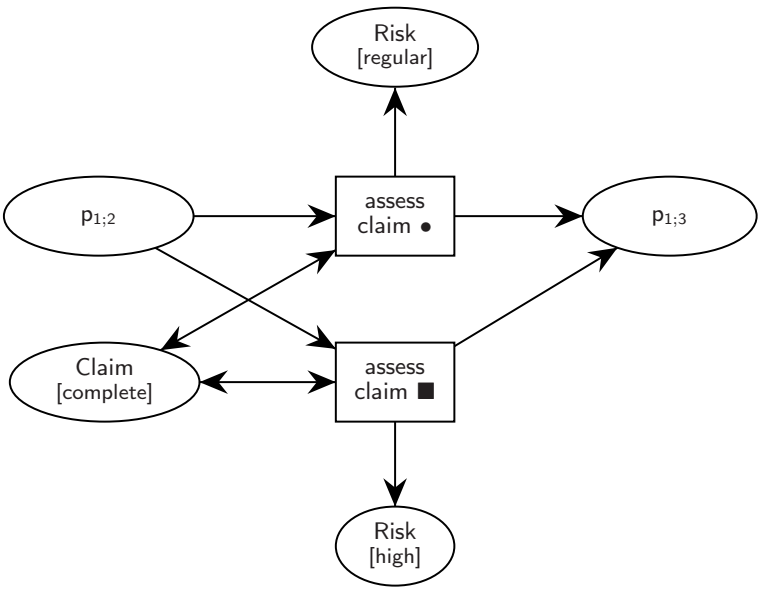


Figure 5.6: The activity "assess claim" is mapped to two transitions, one for each input-output-set combination.

Activity "receive external review" has one valid input-output-set combination but no incoming and no outgoing control flow. It is translated to a single transition (see Figure 5.7). The places *Claim*[*in review*] and *Reviewer*[*external*] in the transition's preset and postset represent the objects that are read but not updated. Furthermore, place *Assignment*[*due*] is in the preset but not in the postset because the data object is updated. The places *Assignment*[*done*] and *Review*[*created*] in the postset represent the objects of the output set. Since the activity has no incoming control flow, the place *running* is added to its preset and to its postset.

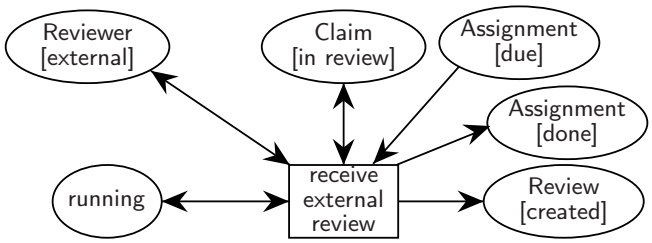


Figure 5.7: The transition for activity "receive external review."

Gateways

Wickr supports only exclusive gateways. A split is an exclusive gateway with multiple outgoing control flows and represents a decision within a fragment. A merge is an exclusive gateway with multiple incoming control flows. When an incoming control flow has been signaled, the gateway is enabled. When it is evaluated, one of the outgoing control flows is triggered. Splits' outgoing control flows may have data conditions: The flow can only be triggered if its condition is satisfied.

In the Petri net formalism, gateways are represented by transitions. We create a set of transitions for each combination of an incoming and an outgoing control flow. More precisely, the set contains one transition for each product term in the data condition of the outgoing control flow. The transition consumes a token for the incoming control flow and produces a token for the outgoing one. Furthermore, places for the data objects required by the product term are added to both the pre- and the postset, so that the case data is not changed. The transition can only fire if the product term is satisfied. If the product term is empty, the control flow is unconditional.

The insurance example has a single gateway. Its outgoing control flow is conditional: The upper branch leading to "reject claim" requires a decision in state *declined*. The lower branch leading to "approve claim" requires a decision in state *approved*. In the Petri net, two transitions represent the gateway (see Figure 5.8). Both transitions have the place $p_{1;3}$ in their presets. The transition for the upper branch consumes a token from *Decision*[*declined*] and produces a token into *Decision*[*declined*] and into $p_{1;4}$. In contrast, the other transition consumes a token from *Decision*[*approved*] and produces a token into *Decision*[*approved*] and

into $p_{1;5}$. Thus, the transitions can only fire if the incoming control flow has been triggered and if their respective branching condition is satisfied.

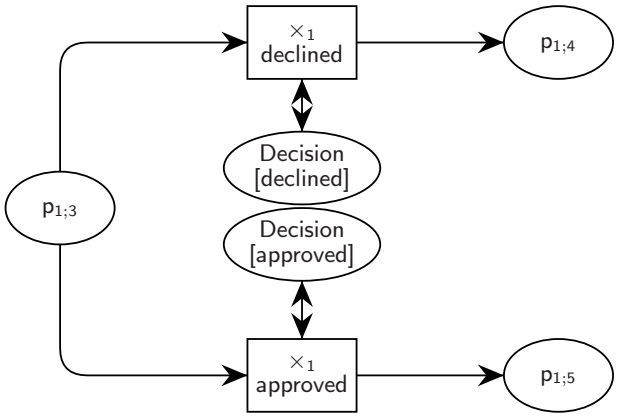


Figure 5.8: The gateway $\times_1$ of the example is represented by two transitions that check the condition and progress the control flow.

5.3.3 Case Termination

By executing activities, knowledge workers advance the case towards the case goal. The goal is modeled by goal multiplicity constraints and the termination condition. If both hold, the knowledge workers can close the case. Once the case has been closed, fragments cannot be instantiated. However, knowledge workers may still have some work left, so already started fragments may still be completed.

The classical Petri net formalization does not capture links among data objects; thus, it does not support goal multiplicity constraints. However, the termination condition is partially supported and mapped to a set of transitions—one for each product term. The transition's preset includes the place *running* and the places for the required data phases. The transition's postset contains the place *terminated* as well as the places for the required data phases. It is enabled, if the case is running and the product term is satisfied. It changes the abstract case state but not the data state or the control flow.

The termination condition of the example requires a claim in state *reviewed* and a decision either in *declined* or in *approved*. Accordingly, the Petri net formalization (see Figure 5.9) has two transitions with the corresponding places in their pre- and postsets.

5.3.4 The Complete Case Model

In this chapter, we described the case from its start to its operation to its termination. The case behavior arises primarily from fragments and the termination condition. However, the domain model and object behaviors limit the case by constraining the input-output-set combinations (cf. Definition 29).

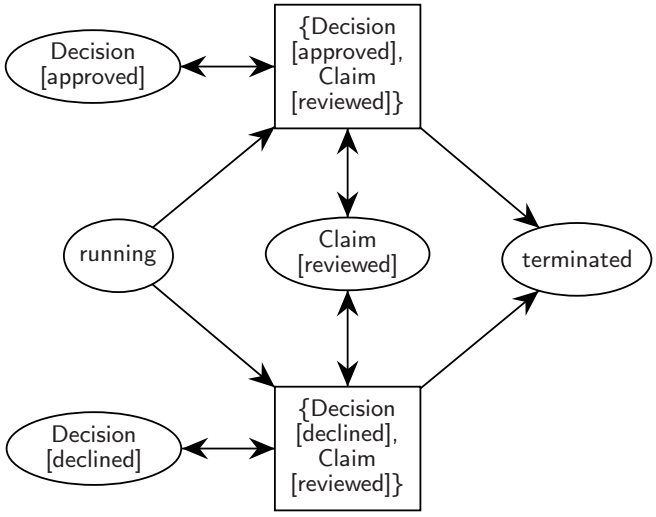


Figure 5.9: Petri net formalization of the termination condition containing one transition for each product term.

While we have not yet considered links and multiplicity constraints, complexity still arises when the full case model is considered: Multiple instances of the same and different fragments can run concurrently and are only synchronized by the data requirements of activities and data conditions. The events, gateways, and activities of one fragment are connected by control flow. Elements of different fragments are connected through data.

This is evident in the Petri net formalization. While each fragment is mapped to a separate set of transitions and places representing control flow, places for data objects can be connected to transitions of different fragments.

Consider fragments one and two of the example. The start event "claim received" instantiates the case and fragment one. Activity "check completeness" may then update the claim to state *incomplete*. If so, fragment two must run to request and receive updates until the claim is complete.

The Petri net mapping in Figure 5.10 shows this dependency. Two transitions represent activity "check completeness." The transition "check completeness ▲" represents the output set for incomplete claims. Upon firing, it produces a token into $\mathit{Claim}[\mathit{incomplete}]$ and $p_{1;2}$. Thereby, transition "request additional information" is enabled. The corresponding activity belongs to fragment two. Similarly, transition "reassess completeness •" enables the "assess claim" transitions belonging to fragment one.

The example in Figure 5.10 covers only two fragments. Yet, it does show the back and forth between fragment instances. Depicting the Petri net for the full case model results in a complex graph. Therefore, we show Petri net excerpts focusing on one fragment at a time. In some cases, we make an exception to this rule to emphasize the connections among fragments.

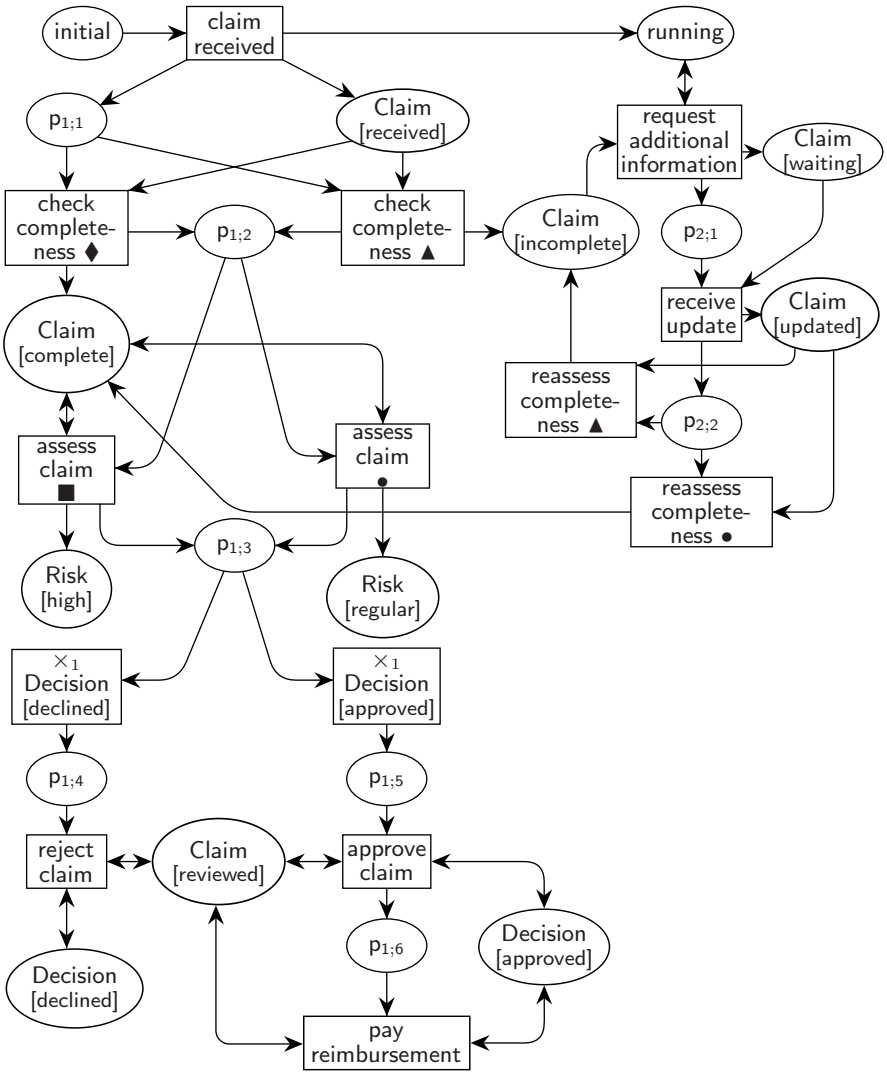


Figure 5.10: Petri net for the first and second fragment of the example.

5.4 Translation to Classical Petri Nets

In this section, we detail the presented translational semantics in a pseudocode algorithm (see Algorithm 1). The labels of places and transitions refer directly to their counterparts in the case model. Yet, in some cases, labels assigned by the algorithm may deviate from the labels used in the previous examples: We use simpler labels in the examples (e.g., for control flow and gateways) to benefit comprehension, and we use detailed labels in the algorithm to describe the mapping more precisely.

We create disjoint sets of places (ll. 1–7) for the abstract case states (l. 1), data objects in particular states (ll. 2–4), and control flow arcs (ll. 5–7).

When a start event occurs, a case is started, data objects for one output set are created, and the outgoing control flow is triggered. Start events are translated to a set of transitions in ll. 8–16. They start a new case

(ll. 11–12) by moving a token from place *initial* to *running*. They create a token for their outgoing control flow (ll. 13–14), and tokens for the data object nodes in one output set (ll. 15–16). If the start event has multiple output sets, it translates to multiple transitions.

Activities are executed by knowledge workers. When one activity is executed, data objects of one input set are read and those of one output set are written. Furthermore, activities my trigger control flow. In the Petri net, activities are represented by a set of transitions (ll. 17–32). We create one transition for each valid input-output-set combination (ll. 19). It consumes a token for each data object that is read (l. 21). If the data object is not updated, the token is reproduced (l. 23). Furthermore, token for data objects that are written are produced (l. 25). If the activity has an incoming control flow, a token is consumed from the respective place (l. 27). Otherwise, a token is consumed from and produced into *running* (ll. 29, 30). Similarly, a token is produced into a control flow place, if an outgoing control flow exists (l. 32).

For gateways, we map each combination of an incoming and an outgoing control flow to a set of transition (ll. 33–42). The set contains one transition for each product term in the condition of the outgoing control flow (l. 37). It consumes a token for the incoming control flow (l. 38) and produces one for the outgoing flow (l. 39). Similarly, a token is consumed and reproduced for each data object node in the product term (ll. 40–42).

The termination condition is also translated to a set of transitions (ll. 43–49). Each transition represents one product term of the termination condition. Such a transition moves a token from *running* to *terminated* (ll. 45, 46). Furthermore, it consumes and reproduces a token for each data object node in the product term (ll. 47–49).

5.5 Summary

The presented mapping formalizes the general control flow and data flow of the case model. A start event instantiates a case. Activities read, create, and update data objects. Once the termination condition is satisfied, one of the respective transitions can fire and the case is closed. Within one fragment, nodes are connected by both data and control flow. Between fragments, only data dependencies exist. The Petri net incorporates both, showing dependencies between different parts of the case model.

Yet, the classical Petri net formalization has limitations. Not supported are object identities, links, multiplicity constraints, set data object nodes, and cross-case data objects are indistinguishable from local ones. Hence, the formalization is less constrained than the case model and may describe unwanted traces. The classical Petri net for the insurance example is incapable of asserting that there are at most three reviews for a claim. It is incapable of considering multiple reviews when “decide on claim” is executed. And it does not know about the relationship between assignments and reviewers.

Algorithm 1: Algorithm mapping a case model to a Petri net. A substring "⟨v⟩" is replaced with the value of variable v.

Input: a well-formed case model *wickr* $= (d, B, \flat, F, \mathcal{CON}_{term})$

Output: a Petri net $pn = (S, T, \xrightarrow{ST}, m_0)$

add places "initial", "running", and "terminated" to $S_{i;r;t}$;
for each *class c in the domain model d*:
  for each *state q in the corresponding object behavior* $\flat(c)$:
    add a place "⟨c⟩[⟨q⟩]" to S_{DO};
for each *fragment f in the set of fragments F*:
  for each *arc* (n, n') *in the control flow* $\xrightarrow{N}$ *of fragment f*:
    add a place "(⟨n⟩,⟨n′⟩)" to S_{cf};
  if *the fragment f has a start event s*:
    for each *output set W of the start event s*:
      add a transition "(⟨s⟩,⟨W⟩)" to T_s;
      add an arc from place "initial" to transition "(⟨s⟩,⟨W⟩)" to $\xrightarrow{i;r;t}$;
      add an arc from transition "(⟨s⟩,⟨W⟩)" to place "running" to $\xrightarrow{i;r;t}$;
      let (s, n) be the control flow leading from s to n;
      add an arc from transition "(⟨s⟩,⟨W⟩)" to place "(⟨s⟩,⟨n⟩)" to $\xrightarrow{cf\ out}$;
      for each *data object node with class c and state q in W*:
        add an arc from transition "(⟨s⟩,⟨W⟩)" to place "⟨c⟩[⟨q⟩]" to $\xrightarrow{write}$;
  for each *activity a in fragment f*:
    for each *valid input-output-set combination* (R, W) *in* $\mathit{io}(a)$:
      add a transition "(⟨a⟩,⟨R⟩,⟨W⟩)" to T_a;
      for each *data object node with class c and state q in R*:
        add an arc from place "⟨c⟩[⟨q⟩]" to transition "(⟨a⟩,⟨R⟩,⟨W⟩)" to $\xrightarrow{read}$;
        if *there exists no data object node with class c in W*:
          add an arc from trans. "(⟨a⟩,⟨R⟩,⟨W⟩)" to place "⟨c⟩[⟨q⟩]" to $\xrightarrow{write}$;
      for each *data object node with class c and state q in W*:
        add an arc from transition "(⟨a⟩,⟨R⟩,⟨W⟩)" to place "⟨c⟩[⟨q⟩]" to $\xrightarrow{write}$;
      if *there exists a control flow* (n, a) *leading to a*:
        add an arc from place "(⟨n⟩,⟨a⟩)" to transition "(⟨a⟩,⟨R⟩,⟨W⟩)" to $\xrightarrow{cf\ in}$;
      else:
        add an arc from place "running" to transition "(⟨a⟩,⟨R⟩,⟨W⟩)" to $\xrightarrow{i;r;t}$;
        add an arc from transition "(⟨a⟩,⟨R⟩,⟨W⟩)" to place "running" to $\xrightarrow{i;r;t}$;
      if *there exists a control flow* (a, n) *starting in a*:
        add an arc from transition "(⟨a⟩,⟨R⟩,⟨W⟩)" to place "(⟨a⟩,⟨n⟩)" to $\xrightarrow{cf\ out}$;
  for each *gateway g in fragment f*:
    for each *control flow arc* (n, g) *leading to g*:
      for each *control flow arc* (g, n') *starting in g*:
        for each *product term* R *in* $f.con(g, n')$:
          add a transition "(⟨n⟩, ⟨g⟩, ⟨n′⟩, ⟨R⟩)" to $T_\times$;
          add an arc from place "(⟨n⟩, ⟨g⟩)" to trans. "(⟨n⟩, ⟨g⟩, ⟨n′⟩, ⟨R⟩)" to $\xrightarrow{cf\ in}$;
          add an arc from trans. "(⟨n⟩, ⟨g⟩, ⟨n′⟩, ⟨R⟩)" to place "(⟨g⟩, ⟨n′⟩)" to $\xrightarrow{cf\ out}$;
          for each *data object node with class c and state q in R*:
            add an arc from place "⟨c⟩[⟨q⟩]" to trans. "(⟨n⟩, ⟨g⟩, ⟨n′⟩, ⟨R⟩)" to $\xrightarrow{read}$;
            add an arc from trans. "(⟨n⟩, ⟨g⟩, ⟨n′⟩, ⟨R⟩)" to place "⟨c⟩[⟨q⟩]" to $\xrightarrow{write}$;
for each *product term R in the termination condition* $\mathcal{CON}_{term}$:
  add a transition "⟨ R⟩" to T_{term};
  add an arc from place "running" to transition "⟨ R⟩" to $\xrightarrow{r;i;t}$;
  add an arc from transition "⟨ R⟩" to place "terminated" to $\xrightarrow{r;i;t}$;
  for each *data object node with class c and state q in product term R*:
    add an arc from place "⟨c⟩[⟨q⟩]" to transition "⟨R⟩" to $\xrightarrow{read}$;
    add an arc from transition "⟨ R⟩" to place "⟨c⟩[⟨q⟩]" to $\xrightarrow{write}$;
Let $S = S_{i;r;t} \cup S_{DO} \cup S_{cf}$;
Let $T = T_s \cup T_a \cup T_\times \cup T_{term}$;
Let $\xrightarrow{ST} = \xrightarrow{i;r;t} \cup \xrightarrow{cf\ in} \cup \xrightarrow{cf\ out} \cup \xrightarrow{read} \cup \xrightarrow{write}$;
Let m_0 be the initial marking with one token in place "initial";

However, these are partly limitations of the mapping, not those of classical Petri nets per se: Multiple cases can be modeled by replicating the Petri net and merging them via places for cross-case data objects. Some multiplicity constraints can be modeled via "reservoir" places from which a token is removed when a data object is created. Yet, to capture the full behavior, classical Petri nets are infeasible. Therefore, we present an extension using colored Petri nets in the next chapter.

6 Associations and Multiplicity Constraints

Data objects, links, and multiplicity constraints are fundamental to *Wickr's* semantics. When activities create and link data objects, they are constrained by multiplicity constraints. When activities read and update data objects, they are constrained by links. Furthermore, the goal multiplicity constraints must be accomplished eventually.

Classical Petri nets are unfit for this semantics. In this chapter, we use colored Petri nets to extend the mapping of the previous chapter. The extended mapping considers object identities, links, and multiplicity constraints.

This chapter is partially based on the publications [158, 160, 166], in which we presented fCM with associations, multiplicity constraints, and corresponding formal semantics, respectively. One of the papers [166] is joint work with Marco Montali.

6.1 Object Identities

In relational databases, primary keys distinguish tuples even if they are otherwise equal. In object-oriented programming, every object has a unique identity. In *Wickr*, data objects have identities, which allow distinguishing objects even if they are otherwise equal. Furthermore, the identities encode the object's class. Identities do not change over time. They are fundamental to *Wickr's* semantics because they are used for linking objects and to bind fragment instances to objects.

When an object is created, a novel identity is produced. The identity must be unique, and it does not change. We implement the identity as a pair that consists of the object's class and an integer. The integer denotes the number of objects of the respective class that existed when the identity was created. Later, we use the class in the identity to find the identities of all objects of a particular class.

The Petri net is extended accordingly. We add a colorset $\mathbb{C}_{ID}$ containing all possible object identities:

$$\mathbb{C}_{ID} = C \times \mathbb{N}_0, \text{where } C \text{ is a set of classes.}$$

Each token representing a data object is a color in $\mathbb{C}_{ID}$, and places holding such tokens are typed accordingly. To count the number of objects, we add $\mathbb{N}_0$ as a colorset and class-specific places with the label $count(Class)$, i.e., $count(Review)$. The places are typed $\mathbb{N}_0$ and have an initial token with value 0.

To access tokens, we add two variables for each class: one for the identity and one for the counter. They are named *Class_ID* and *cnt_Class*

respectively, i.e., *Review_ID* and *cnt_Review*. A transition consuming an identifier does so via the respective variable. A transition creating a new identifier consumes the counter, uses its value for the new identifier, and produces an incremented value.

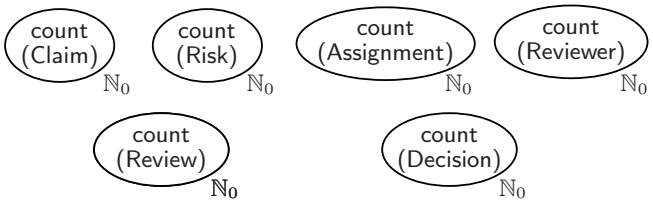


Figure 6.1: Counter places for each class hold an unsigned integer token. The token's value denotes the number of respective objects.

Figure 6.1 shows the counter places for the insurance example. Table 6.1 lists the variables for counters and identities. When an object is created, the counter is used to create a new identity. Figure 6.2 depicts the formalization of fragment four, "receive external review" (cf. Figure 4.7, p. 65) The claim, reviewer, and assignment are read. The transition consumes and reproduces identity tokens, respectively. Since the state of the assignment is changed to *done*, the token is moved to place *Assignment*[*done*]. Also, a review is created. Therefore, the counter is consumed (*cnt_Review*), the novel identity is constructed (*Review, cnt_Review*), and the counter's value is incremented (*cnt_Review+1*).

Table 6.1: Variables for counters and identities used in the insurance example.

	Claim	**Risk**	**Assignment**
var. counter	cnt_Claim	cnt_Risk	cnt_Assignment
var. identity	Claim_ID	Risk_ID	Assignment_ID

	Reviewer	**Review**	**Decision**
var. counter	cnt_Reviewer	cnt_Review	cnt_Decision
var. identity	Reviewer_ID	Review_ID	Decision_ID

In fragment instances, object identities are also used to remember and recognize objects. A fragment instance memorizes the identity of data objects that it has accessed. If the instance subsequently accesses an object of the same class, it must be the memorized one.

Fragment six of the example has two activities: "assign internal reviewer" and "create internal review." When "assign internal reviewer" is executed, an assignment object is created and memorized by the fragment instance. When "create internal review" is executed, the same object is read and updated, although multiple assignments in state *due* may exist.

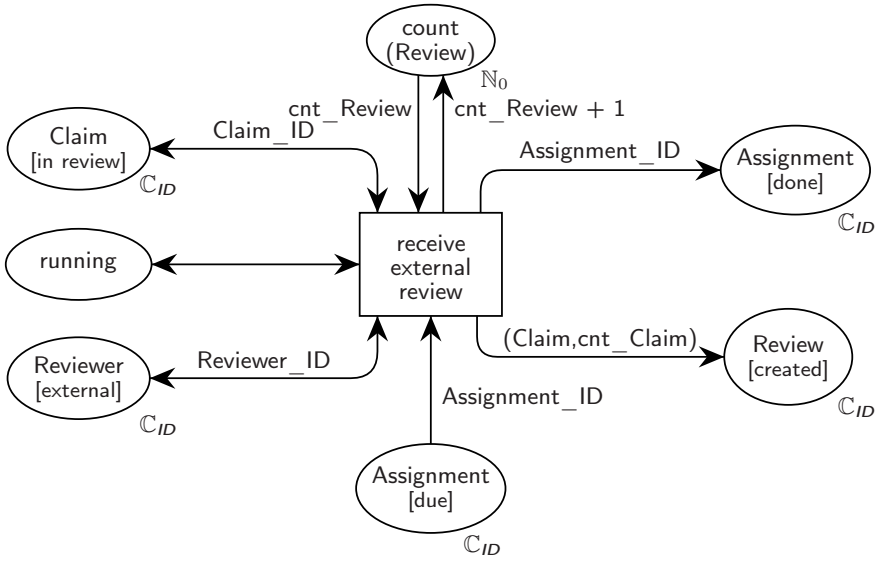


Figure 6.2: Colored Petri net formalization, including identity creation, of fragment five.

In the formalization, we pass the instance's memory alongside the control flow. Therefore, we introduce a colorset $\mathbb{C}_{cf}$, where each color is a function assigning each class an identity or NULL ($\perp$).

$$\mathbb{C}_{cf} = C \to (\mathbb{C}_{ID} \cup \{\perp\}), \text{where}$$

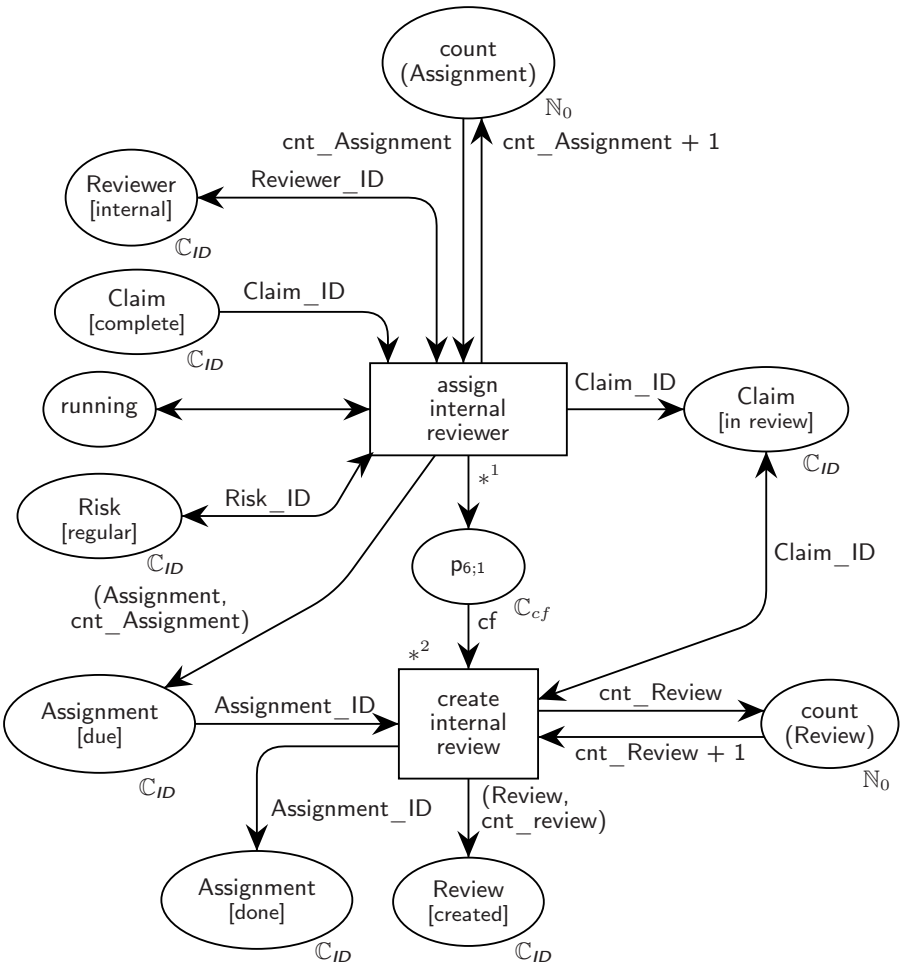
C is the set of classes and $C \to (\mathbb{C}_{ID} \cup \{\perp\})$ is the set of all functions with domain C and codomain $(\mathbb{C}_{ID} \cup \{\perp\})$. All places representing control flow are typed $\mathbb{C}_{cf}$, and we add a variable *cf* with colorset $\mathbb{C}_{cf}$.

Figure 6.3 depicts the updated formalization of fragment six. The data objects accessed by "assign internal reviewer" are memorized in the control flow token. The guard of transition "create internal review" asserts consistent data access: If the control flow's memory for a class is NULL ($\perp$), any object can be accessed. If an identity has been memorized, access is restricted to this particular object.

Of course, if a control flow node has both an incoming and an outgoing control flow, the corresponding Petri net transitions update the fragment instance's memory. The gateway in fragment one of the example accesses a decision object. Since the gateway is the first node in the fragment accessing a decision, the object is memorized. However, the instance also remembers all the objects it has accessed before.

6.2 Links

Classes in a domain model may be associated. In turn, data objects in a case may be linked. The case model's semantics specify when objects get linked, and how links affect the case behavior. For this purpose, information from the domain model and fragments are combined.



$*^1$ arc expression for arc leading to $p_{6;1}$:
$\{(\text{Claim}, \text{Claim\_ID}), (\text{Reviewer}, \text{Reviewer\_ID}), (\text{Decision}, \bot)$
$(\text{Risk}, \text{Risk\_ID}), (\text{Assignment}, (\text{Assignment}, \text{cnt\_Assignment}))\}$

$*^2$ guard of transition "create internal review:"
$[\text{cf}(\text{Claim}) = \bot \vee \text{cf}(\text{Claim}) = \text{Claim\_ID} \wedge$
$\text{cf}(\text{Assignment}) = \bot \vee \text{cf}(\text{Assignment}) = \text{Assignment\_ID}]$

Figure 6.3: Colored Petri net formalization, including control flow, of fragment six.

Links are only created together with objects. This is because associations are limited to existential one-to-one and one-to-many associations: When an object (dependent) whose existence depends on other objects (supporters) is created, it is linked to all respective supporters. Since each association is existential, each link connects a dependent to a supporter. This concludes that all links are established when a dependent is created. Therefore, the supporters must either be read or co-created by the same action that creates the dependent (Contextual Object Creation).

Once established, links affect how data objects are read and data conditions are evaluated. If an activity instance reads two objects whose classes are associated, the objects must be linked. The same is true for data conditions. The links must also exist to objects memorized by the fragment instance. Otherwise, a conflict may occur: If a fragment has three sequential activities, the first one reading an object of class c_1, the second an object of c_2, and the third both objects, and if c_1 and c_2 are associated, then the two objects must be linked. Otherwise, the third activity would either read two objects that should be linked but are not, or it would violate the memory of the fragment. This example shows that objects that are read must be linked to memorized objects.

Fragment three of the example contains the activity "request external review." It creates an assignment, which is linked to the claim and a reviewer. Activity "cancel external review" reads an assignment and a reviewer. The two must be linked, i.e., to inform the reviewer of the canceled assignment and not another one.

In *Wickr*, a link is a bidirectional connection between two objects. We can, therefore, model a link as an unordered pair. We introduce a respective coloreset $\mathbb{C}_l$ as the set of all unordered pairs of object identities:

$$\mathbb{C}_l = \{\{id_1, id_2\} | id_1, id_2 \in \mathbb{C}_{ID}\}$$

This straightforward formalization bears a challenge: In order to check multiplicity constraints and to calculate batches, it must be possible to determine all objects linked to a reference object. Yet, it is, in general, impossible to consume all tokens on a place or to inspect/query the set of tokens in a place. Instead, the tokens consumed by transition must be calculated unambiguously based on other tokens that are consumed. As a workaround, a token can be a set.[1] For this reason, we introduce an additional colorset $\mathbb{C}_L$ containing all possible sets of links:

$$\mathbb{C}_L = \mathcal{P}(\mathbb{C}_l)$$

All links can be stored in a single token whose content can be queried.

Therefore, we add a place *Links* and a variable *links* with colorset $\mathbb{C}_L$ to the Petri net. The place initially holds the empty set. Whenever a new link is established, the token is consumed. The new link is added to the set, and the updated set is stored as a token in place *Links*. Transitions that represent activities, gateways, or the termination

[1] The limitation and workaround have been described by Westergaard in `https://westergaard.eu/2012/07/rfc-conveniences-in-cpn-tools/` (2021/11/06)

condition consume the token in *Links*. Their guards assert that the objects they access are linked according to the associations in the data model. If no additional links are produced, the token is reproduced. Otherwise, it is updated.

Figure 6.4 depicts the colored Petri net of fragment three (cf. Figure 4.7, p. 65). Its activity "request external review" is translated to two transitions. Besides consuming and producing tokens for data objects, the set of links is read and new links between the freshly created assignment, the reviewer, and the claim are added.

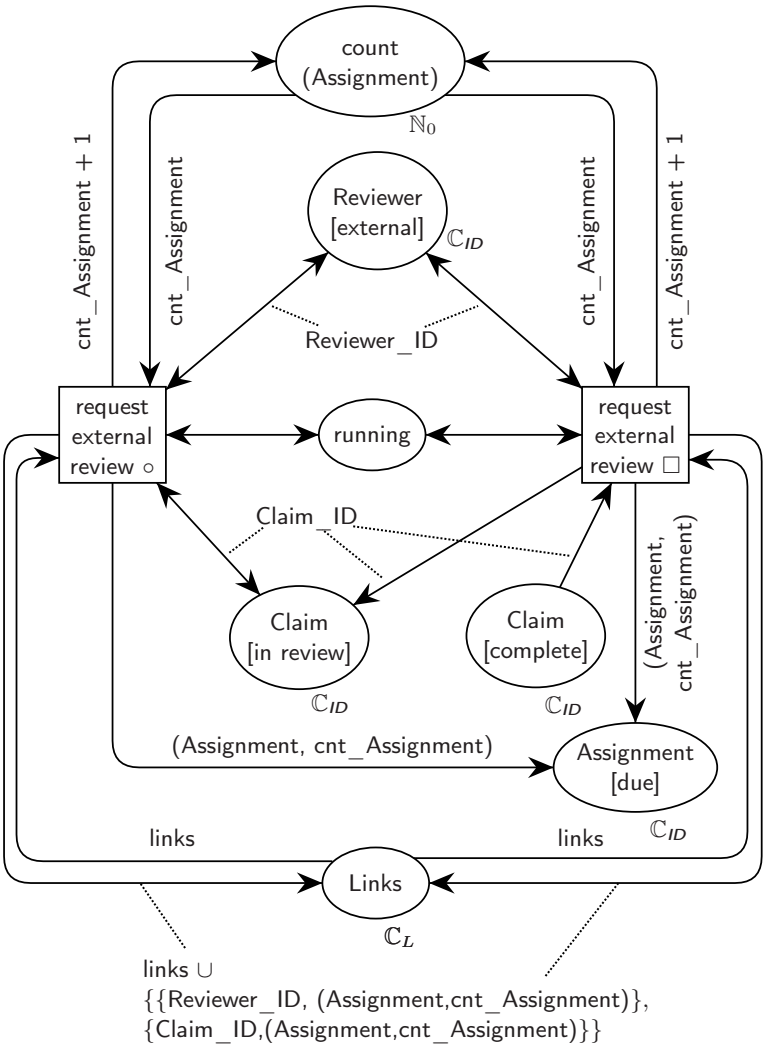


Figure 6.4: Colored Petri net for fragment three demonstrating the creation of links.

Figure 6.5 depicts the colored Petri net of fragment five. Its activity "cancel external review" is translated to a single transition. It consumes and reproduces a token the case's running state, the reviewer, the assignment, and the set of links. The transition's guard checks that a link between the reviewer and the assignment exists.

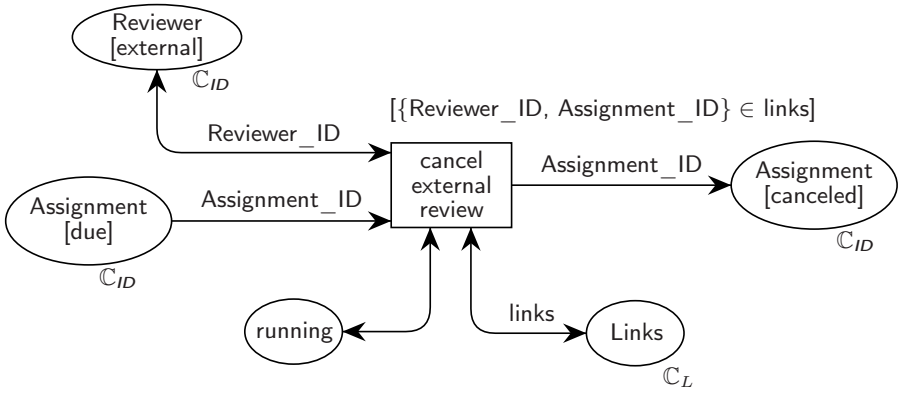


Figure 6.5: Colored Petri net for fragment five of the insurance example. Activity "cancel external review" requires an assignment and a linked reviewer.

6.3 Set Data Object Nodes

Links in *Wickr* serve multiple purposes. Among other things, they are used during batch processing, where a single activity instance operates on a set of similar objects. The execution semantics define this set clearly using links. In the fragments, batch processing is modeled by a set data object node in activities' input (and output) sets.

An input set containing a set data object node must also contain a single node for a reference object (Contextual Batch Processing). The classes of both nodes must be associated. The batch contains all objects of the specified class that are linked to the reference object. Given a claim as a reference object and a set of reviews, the latter contains all reviews linked to the claim.

In the colored Petri net, we have tokens with the identity of data objects and a single token containing the set of links. Given the identity of the reference object and the class of the set data object node, we can query the set of links to determine the content of the set. The query is independent of the data object's state.

Formally, let c_r be the class of the reference object, (c_r, n_r) its identity, c_s the class of the set data object node, and L the set of links. The batch can be determined as the objects of class c_s linked to (c_r, n_r):

$$\{(c_s, n) | \{(c_s, n), (c_r, n_r)\} \in L, n \in \mathbb{N}_0\}$$

This is incorporated into the colored Petri net as follows:

- We introduce a new colorset $\mathbb{C}_{\mathcal{IDS}}$ for sets of identities:

$$\mathbb{C}_{\mathcal{IDS}} = \mathcal{P}(\mathbb{C}_{ID})$$

- We add a variable with colorset $\mathbb{C}_{\mathcal{IDS}}$ for each class. As a name, we choose the plural of the class name (i.e., "Reviews" for the set of reviews).

- When objects are processed in sets, the respective arcs in the Petri net are labeled with this variable.
- When a set of objects is accessed, the guard conditions of the corresponding transition queries the set of links to set the value of the respective variable.
- The guard also asserts that the set is not empty.

Activity "decide on claim" in fragment seven of the insurance example (cf. Figure 4.7) operates on a set of reviews. It has a single input set, in which *Claim*[*in review*] is the reference data object node for the set of reviews. The activity is enabled if the case is running, a claim in state *in review* exists, all linked reviews are in state *created*, and an external reviewer is available. When executed, the activity reads the claim, all corresponding reviews, and the reviewer. It may either consider the reviews to decline the claim (output set ■), approve the claim (output set •), or assign an additional external reviewer (output set ▲).

In the colored Petri net, three respective transitions exist. Figure 6.6 shows the transition for output set •. Based on all reviews, the claim is approved. The transition consumes identity tokens for the claim, the reviewer, and the reviews, and a token for the set of links. The set of reviews is determined by querying the links to retrieve all reviews linked to the claim (see guard condition). Tokens are produced according to the input-output-set combination: The claim and reviews are updated, and new links are added.

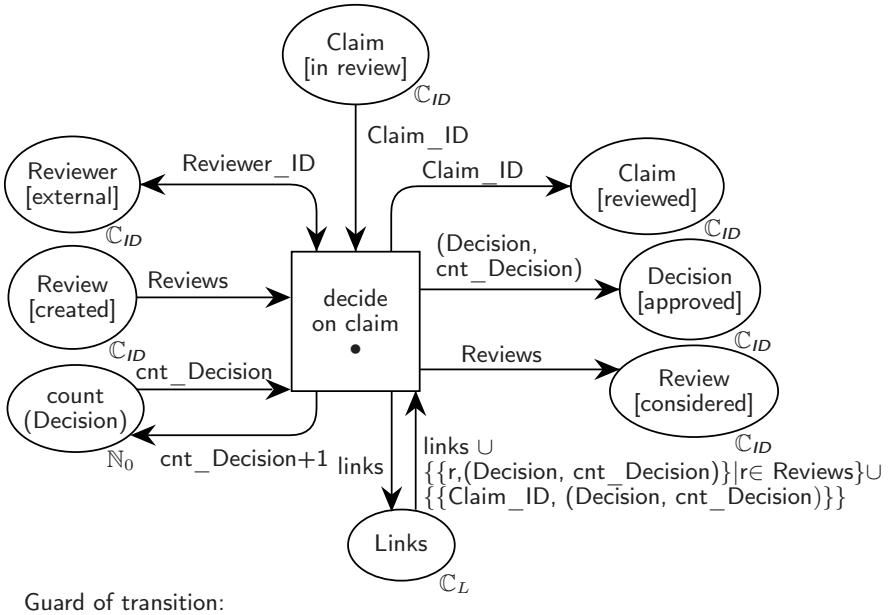


Figure 6.6: Colored Petri net for the input/output behavior of activity "decide on claim" with output set • (place "running" has been omitted).

Figure 6.7 shows the transition for output set ▲. This output set does not include a decision or the reviews. Instead, a new assignment is created and linked to the claim and the reviewer. Respectively, the tokens are consumed/produced and the set of links is updated. The guard condition is again used to determine the set of reviews.

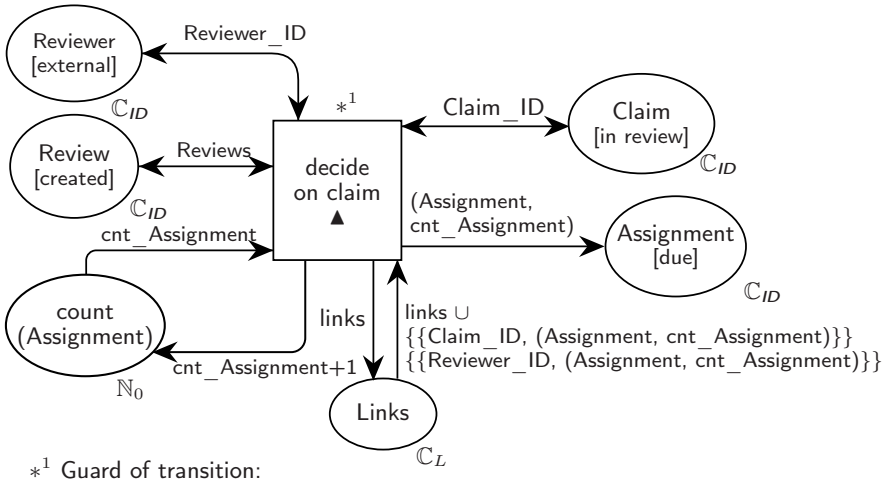


Figure 6.7: Colored Petri net for the input/output behavior of activity "decide on claim" with output set ▲ (place "running" has been omitted).

6.4 Global Multiplicity Constraints

A link is an instance of an association and connects two objects. Both links and objects are created by activities. However, the number of links is constrained by multiplicity constraints. In turn, the execution of activities is limited by multiplicity constraints as well.

First, we consider global multiplicity constraints that define lower and upper bounds that must *never* be violated. Lower bounds must be satisfied during object creation, and upper bounds must not be exceeded. Thus, if executing an activity led to a violation of multiplicity constraints, it could not be executed.

In the example, activity "receive external review" creates a review and links it to the claim. However, a claim is linked to at most three reviews. If this limit has been reached, no new reviews can be created, and "receive external review" is disabled.

Similarly, a decision is linked to at least two and at most three reviews. The activity "decide on claim" can only produce a decision if it reads two or three reviews. If it reads only one, output sets ■ and •, which contain a decision, cannot be written.

In the colored Petri net, the transitions that create links are updated, respectively. Additional guard conditions assert that global multiplicity constraints are met. The current multiplicities are determined by querying the set of links. Given an object with identity id_r and a class

c_l, the number of links from id_r to an object of class c_l is calculated as follows:

$$|\{n_l \in \mathbb{N}_0 | \{id_r, (c_l, n_l)\} \in L\}|, \text{where } L \text{ is the set of links.}$$

Activities that link an object id_r of class c_r to a new object of class c_l must not exceed the multiplicity constraint:

$$|\{n_l \in \mathbb{N}_0 | \{id_r, (c_l, n_l)\} \in L\}| < u(c_l, c_r)$$

Consider activity "receive external review." It instantiates the class Review and links the new object to a claim and an assignment. However, a claim has at most three reviews, and an assignment as at most one review. The activity can only be executed for an assignment and a claim if it does not violate the multiplicity constraints. In the colored Petri net (Figure 6.8), respective guard conditions are added.

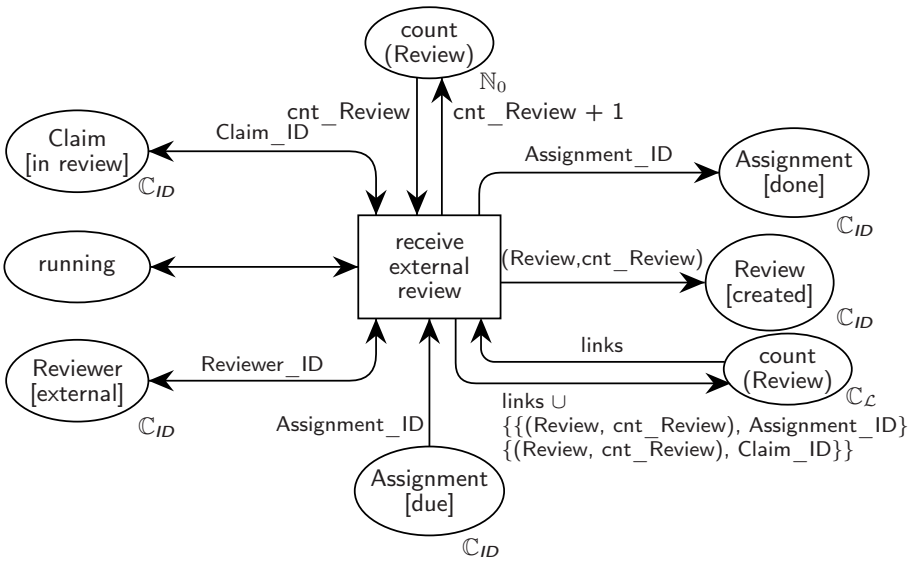


Figure 6.8: Colored Petri net, including guard conditions for the global multiplicity constraints, for fragment four.

The formalization of activity "decide on claim" and output set • is depicted in Figure 6.6. The figure misses the guard conditions for the multiplicity constraint, though:

- The claim must have less than one linked decision.

$$|\{n_d \in \mathbb{N}_0 | \{\textit{Claim\_ID}, (\textit{Decision}, n_d)\} \in \textit{links}\}| < 1$$

- Each read review must have less than one decision.

$$\forall\, \textit{Review\_ID} \in \textit{Reviews}:$$
$$|\{n_d \in \mathbb{N}_0 | \{\textit{Review\_ID}, (\textit{Decision}, n_d)\} \in \textit{links}\}| < 1$$

- Two to three reviews must be read: $2 \leq |\textit{Reviews}| \leq 3$

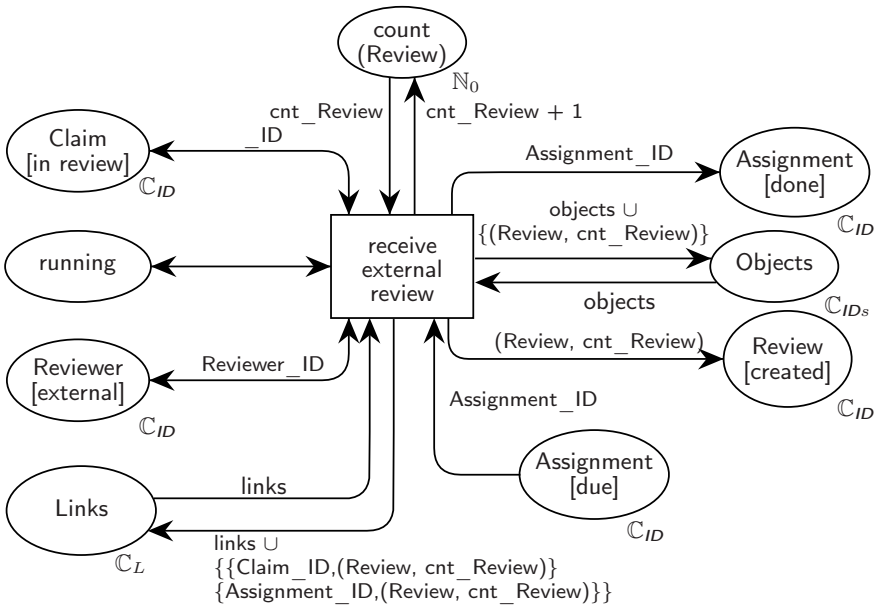
6.5 Goal Multiplicity Constraints

In *Wickr*, domain models contain goal multiplicity constraints. They can refine their global counterpart by defining higher lower bounds. *Wickr's* semantics assert that goal multiplicity constraints hold at case termination. Furthermore, state transitions that hinder the goal multiplicity constraints from ever being accomplished must be prevented. We implement both in our colored Petri net semantics.

To close the case, both the termination condition and the goal multiplicity constraints must hold. The goal multiplicity constraints must be checked for all objects and associations. Therefore, transitions that represent the case termination must query the set of links for all data objects. But objects are represented by tokens that are distributed in the net so that they cannot be accessed all at once. We introduce a global registry of objects as follows:

- A place *Objects* with colorset $\mathbb{C}_{\mathcal{ID}_s}$ that has the empty set as an initial token and acts as a registry of objects.
- A variable *objects* with colorset $\mathbb{C}_{\mathcal{ID}_s}$.

Whenever an object is created, its identity is stored in the registry. Consider the update Petri net for activity "receive external review" (Figure 6.9). The identity of the new review is stored in a token on *Review*[*created*] and in the registry on *Objects*.

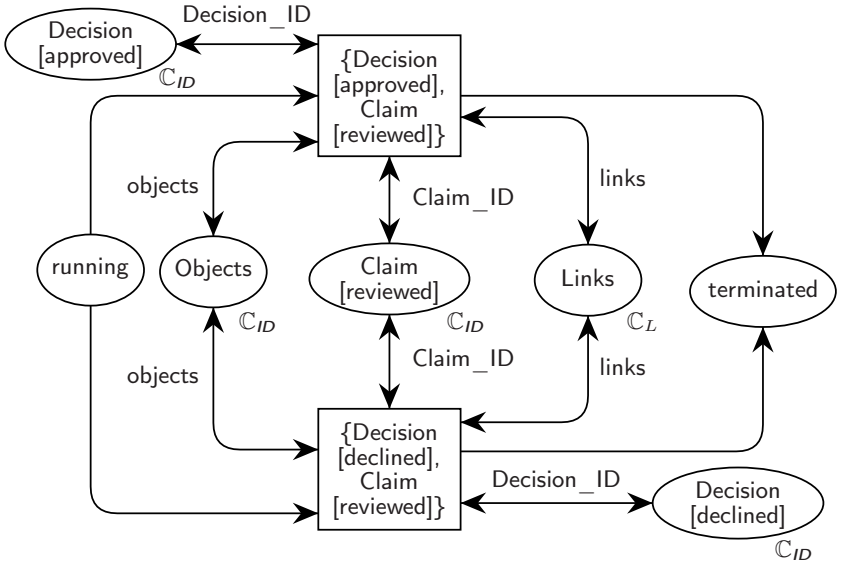


Guard of the transition:
[{Claim_ID, Assignment_ID} ∈ links ∧
{Reviewer_ID, Assignment_ID} ∈ links ∧
$|\{n_r \in \mathbb{N}_0 | \{(\text{Review}, n_r), \text{Claim\_ID}\} \in \text{links}\}| < 3 \wedge$
$|\{n_r \in \mathbb{N}_0 | \{(\text{Review}, n_r), \text{Assignment\_ID}\} \in \text{links}\}| < 1 \wedge]$

Figure 6.9: Colored Petri net formalization, including the registry of objects, of activity "receive external review."

Using the registry of objects and the registry of links, transitions representing the termination condition can check goal multiplicity constraints. Therefore, we check the constraints for each identity in the object registry. We i) find relevant associations, ii) query the set of links to determine the multiplicity, iii) and compare it to the corresponding constraint. An association is relevant if it involves the identity's class and if the corresponding goal multiplicity refines the global one.

In the example, every claim must eventually have one risk, one decision, and at least two assignments and two reviews. Furthermore, each review must be linked to one decision. These requirements are added as guards to the Petri net transitions representing the termination condition (see Figure 6.10).



Guard of both transitions:
$[\{\text{Decision\_ID, Claim\_ID}\} \in \text{links} \wedge$
$(\forall n_c \in \mathbb{N}_0 : (\text{Claim}, n_c) \in \text{objects} \Rightarrow$
$(|\{n_r \in \mathbb{N}_0 | \{(\text{Risk}, n_r), (\text{Claim}, n_c)\} \in \text{links}\}| \geq 1 \wedge$
$|\{n_d \in \mathbb{N}_0 | \{(\text{Decision}, n_d), (\text{Claim}, n_c)\} \in \text{links}\}| \geq 1 \wedge$
$|\{n_r \in \mathbb{N}_0 | \{(\text{Review}, n_r), (\text{Claim}, n_c)\} \in \text{links}\}| \geq 2 \wedge$
$|\{n_a \in \mathbb{N}_0 | \{(\text{Assignment}, n_a), (\text{Claim}, n_c)\} \in \text{links}\}| \geq 2)) \wedge$
$(\forall n_r \in \mathbb{N}_0 : (\text{Review}, n_r) \in \text{objects} \Rightarrow$
$(|\{n_d \in \mathbb{N}_0 | \{(\text{Decision}, n_d), (\text{Review}, n_r)\} \in \text{links}\}| \geq 1))]$

Figure 6.10: Colored Petri net formalization of the case termination. Upon termination, the termination condition and goal multiplicity constraints must hold. Therefore, all objects and all links are considered.

Goal multiplicity constraints may also be important before case termination: Situations that cannot be resolved, i.e., from which the goal cannot be reached, should be avoided. For an example, consider the lifecycle of the claim (cf. Figure 6.11): There is a transition from *complete* to *in review*. Activity "assess claim" reads a claim in state *complete* and links it to a risk. Once the claim is in state *in review*, it is impossible to reach state *complete* again, and hence to link the claim to a risk. Yet, every claim eventually needs a risk. Therefore, a risk must exist when the

claim is changed to state *in review*. In other words, the goal multiplicity must be checked by transitions performing the state change.

We extend the Petri net mapping, respectively. We determine for each state transition in the object behaviors whether an association is *finalized*. An association is finalized if no respective links can be created anymore. Every Petri net transition that implements this state change checks the respective goal multiplicity constraint.

In more detail, given an input-output-set combination, we can determine the associations that are instantiated. Using this knowledge for all input-output-set combinations, we can determine for each phase which associations may get instantiated. For the class Claim, we find that the association to risk is instantiated in state *complete*, the association to assignment in states *complete* and *in review*, and the associations to review and decision in state *in review*. Next, we consider the object behavior and its transitive closure. For each state, we determine the associations that can be instantiated in this state or a future state. A state transition *finalizes* all associations that can be instantiated in the transition's source state but not in its target state or any of its future states. In the example, the association between claim and risk is finalized when the claim transitions to state *in review*. All other associations of the claim are finalized when it changes to state *reviewed*. Figure 6.11 enriches the object lifecycle for claims with respective conditions.

Figure 6.11: Object behavior with guards for goal multiplicities.

Petri net transitions implementing a state change that finalizes an association can only fire if the goal multiplicity constraint holds. Therefore, we add respective guard conditions. In the example, "request external review□" and "assign internal reviewer" change the state of the claim from *complete* to *in review*. Hence, the claim must be linked to one risk when the transition fires. Figure 6.12 shows the extended Petri net for activity "request external review" and input set □. The guard asserts that the claim is linked to at least one risk.

6.6 Extended Translation

In this chapter, we refined the role of data during case execution. Objects are created and linked by activity. Meanwhile, we make sure that i) every object has a unique identity, ii) links do not violate multiplicity constraints, and iii) fragment instances access data consistently. As a result, we can execute batch activities, which operate on sets of similar objects. We also refine the case goal with a separate set of multiplicity constraints. The Petri net mapping was extended respectively. In this section, we provide an overview of the formalization.

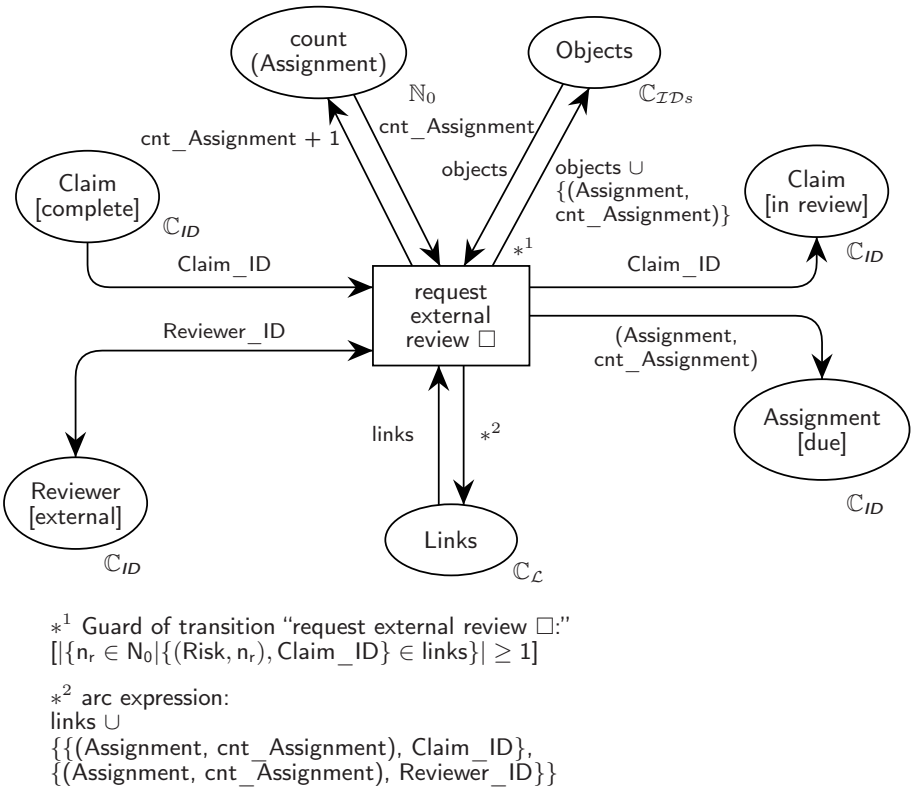


Figure 6.12: Formalization of "request external review" with output set □ including guards for goal multiplicity constraints.

First we revise the formalization of the case state (see Algorithm 2). It is given by the case's abstract state, the case data, running fragment instances, as well as auxiliary information, such as the number of objects for each class. For this, we have to add new places:

- a place "Objects" for the registry of objects (l. 1)
- a place "Links" for the registry of links (l. 1)
- a place "count(c)" for each class c in the domain model d (ll. 2,3)

Algorithm 2: Extending the Petri net produced by Algorithm 1 with colorsets and additional places. Substrings "$\langle v \rangle$" are replaced with the name of element v.

1 add a place "Objects" and a place "Links" to S_{reg};
2 **for each** *class c in the domain model d*:
3 add place "count($\langle$c$\rangle$)" to S_{cnt};
4 Let $S' = S \cup S_{reg} \cup S_{cnt}$ be the set of places;

The set Σ contains all colorsets:

$$\Sigma = \{\{()\}, \mathbb{C}_{cf}, \mathbb{C}_{ID}, \mathbb{C}_{\mathcal{IDs}}, \mathbb{C}_{l}, \mathbb{C}_{L}, \mathbb{N}_0\}$$

The function $c : S' \to \Sigma$ assigns each place a colorset:[2]

$$c(s) = \begin{cases} \mathbb{C}_{cf} & \text{if } s \in S_{cf} \\ \mathbb{C}_{ID} & \text{if } s \in S_{DO} \\ \mathbb{N}_0 & \text{if } s \in S_{cnt} \\ \mathbb{C}_{\mathcal{ID}s} & \text{if } s = \mathit{Objects} \\ \mathbb{C}_{L} & \text{if } s = \mathit{Links} \\ \{()\} & \text{otherwise} \end{cases}$$

We furthermore add arcs to connect the net's transitions to the new places. Every transition must consume and reproduce the set of Links (ll. 5–7) because start events and activities may link data objects and the evaluation of conditions depends on the existing links. All transitions that can create objects must consume and produce the registry of objects. It is furthermore important for closing the case: The contained information is used to check goal multiplicity constraints. Therefore, all transitions, except those representing gateways, consume and reproduce the respective token (ll. 8–10). Furthermore, a class specific counter must be read and incremented whenever the class is instantiated. Thus, respective arcs need to be connected to transitions that represent start events (ll. 11–14) and activities (ll. 15–18). For the latter, we first need to determine which objects are created (ll. 16).

Algorithm 3: Extending the Petri net produced by Algorithm 1 with additional arcs.

5 **for each** *transition t in the set T of transitions*:
6 add the arc (Links, t) to $\xrightarrow{\mathit{read\ L}}$;
7 add the arc (t, Links) to $\xrightarrow{\mathit{write\ L}}$;
8 **for each** *transition t in the set $T \setminus T_\times$ of transitions not representing a gateway*:
9 add the arc $(\mathit{Objects}, t)$ to $\xrightarrow{\mathit{read\ O}}$;
10 add the arc $(t, \mathit{Objects})$ to $\xrightarrow{\mathit{write\ O}}$;
11 **for each** *transition $t = (\langle s \rangle, \langle W \rangle)$ in the set T_s*:
12 **for each** *data object node with class c in W*:
13 add the arc $(\mathit{count}(\langle c \rangle), t)$ to $\xrightarrow{\mathit{cnt}}$;
14 add the arc $(t, \mathit{count}(\langle c \rangle))$ to $\xrightarrow{\mathit{cnt+1}}$;
15 **for each** *transition $t = (\langle a \rangle, \langle R \rangle, \langle W \rangle)$ in the set T_a*:
16 **for each** *class c that is instantiated by (R, W)*:
17 add the arc $(\mathit{count}(\langle c \rangle), t)$ to $\xrightarrow{\mathit{cnt}}$;
18 add the arc $(t, \mathit{count}(\langle c \rangle))$ to $\xrightarrow{\mathit{cnt+1}}$;
19 Let $\xrightarrow{ST'} = \xrightarrow{ST} \cup \xrightarrow{\mathit{read\ L}} \cup \xrightarrow{\mathit{write\ L}} \cup \xrightarrow{\mathit{read\ O}} \cup \xrightarrow{\mathit{write\ O}} \cup \xrightarrow{\mathit{cnt}} \cup \xrightarrow{\mathit{cnt+1}}$;

To access tokens that are in these places, we need variables (see Algorithm 4). We add one variable for the registry of objects (l. 20), one for the registry of links (l. 20), and one for control flow (l. 21). For each class, we add three variables—for the counter (l. 24), a single identity (l. 25), and a set of identities (l. 26) footnoteThe phrase "⟨c⟩s"

[2]Tokens without a value are called *colorless*. In practice, they belong to the colorset $\{()\}$ with a single value.

is replaced with the class name followed by an "s", i.e., "Reviews". respectively.

Algorithm 4: Extending the Petri net produced by Algorithm 1 with variables.

20 add variable "objects" with colorset $\mathbb{C}_{\mathcal{IDs}}$;
21 add variable "links" with colorset $\mathbb{C}_{\mathcal{L}}$;
22 add variable "cf" with colorset $\mathbb{C}_{cf}$;
23 **for each** *class c in the domain model d*:
24 add variable "cnt_⟨c⟩" with colorset $\mathbb{N}_0$;
25 add variable "⟨c⟩_ID" with colorset $\mathbb{C}_{\text{ID}}$;
26 add variable "⟨c⟩s" with colorset $\mathbb{C}_{\mathcal{IDs}}$;

Variables are used in arc expressions and guards to determine the binding, i.e., the combination of tokens consumed by a transition and the resulting assignment of values to variables.

Arcs in $\xrightarrow{read}$ start in a place $c[q]$, where c is a class and q is a state. They lead to transitions representing activities, gateways, and the termination condition. Such an arc has been added for a data object node in the activity's input set or in a condition's product term. Its arc expression depends on the set indicator of the node (see Algorithm 5): If it represents a non-set data object node, it is labeled with a variable for an individual token (l. 31), otherwise with the variable for a set (l. 29).

Algorithm 5: Extending the Petri net produced by Algorithm 1 with arc expressions for consuming data object identities.

27 **for each** *arc* $(\langle c\rangle[\langle q\rangle, t)$ *in* $\xrightarrow{read}$:
28 **if** *the corresponding data object node represents a set*:
29 set the expression of the arc to "⟨c⟩s";
30 **else:**
31 set the expression of the arc to "⟨c⟩_ID";

Arcs in $\xrightarrow{write}$ lead from a transition t to a place $c[q]$. Such an arc represents one or two data object nodes. If a single existing data object is read or updated by an activity, or if it is used in a condition, the corresponding variable is used (Algorithm 6, l. 34). If a novel object is created, the corresponding counter is used to produce a new identity token (l. 36). If a set of objects is read or updated, the respective tokens are bound to the set variable and reproduced (l. 38). However, it is possible that additionally a new object is created and added to the set of existing objects (l. 41). For object creation, we also need to consume and increment the counter (ll. 42–45).

All transitions consume and produce the registry of links (Algorithm 7, l. 47). If no new objects are created, the links remain the same (l. 50). Otherwise, additional links may be added (l. 52). If, for example, a claim and reviewer are linked to a new review, the arc expression reads:

$$\begin{aligned} &\mathit{links} \cup \{\{\mathit{Claim\_ID}, (\mathit{Review}, \mathit{cnt\_Review})\}, \\ &\qquad \{\mathit{Reviewer\_ID}, (\mathit{Review}, \mathit{cnt\_Review})\}\} \end{aligned}$$

Algorithm 6: Extending the Petri net produced by Algorithm 1 with arcs and arc expressions for producing data object identity tokens.

32 **for each** *arc* $(t, \langle c \rangle[\langle q \rangle])$ *in* $\xrightarrow{write}$**:**
33 **if** *a single corresponding data object is read or updated*:
34 set the expression of the arc to "⟨c⟩_ID";
35 **elif** *a single corresponding data object is created*:
36 set the expression for the arc to "(⟨c⟩, cnt_⟨c⟩+1)";
37 **elif** *a set of corresponding data objects is read or updated to another state*:
38 set the expression of the arc to "⟨c⟩s";
39 **elif** *there are two corresponding data object nodes in the output set OR*
40 *a set data object node in the input set and a non-set data object node in the output set*:
41 set the expression of the arc to "$\langle c \rangle s \cup \{(\langle c \rangle$,cnt_$\langle$ c$\rangle)\}$";
42 **for each** *arc in* $\xrightarrow{cnt}$**:**
43 set the arc expression to "cnt";
44 **for each** *arc in* $\xrightarrow{cnt+1}$**:**
45 set the arc expression to "cnt+1";

Transitions for events, activities, and the termination condition consume the registry of objects (l. 54). If no new objects are created, the registry is reproduced (l. 57) Otherwise, the new objects are added to the registry (l. 59). A transition creating a review (and only a review) leads to the following arc expression:

$$objects \cup \{(Review, cnt\_Review)\}$$

Algorithm 7: Extending the Petri net produced by Algorithm 1 with arcs and arc expressions for the registries.

46 **for each** *arc* $(Links, t)$ *in* $\xrightarrow{read\ L}$**:**
47 set the arc expression to "links";
48 **for each** *arc* $(t, Links)$ *in* $\xrightarrow{write\ L}$**:**
49 **if** *no new links are created*:
50 set the arc expression to "links";
51 **else:**
52 set the arc expression so that new links are added;
53 **for each** *arc* $(Objects, t)$ *in* $\xrightarrow{read\ O}$**:**
54 set the arc expression to "objects";
55 **for each** *arc* $(t, Objects)$ *in* $\xrightarrow{write\ O}$**:**
56 **if** *no new objects are created*:
57 set the arc expression to "objects";
58 **else:**
59 set the arc expression so that new objects are added;

Finally, we update the arcs from and to control flow places (Algorithm 8). Arcs originating in places representing control flow nodes have the label "cf" (l. 61). Arcs leading to such places have a novel function: If the corresponding control flow node has an incoming control flow, the consumed token is updated by assigning identities to additional classes (l. 64). Otherwise, a new control flow token is created that

assigns identities for the consumed and produced data object tokens to the respective class (l. 66). If a set of data objects is accessed (batch processing), it is not considered in the control flow.

Algorithm 8: Extending the Petri net produced by Algorithm 1 with arcs and arc expressions for control flow.

60 **for each** *arc in* $\xrightarrow{cf\,in}$**:**
61 set the arc expression to "cf";
62 **for each** *arc in* $\xrightarrow{cf\,out}$**:**
63 **if** *the corresponding control flow node has an incoming control flow*:
64 set the arc expression to update "cf";
65 **else:**
66 set the arc expression to a novel function memorizing accessed objects;

Now that arc expressions to consume and produce tokens are in place, we can add guards to detail the relationship among tokens further. Whenever two identity tokens of associated classes c_1 and c_2 are consumed, the objects must be linked. We add the term

$$\{c_1\_ID, c_2\_ID\} \in \mathit{links}$$

Whenever a set of objects of class c_b that is linked to a reference object of class c_r is consumed, we add the term

$$c_bs = \{(c_b, n) | \{(c_b, n), c_r\_ID\} \in \mathit{links}\},$$

where c_bs is the set variable for c_b. Whenever a control flow token and a data object token of class c are consumed, we add the term

$$\mathit{cf}(c) = \bot \vee \mathit{cf}(c) = c\_ID$$

Whenever a new link between an existing object of class c_e and a novel object of class c_n is created, we add the term

$$|\{n \in \mathbb{N}_0 | \{(c_n, n), c_e\_ID\} \in links\}| < u(c_n, c_e)$$

Whenever a link between a set of objects of class c_b and a novel object of class c_n is added, we add two terms. One to check that the multiplicity constraint for the new object is not violated:

$$l(c_b, c_n) \leq |c_bs| \leq u(c_b, c_n)$$

Furthermore, we add a term to check the multiplicity constraints for each object in the set:

$$\forall c_b\_ID \in c_bs : |\{n \in \mathbb{N}_0 | \{(c_n, n), c_b\_ID\} \in \mathit{links}\}| < u(c_n, c_b)$$

We must also consider goal multiplicities. If an association is finalized for an object, the goal multiplicity must hold. Let the object belong to class c_r, and let the association that is finalized connect c_r to class c_f. If the transition creates a novel identity token for an object of class c_f, we add the term

$$|\{n \in \mathbb{N}_0 | \{c_r\_ID, (c_f, n)\} \in \mathit{links}\}| = \diamond l(c_f, c_r) - 1$$

Otherwise, we add the term

$$|\{n \in \mathbb{N}_0 | \{c_r\_ID, (c_f, n)\} \in \mathit{links}\}| = \diamond\ell(c_f, c_r)$$

An association can also be finalized when the corresponding class is instantiated. Then it may also be linked to a set of objects. We add the term

$$\diamond\ell(c_f, c_r) \leq |c_f s|$$

where $c_f s$ is the set of c_f objects. Associations can also be finalized for a set of c_r objects, in this case we have to add the all quantifier "$\forall c_r\_ID \in c_r s''$ in front, where $c_r s$ refers to the set variable of c_r. Furthermore, a new object of class c_f may be created in the same step—if so, we must add 1 to the size of the set $c_f s$.

If a transition represents a termination condition, we furthermore add a term checking all goal multiplicities:

$$\forall (c_r, n_r) \in \mathit{objects}, \forall c \in d.C :$$
$$|\{n \in \mathbb{N}_0 | \{(c_r, n_r), (c, n)\} \in \mathit{links}\}| \geq \diamond\ell(c_n, c_r)$$

We can also omit checks, if the goal multiplicity constrained equals the global multiplicity constrained.

The set of terms in a guard asserts data access that respects links, consistent data access across a fragment instance, proper batch processing, as well as goal and global multiplicity constraints.

Finally, we revise the initial marking. In colored Petri nets, it is defined by an initialization function *init*, which assigns each place $s \in S$ a multiset of tokens. Our colored Petri nets have a token without value on the place *initial* and a token with value $\emptyset$ on the places *Objects* and *Links*.

$$\mathit{init}(s) \begin{cases} \{()\}_{ms} & \text{if } s = \mathit{initial} \\ \{\emptyset\}_{ms} & \text{if } s = \mathit{Objects} \vee s = \mathit{Links} \\ \emptyset_{ms} & \text{otherwise} \end{cases}$$

6.7 Summary

In summary, associations and multiplicity constraints have significant impact on the execution of flexible processes. During process execution, activities create, link, read, and update data objects. Since links are constrained by multiplicity constraints, activities are constrained as well. Therefore, multiplicity constraints may pose lower and upper bounds on the number of activity instances. Formal execution semantics must consider multiplicities and respective constraints to describe the process behavior precisely. In this chapter, we presented respective details of the mapping, which involve all control flow nodes and the termination condition.

Start events create and link data objects. Furthermore, they pass a reference to all created objects along the control flow.

Activities read, create, update, and link data objects. They can even read, update, and link sets of objects for a single data object node. When an activity is executed, the semantics assert that possibly linked data objects are linked, the inputs comply to the references passed along the control flow, and global and goal multiplicity constraints are not violated.

Gateways may evaluate data conditions. In contrast to the classical Petri net formalization, links, references in the control flow, and set data object nodes are considered.

Case termination can only occur if a set of objects that adheres to links satisfies the termination condition and if the goal multiplicity constraints are satisfied.

To support this behavior, we extended the mapping by using colored Petri nets. We introduced object identities and global registries for links and objects. Furthermore, batch processing is supported. The constraints of the data model and the object behavior affect the execution of cases directly. This is mostly realized through guards on the transitions in the colored Petri net. Violations of multiplicity constraints are prevented by the semantics.

The formalization, however, is limited to a single case. Even if we place multiple tokens in the place *initial*, where each token represents a case, the mapping is incapable of keeping the cases separate. In the next chapter, we present an extension that handles multiple concurrent cases of the same and different case models. The cases connected through cross-case data objects.

7 Sharing Data Among Cases

So far, we only considered one case of one case model at a time, but cases are not always isolated [110]. They may communicate and/or share data. In *Wickr*, cross-case data objects exist beyond the boundaries of a single case and can be accessed by multiple cases concurrently. These cases can be instances of the same or different case models.

In this chapter, we extend the formal semantics to differentiate between local and cross-case data objects. The semantics supports multiple cases of the same and/or different case models. Furthermore, we present and discuss attribute-based correlation of data objects to cases and the influence of links and multiplicity constraints.

This chapter is based on a previous publication [159]. In the paper, we present the idea of cross-case business processes using BPMN process models. In contrast to the paper, we also consider links in this chapter.

7.1 Case Identities

We introduced object identities (see Section 6.1) to distinguish data objects. To distinguish cases, we introduce case identities, and we link local data objects and fragment instances to their case.

When a start event occurs, a case changes from state *initial* to *running*. At the same time, a new case enters the *initial* state. To handle multiple concurrent cases, case-specific information must be clearly linked to one case, this includes the abstract case state, the states of running fragment instances, and local data objects.

We extend the colored Petri net mapping accordingly. New colorsets are added, and the assignment to places is adapted to store the case identity. We introduce the colorset $\mathbb{C}_{\text{case}}$ of possible case identities:

$$\mathbb{C}_{case} = Names \times \mathbb{N}_0, \text{where } Names \text{ is a set of case model names.}$$

The places *initial*, *running*, and *terminated* have the colorset $\mathbb{C}_{case}$. Furthermore, we add a variable *Case_ID* with the colorset $\mathbb{C}_{case}$.

Local data objects belong to a case. For this reason, we introduce a colorset $\mathbb{C}_{local}$ and type all respective places accordingly:

$$\mathbb{C}_{local} = \mathbb{C}_{case} \times \mathbb{C}_{ID}$$

This means the mapping has to create separate places for local and cross-case data object nodes. In our examples, the labels of places for cross-case data objects have the prefix "cc" (cf. Figure 7.1). Their colorset is $\mathbb{C}_{ID}$.

Furthermore, a new colorset for control flow places is added:

$$\mathbb{C}_{case,\,cf} = \mathbb{C}_{case} \times \mathbb{C}_{cf}$$

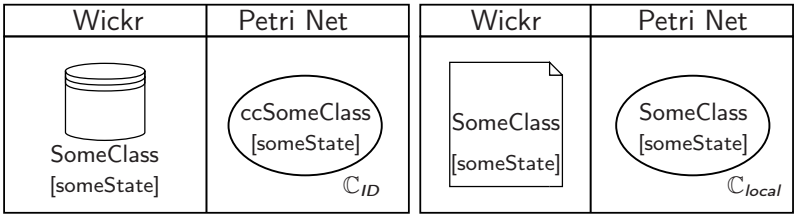


Figure 7.1: Places for local and cross-case data objects are mapped separately. Therefore, two places for the same phase can exist.

Goal multiplicity constraints are part of a case model's goal specification and translate directly to the case goal. They have to be checked separately for each case. Therefore, we change the colorset for the registry of objects:

$$\mathbb{C}_{case,\mathcal{IDS}} = \mathbb{C}_{case} \times \mathbb{C}_{\mathcal{IDS}}$$

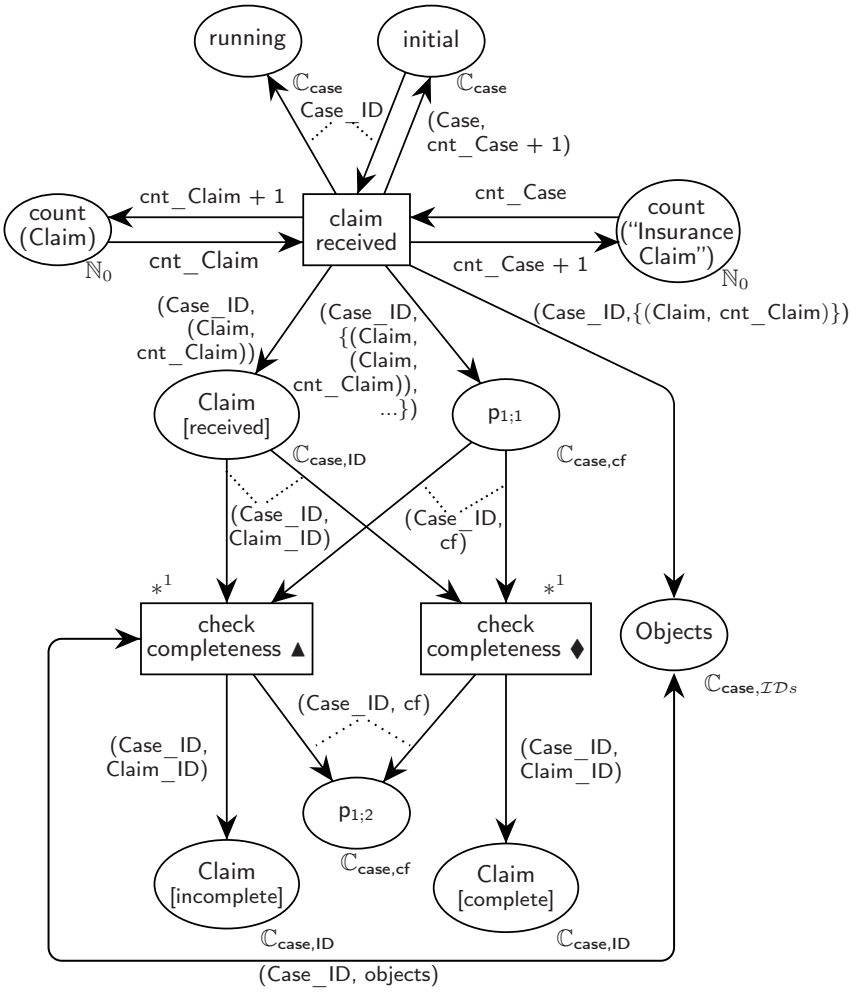
In conclusion, tokens that belong to one case—such as local data objects, the object registry, and control flow tokens—link to their case.

We can model multiple cases by placing respective tokens in *initial*. Yet, in reality, arbitrarily many cases may exist. Therefore, we generate a novel case identity whenever a case starts. The generation is similar to the creation of object identities (see Section 6.1). In the initial marking, a token "(case model name, 0)" is in place *initial*. Furthermore, a place *count*(*case model name*) and a variable *cnt_case* of type $\mathbb{N}_0$ are added. In the initial marking, the place holds a token with value 0. When a new case begins, the case identity token on *initial* is moved to *running*, a case-specific object registry is created by creating a new token on *Objects*, the case-specific counter is incremented, and a novel case identity is produced into place *initial*. All subsequent transitions consume at least one token with a case identity. If multiple tokens with a case identity are consumed, they must belong to the same case.

The updated formalization of the insurance example's start event and the subsequent activity "check completeness" is depicted in Figure 7.2. Mapped to a single transition, the start event begins a new case by moving the respective token from *initial* to *running*. In the scope of the case, the event creates an object of type claim and initializes the case's object registry. Finally, it also produces a control flow token for fragment one and an identity token for the next case. Two transitions exist for "check completeness." Each consumes the control flow token and the token of the claim. By binding the case identity of both tokens to the same variable, the formalization asserts that the tokens belong to the same case. Of course, the case identity does not change when the state of objects is updated (here to *incomplete* or *complete*).

7.2 Cross-Case Data Objects

Fragment instances and local data objects belong to exactly one case, but *cross-case data objects* may be accessed—i.e., created, read, updated,



$*^1$ Guard of both "check completeness" transitions:
[cf(Claim)=$\perp \vee$ cf(Claim)=Claim_ID]

Figure 7.2: Colored Petri net formalization of the start event "claim received" and the subsequent activity "check completeness." To handle multiple cases, tokens refer to a case. The place *Links* is omitted because all transitions consume and reproduce the registry of links without operating on its color/value.

and linked—by multiple cases concurrently. Therefore, they can be used to synchronize different cases.

In the colored Petri net formalization, tokens that represent local objects refer to a case, tokens for cross-case data objects do not. Yet, multiplicity constraints must also hold for cross-case data objects. Since multiple cases may create links for a cross-case data object, we have to check multiplicity constraints whenever we access such an object. Also, to check goal multiplicity constraints, we add cross-case data objects to the case-specific object registry when they are accessed.

Consider fragment three of the example (see Figure 4.7, p. 65). Its activity "request external review" requests a review from an external reviewer. The reviewers are cross-case data objects and created by a different process. Yet, the activity links the reviewer to a local assignment. Thereby, it is also added to the registry of objects.

In the colored Petri net (see Figure 7.3), the place *ccReviewer*[*external*] has colorset $\mathbb{C}_{ID}$. The activity "request external review" is mapped to two transitions. Each of the transitions consumes and produces a token representing a reviewer. Furthermore, the set of links is extended by linking the new assignment to the reviewer and the claim. While the reviewer object has existed before, it is now added to the object registry. Since the registry is a set, it does not contain duplicates.

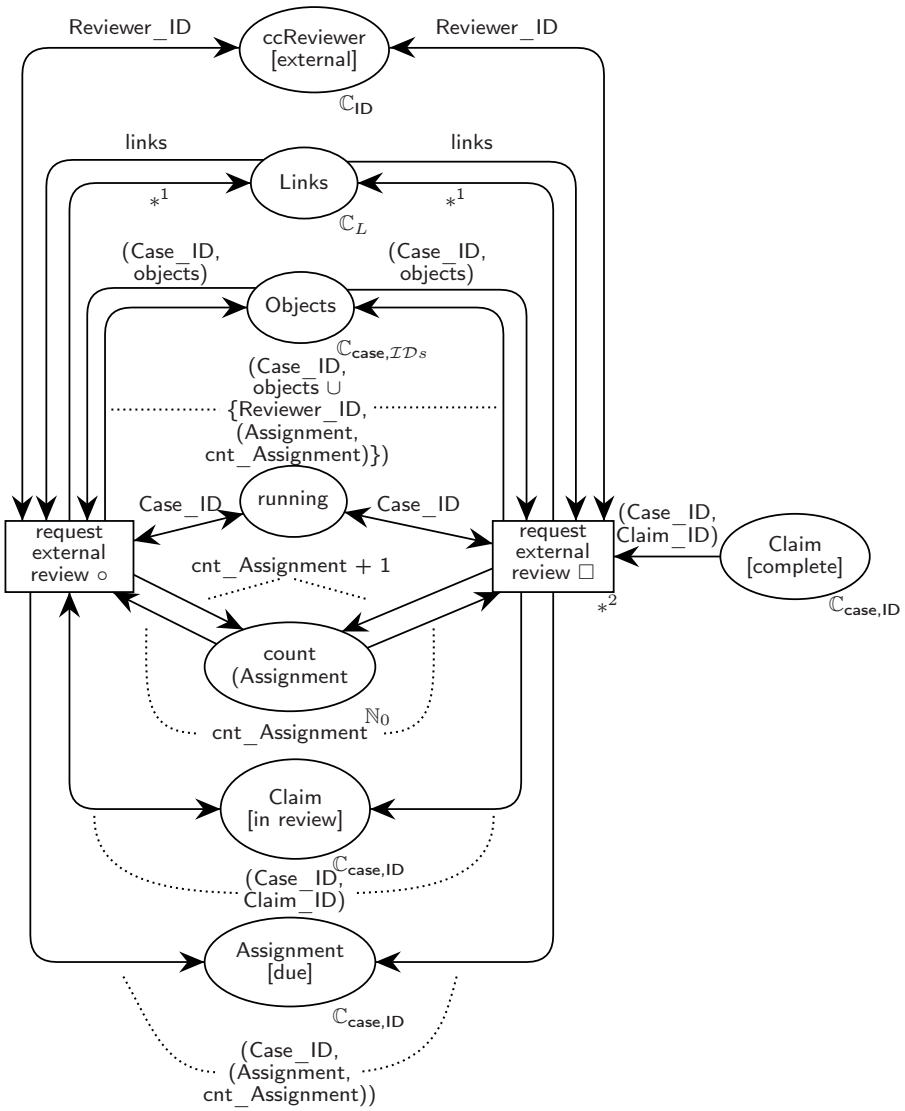
A cross-case data object may exist before a case. This is modeled by respective tokens in the initial marking. Furthermore, cross-case data objects can connect cases of different case models. To model this, each case model can be formalized separately. The resulting Petri nets can be merged: All data places—those for data object nodes, the object registry, and the registry of links—exist only once. This means that if two case models have the same data object node, the transitions are connected to the same place. All other places—*initial*, *running*, and *terminated* as well as control flow places—are case-model-specific.

7.3 Allocating and Publishing Cross-Case Data

Cross-case data objects represent data that is available to many cases. They can be used to model various relationships among cases: There can be a database that is accessed by cases concurrently. There can be a handover where one case takes over an object that has previously been used by another case. And there can be a back-and-forth between cases. To model these behaviors accurately, we must be able to move cross-case data objects into the scope of one case and to publish local objects as cross-case data objects.

An activity can allocate a cross-case data object to a case. Therefore, it must have a valid input-output-set combination that reads a cross-case data object and writes a local data object of the same class.

In the Petri net mapping, a token for the cross-case data object is consumed and one for the local data object is produced. The tokens' object identities are the same. However, the local object links to the case, but the cross-case object did not.



arc expressions $*^1$:
links ∪
{{Reviewer_ID, (Assignment, cnt_Assignment)},
{Claim_ID, (Assignment, cnt_Assignment)}}

guard $*^2$ of transition "request external review □:"
$[|\{n_r \in \mathbb{N}_0 | \{(Risk, n_r), Claim\_ID\} \in links\}| \geq 1]$

Figure 7.3: Colored Petri net formalization of activity "request external review." The cross-case data object *reviewer* does not reference the case. Yet, it is linked to local objects and may be referenced by fragment instances. We use dotted lines to link expressions to arcs.

Similarly, a local data object may be published by activities that read a local data object and write a cross-case data object of the same type. Thereby, the case identity is stripped from the local data object token, and the object identity is produced as a cross-case data object token. However, the central case object (i.e., the claim) must remain local during the whole case.

The abstract fragment in Figure 7.4 has two activities. Activity "a_1" reads a cross-case data object of class c_1 in state q_1 and allocates it to the case (local data object in state q_1). Afterwards, activity "a_2" reads the local data object and releases it (cross-case data object in state q_2). The figure also contains the colored Petri net formalization. To identity the cross-case object, the token is enriched with the case identity and moved to the respective place. To publish it, the case identity is removed.

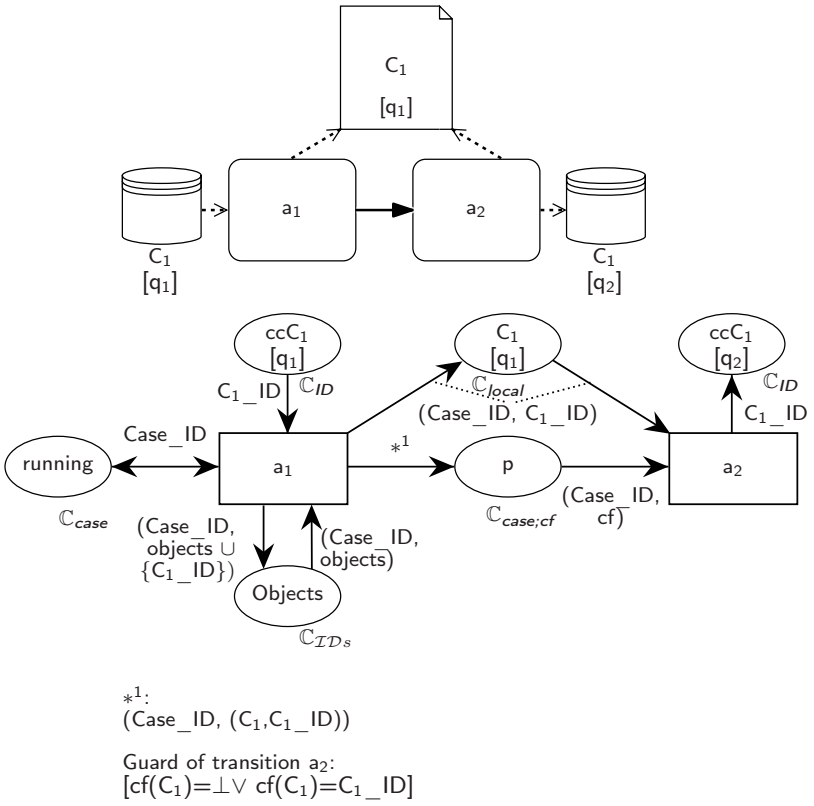


Figure 7.4: Fragment with two activities and its formalization. The first activity allocates a cross-case data object; the second one publishes a local object.

7.4 Correlating Cross-Case Data to Cases

Given multiple cross-case data objects of the same class in the same case, the semantics allow activities to access any of them. However, the correlation of cross-case data objects to cases may be constrained. Therefore, we discuss attribute-based correlation and its implementation in colored Petri nets.

First, we revise the insurance example. The insurance offers different products to insure clients against theft, accidents, natural disasters, and more. Furthermore, a bank account is required to pay the reimbursement. We add three respective classes (see Figure 7.5): *Category*, *Categorization*, and *BankAccount*. Categories and bank accounts are cross-case data objects, while the categorization is a local object that connects a claim to its category. Every claim belongs to one of these categories, and reviewers are specialized in one category. We also update fragment one as depicted in Figure 7.6: Initially, each claim is categorized, and a solvent bank account is accessed to pay the reimbursement.

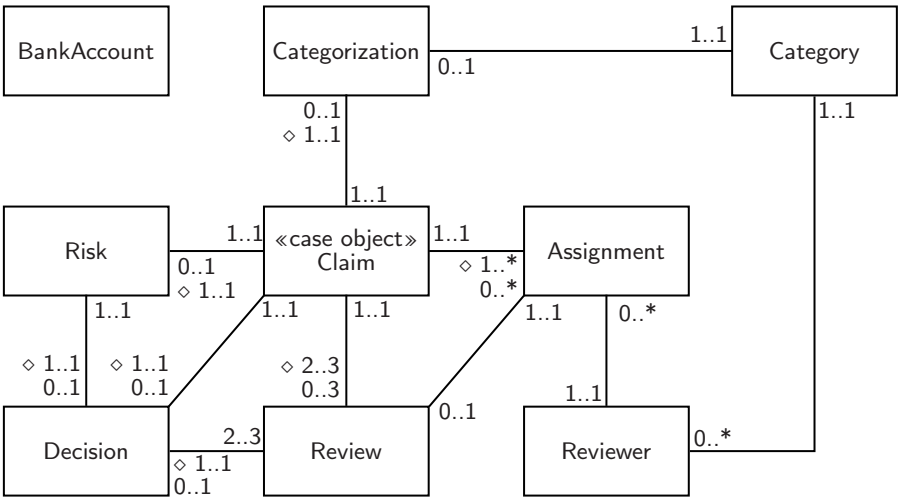


Figure 7.5: The domain model of the insurance example including the *Category*, *Categorization*, and *BankAccount* classes.

7.4.1 Attribute-Based Correlation

To correlate means finding or defining a relationship—in *Wickr*, between cases and cross-case data objects. We aim to capture dependencies among cases that influence the case behavior. The refined behavioral model can then be used for analysis, verification, and execution of multiple connected cases. However, correlation occurs at runtime, which limits our design-time approach to a subset of models and applications.

Computer science has many examples for correlation: Relational databases correlate entities through foreign keys. In interorganizational business processes, messages are correlated to process instances through a correlation key [92, p. 72], such as a tracking number. This key-based correlation is a special kind of attribute-based correlation. Attributes, i.e., the foreign key in one tuple and the primary key in another, are compared to match two or more entities. Attribute-based correlation allows one-to-one, one-to-many, and even many-to-many relationships.

The insurance from our example has multiple bank accounts. Employees' salaries are deducted from one account and reimbursements

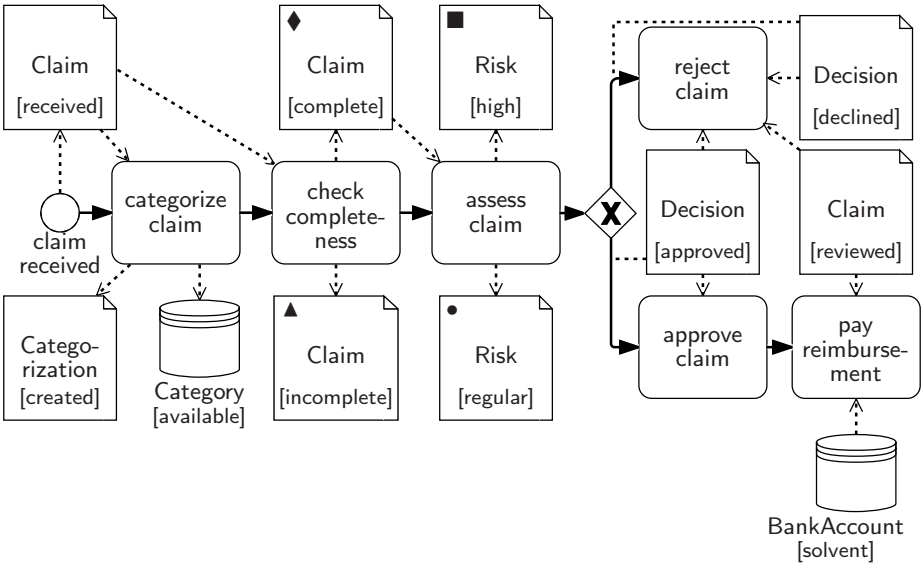


Figure 7.6: After a claim is received, it is categorized. Categories are represented by a cross-case data object. Furthermore, a bank account is required to pay the reimbursement.

from another one. However, our current formalization (see Figure 7.7a) does not make this distinction. Therefore, a process responsible for paying salaries may affect reimbursements and vice versa. Through explicit correlation, we want to connect cases that are related to each other, such as multiple reimbursement cases, and keep others separate.

In the Petri nets, attribute-based correlation can be implemented in different ways. We present implementations based on states, constants, mathematical relations, and attributes. We briefly discuss the potential applications and limitations of each approach.

States. Data object nodes abstract from data objects' individual attributes and use a finite number of states instead. We can use these states to limit the cross-case data objects that can be accessed by cases of a specific model (see Figure 7.7b). In the current version, reimbursement requires a *solvent* bank account. We can extend the model with additional states, e.g., to encode the purpose of the account. A state *solvent_reimbursement* may be used for all bank accounts that are solvent and that may be used for reimbursements. This mechanism requires no change in the formalization nor additional knowledge. However, it is limited to a fixed number of correlation keys and enlarges the object behaviors.

Constants. If we know the identities of cross-case data objects at design-time, we can use them directly in the case model's formalization. In the Petri net transitions, we can add additional guards, that restrict access to the predefined cross-case data objects. When, for example, "pay reimbursement" accesses a bank account, the

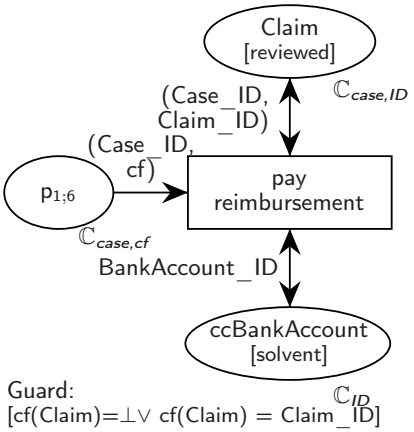
respective transition checks that it is a predefined reimbursement account (see Figure 7.7c). This approach is independent of the state of objects. On the downside, it requires instance-specific information at design-time. It is static since only a predefined set of objects can be correlated. And it correlates all cases of one model to the same set of cross-case objects.

Mathematical Relations. On an instance level, multiple cases of the same model and multiple objects of the same class may exist. Furthermore, cases may produce new cross-case data objects. In the end, correlation may be defined by a relationship that cannot be expressed by states and constants. Instead, we can use mathematical relations between case identities and object identities. We can enforce that the integer parts of both identities are equal to implement a 1-to-1 relationship. We can balance access to two objects by having cases whose identities' integer part is even access one object and cases with an odd integer part access another one. In the formalization, respective relationships can be implemented as a guard condition. In Figure 7.7d all accounts for reimbursements have an identity whose integer part is even. This method supports scenarios in which new objects may be created. It is useful to model complex relationships among cases, yet it is unlikely to reflect the real world accurately. It may be useful for analysis, verification, and simulation to investigate the behavior of multiple connected cases, but it is unsuited for implementation and execution.

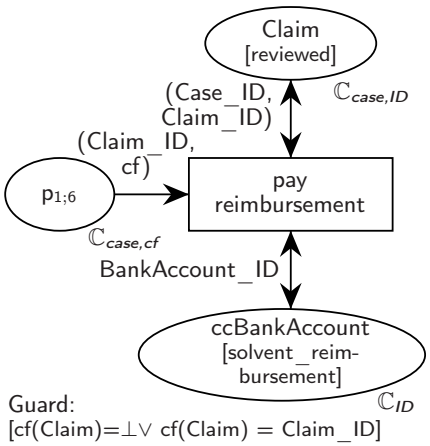
Attributes. Finally, we can incorporate attributes explicitly in our formalization. In the real case, each class defines a set of attributes and each respective object has a value for this attribute. Some of these attributes are used during correlation. However, this method requires changes to the formalization: Class-specific colorsets including attributes must be added. To be used for analysis and verification, appropriate values must be generated within the Petri net. Furthermore, if values are generated probabilistically, the state space may not cover all possible variants, which limits its value for analysis and verification. Nevertheless, this method is used in implemented case models in which attributes can be set by knowledge workers and/or services. In Figure 7.7e, each bank account has an attribute *purpose*. When a new account is added, the worker assigns a respective purpose. When a payment is made, the process searches for a bank account with the right purpose, i.e., reimbursement.

7.4.2 Links and Cross-Case Data

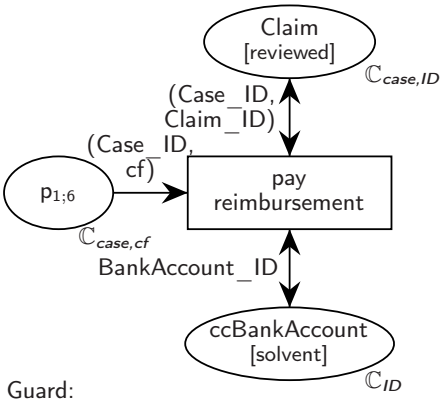
A cross-case data object can be linked to other cross-case data objects or to local data objects. The extended domain model (Figure 7.5) includes an association between *Categorization* and *Category* as well as between



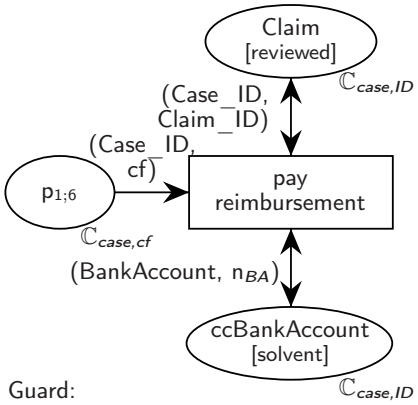
(a) No correlation.



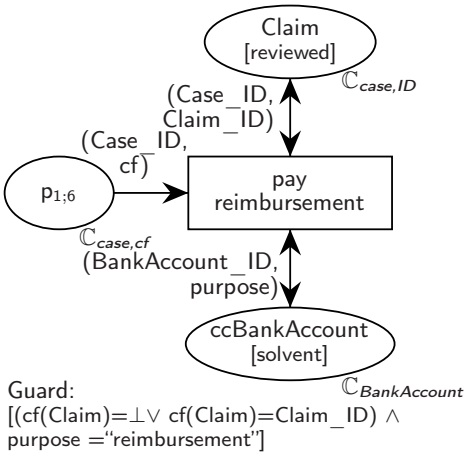
(b) State-based correlation.



(c) Correlation using constants.



(d) Correlation using mathematical relations.



(e) Correlation using an explicit attribute.

Figure 7.7: Cross-case data objects can be correlated to cases. The correlation can, thereby, be constrained using the state, constants, math, or explicit attributes.

Reviewer and *Category*. Links of cross-case data objects are subject to multiplicity constraints and affect cases: Links among cross-case data objects define how they can be used together. A link between a cross-case object and a local one can assert consistent correlation throughout a case. Its multiplicity constraints furthermore limit the objects and the number of objects that can be correlated to a single case.

Links limit the combinations of data objects that are accessed by conditions and activities. Although cross-case data objects may have been linked by a different case, every combination of such objects accessed must meet the constraints of the domain model. This also means objects that violate case-model-specific multiplicity constraints cannot be accessed by respective cases. Furthermore, cross-case objects may also be linked to local objects. This may limit the choice of other objects in the future.

In the example, each claim eventually links a categorization which connects the claim to a category. Furthermore, reviewers have a specialization, which is also a category. The reviewers that are assigned to a claim must have a respective specialization. To assert this, we have to add the category and categorization to the input sets of "request external review" and "assign internal reviewer" (see for example Figure 7.8). The semantics of links enforce that all reviewers have the required specialization.

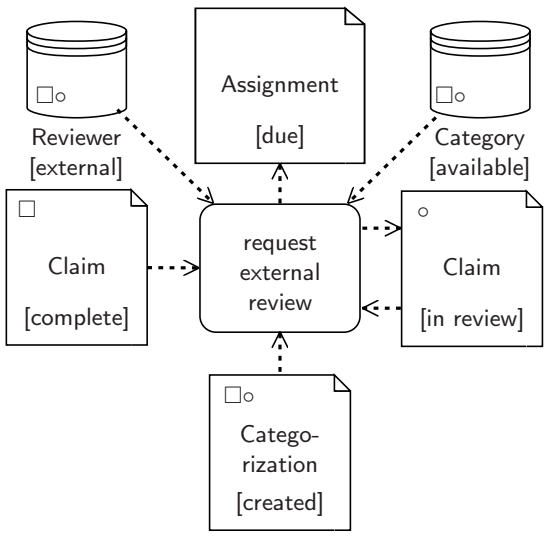


Figure 7.8: Adapted fragment for assigning external reviewers regarding their specialization and the case's categorization.

In the formalization, links between cross-case data objects must be added to the respective registry. However, the registry of links is not case-specific. Thus, all cases can access the information about links. Just as cross-case data objects, links may exist before a case.

In the insurance example, we may add categories and reviewers as well as links for the specialization to the initial marking. Transition "categorize claim" consumes a token for the category and links it to the newly created categorization, which is linked to the claim. When

transition "request external reviewer" fires (Figure 7.9), it consumes—among others—tokens for the claim, its categorization, its category, and a reviewer. Using the registry of links, it is asserted that the categorization is linked to the claim and to the category, and that the reviewer is linked to the category as well.

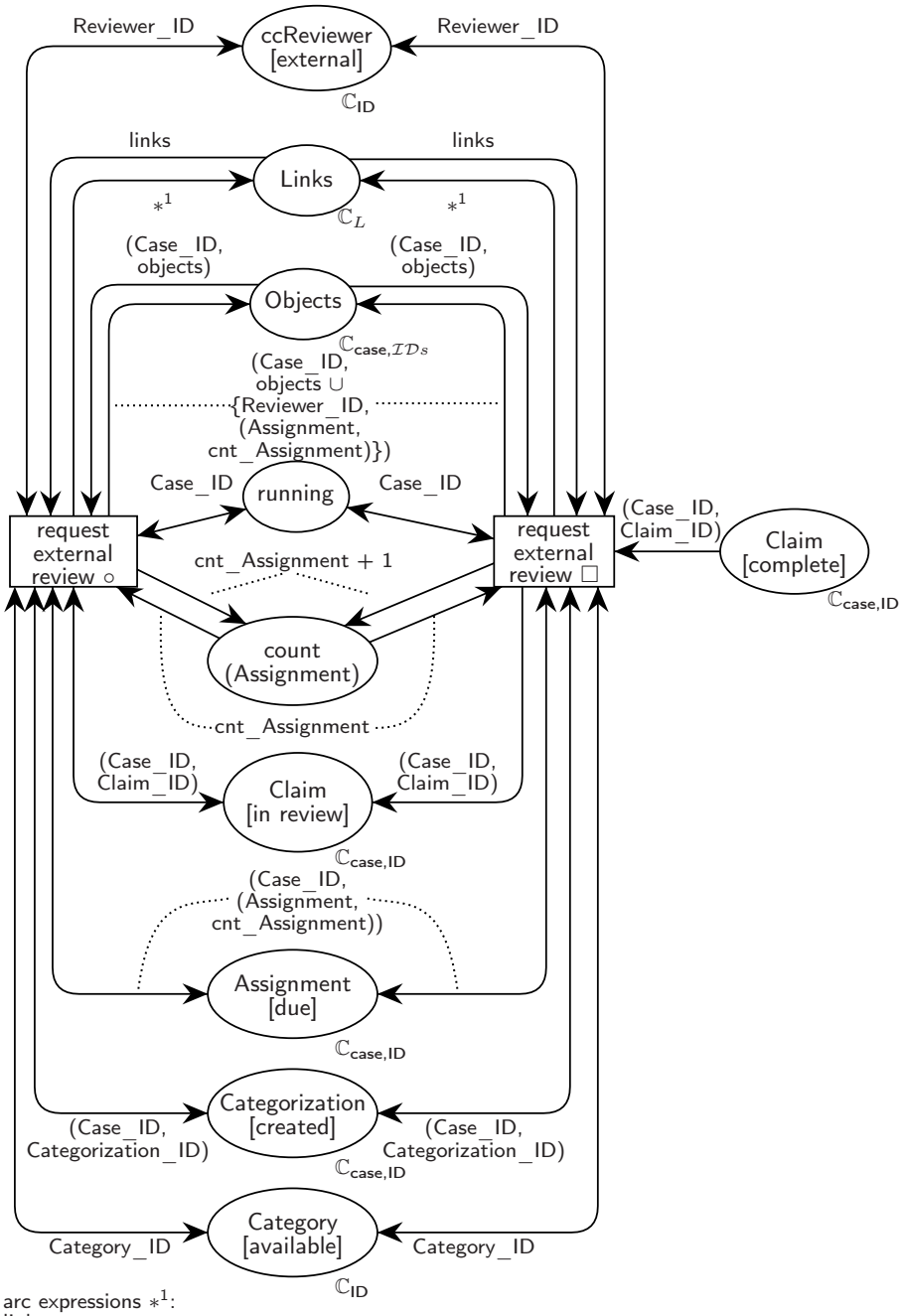
Finally, multiplicity constraints in combination with cross-case data objects can serve multiple purposes: We can limit the number of cross-case data objects per case, the number of cases per cross-case data object, and transitively the number of cases in regard to the number of cases of a different model. In the example, a case has at most one category.

7.5 The Case and Cross-Case Data

Cross-case data objects allow communication among cases. One case can influence another case directly by changing the state of a cross-case data object, by linking it to other objects, by allocating it, or by publishing it. Since the subject and situation of a case is described by data (see Definition 30, p. 70), they are directly affected by such changes. Consequently, enabled actions are affected as well because they may require cross-case data.

This may have severe impact on the case behavior. A case can deadlock if it requires a cross-case data object that is allocated by another case, or if the object's state changed. We focus on describing the dependencies among cases and not on resolving resulting problems. Solutions may include transactions, which have been intensely researched for highly structured process [10, 26, 36, 49]. However, transactions for *Wickr* are out of scope for this thesis.

Yet, our semantics for cross-case data objects may raise the awareness for potential risks and enables verification. We may verify that every allocated object will eventually be published, or that the back-and-forth between cases does not end in a deadlock. However, manual verification and interpretation of case models can be hard. In the next chapter, we present tools that support knowledge workers at design-time and runtime and that use *Wickr's* formal execution semantics.



arc expressions $*^1$:
links ∪
{{Reviewer_ID, (Assignment, cnt_Assignment)},
{Claim_ID, (Assignment, cnt_Assignment)}}
guard of transition "request external review ○:"
[{Claim_ID, Categorization_ID} ∈ links ∧
{Category_ID, Categorization_ID} ∈ links ∧
{Category_ID, Reviewer_ID} ∈ links]
guard of transition "request external review □:"
[|{$n_r \in \mathbb{N}_0$|{(Risk, n_r), Claim_ID} ∈ links}| ≥ 1∧
{Claim_ID, Categorization_ID} ∈ links ∧
{Category_ID, Categorization_ID} ∈ links ∧
{Category_ID, Reviewer_ID} ∈ links]

Figure 7.9: Petri net formalization of fragment three as depicted in Figure 7.8.

8 Technical Evaluation

BPM can be challenging: Processes need to be designed, implemented, executed, and constantly improved. However, a single enterprise has many processes, which can be complex. BPM software supports organizations in keeping their processes under control. Software may be used in every phase of the BPM lifecycle (cf. Figure 1.1, p. 2).

During design, tools for modeling, analyzing, verifying, and validating process models are used. Furthermore, models may be stored in a searchable process model repository [124, 149].

Process Modeling Tools are used to create visual models of business processes, e.g., using BPMN [45]. Many process modeling tools support syntax checks and integrate a *process repository*.

Process Repositories store a collection of process models [73]. They commonly support search, versioning, and a release process.

Simulators take a process model and simulate the process [63]. Simulation can be used for what-if-analysis, improving the process understanding, and validation [63, 134].

Model Checkers take a model and generate the full state space [17, 44]. The state space can be searched, i.e., to verify behavioral properties. These properties can be generic, such as soundness [64], or domain specific, such as compliance rules [52].

During implementation, processes need to be configured and implemented. In some cases, this is done within *process modeling tools*. In other cases, generic or task-specific tools are used.

At runtime, processes are executed. Tasks need to be orchestrated; performance indicators may be tracked; and additional constraints may be monitored [149].

Process/Case Engines take a process model as input and orchestrate the process accordingly. They receive user input and interact with services. Using a process engine, a process is executed in compliance with the process model.

Monitoring Tools can be used to monitor business rules not included in the process model and to keep track of performance indicators.

In the analysis phase, *process mining* tools can be used to provide insights into the executed process [104, 128]. Common tasks include the following:

Process Discovery derives a process model from an event log. The model describes the process as it has been executed, which may include deviations from the designed process.

Process Conformance detects deviations between a designed and a logged process.

Quantitiatve Process Analysis investigates past process instances to make quantitative statements. It can be used to detect bottlenecks or to analyze temporal constraints.

Of course, tools play an important role in case management as well. They can help to cope with the hidden dependencies and the complexity of the case models [12]. For the fCM approach, a set of tools has been developed.

Gryphon is a modeling tool for fCM.[1] Fragments and object lifecycles can be modeled visually, and classes textually. It has a process repository that can store a collection of case models.

Chimera is an fCM execution engine.[2] It provides a user interface to receive inputs, can call web-services via REST, and receives external events through the event processing engine *Unicorn*.[3] Models can be deployed directly from within *Gryphon*.

LoLA Integration allows model checking for fCM models.[4] An extension for Gryphon and Chimera translates a set of fragments into a Petri net. Users can choose between two initial markings: the initial state of the case or the state of a running case. The extension uses the model checker LoLA [25] to verify the fragments against custom temporal logic formulas.

The tooling for fCM is feature-rich but not designed for changes in the execution semantics. To evaluate *Wickr*, we developed a new set of tools that implement the presented semantics.

8.1 Architectural Overview

Wickr's tooling supports users during design-time and runtime. The tools show *Wickr's* technical feasibility. As proof-of-concepts, they are lightweight and adaptable but do not focus on the end-user experience and scalability. An overview of the architecture is shown in Figure 8.1. We developed the following tools especially for *Wickr*:

The Compiler translates *Wickr* case models to colored Petri nets that are compatible with CPNTools[5] [56]. Therefore, CPNTools capabilities for verification, analysis, and simulation can be used for *Wickr* as well. Furthermore, the compiler performs a syntax check on the case model and provides respective feedback. Leon Bein contributed significantly to the implementation of the compiler in his role as a student assistant.

[1] `https://github.com/bptlab/Gryphon` (2021/11/06)
[2] `https://github.com/bptlab/Chimera` (2021/11/06)
[3] `https://github.com/bptlab/unicorn` (2021/11/06)
[4] `https://github.com/bptlab/gryphon/tree/compliance` (2021/11/06)
[5] `https://cpntools.org` (2021/11/06)

The Engine uses CPNTools as a backend and provides a *Wickr*-specific frontend. Furthermore, it parses the domain model to store attribute level information for data objects.

The Goal Modeling and Planning component allows modeling knowledge workers' goals and translating them into a planning problem (query). The tool was created by Anjo Seidel in the context of a master's seminar and is, as of this writing, extended for his master thesis.

The tools have been presented in previous publications [166, 167]. In this chapter, we look at the different prototypes. First, we talk about the tools for creating *Wickr* case models. Next, we talk about the compiler that verifies a case model structurally and generates a colored Petri net, which can be used to verify the case behavior using CPNTools. Afterwards, runtime components are presented: the engine and a component for goal modeling and planning.

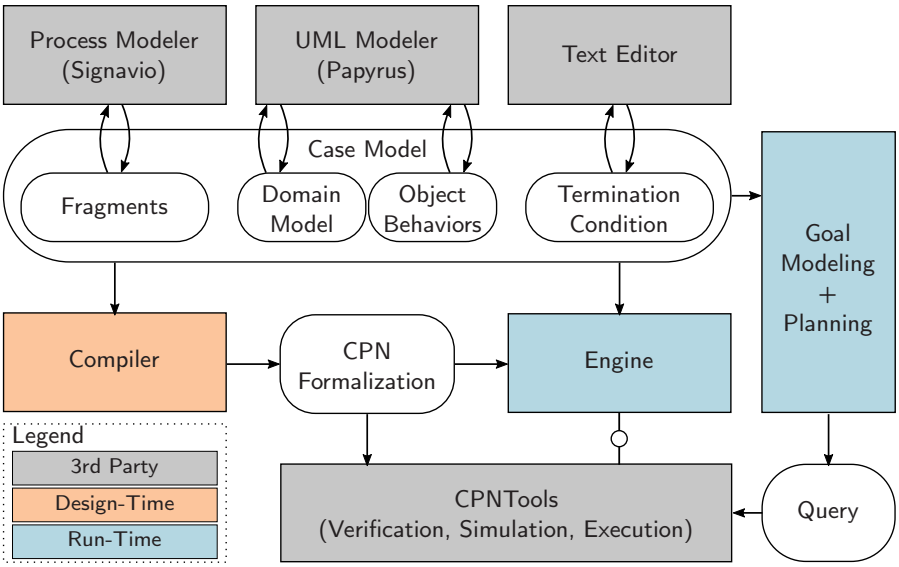


Figure 8.1: FMC [35] block diagram of the Wickr architecture combining design-time, runtime, and 3rd-party components.

8.2 Modeling

At design-time, by collaborating domain and modeling experts, case models that describe the intended behavior are created. A case model includes fragments, a domain model, state transitions systems, and the termination condition. These parts depend on each other, and only their combination describes the case in detail. The way in which the parts come together makes *Wickr* unique, yet each component isolated resembles known models and concepts. Therefore, existing tools for modeling processes and data can be reused.

Wickr's fragments are made of BPMN elements—events, activities, gateways, and control flow as well as data object nods and input and output sets. Hence, BPMN modeling tools supporting these concepts can be used to model fragments. For our prototypes and examples, we use Signavio[6] and model all fragments of a case model in a single BPMN. Of course, elements unsupported by *Wickr* must not be used. Furthermore, Signavio may report syntactical errors and violations of BPMN guidelines, although the model is correct for *Wickr*.

The domain model uses elements from UML class diagrams, and extends them with goal multiplicity constraints and a new stereotype for the case class. The general structure, classes connected by associations, can be created with any respective modeling tool. If stereotypes are supported, the case class can be marked. Furthermore, we decided to add goal multiplicity constraints as comments of associations. While this may downplay their role, it allows us to use any UML compliant modeling tool. In our examples, we use *Papyrus*.[7] While not made explicit in this thesis, classes can have typed attributes. However, other concepts, such as inheritance, packages, or specialized associations are not supported by *Wickr*.

Object behaviors are modeled by class-specific state transition systems. We use non-hierarchical UML state machines [123] to model the object behavior, but only states and transitions are relevant. Other elements and details, such as triggers, effects, initial and final states, are ignored by *Wickr*. For our examples, we use *Papyrus*.

Finally, the termination condition is part of the model but not visualized. As mentioned, it is a logic formula in disjunctive normal form that can be modeled as a set of sets of data object nodes. In our prototypes and examples, we choose a textual representation: The termination condition is modeled by a JSONArray.[8] Product terms are modeled by arrays containing JSON objects that describe data object nodes.

By reusing existing modeling languages, we are equipped with a set of sophisticated and mature modeling tools. Yet, the tools do not support the integration of models as required by *Wickr*. Therefore, we rely on simple name matching: The name of data object nodes must equal the name of classes in the domain model. The state of data object nodes must equal a state in the respective state transition system. While this connects the models, it does not enforce correct models—parts may not fit to each other, e.g., activities may model state transitions not supported by the object behavior. In the next section, we present a compiler that does not only produce a colored Petri net but also detects structural errors.

[6] `https://www.academic.signavio.com/` (2021/11/06)
[7] `https://www.eclipse.org/papyrus/` (2021/11/06)
[8] `https://json.org` (2021/11/06)

8.3 Compilation and Verification

In this thesis, we specified *Wickr's* semantics by translating case models to colored Petri nets. Such formal semantics are important for verification and execution. CPNTools is a tool for modeling, analyzing, verifying, and simulating colored Petri nets. By translating case models to compatible nets, we can use CPNTools for *Wickr*.

Since, translating models manually is tedious and error-prone, our tool collection includes a compiler[9] (Figure 8.2), which automatically creates a colored Petri net for a case model. It reads a set of fragments contained in a single BPMN, a UML class diagram describing the domain model, and a JSON file containing the termination condition. The object behavior is not explicitly provided, instead we derive it from the set of fragments: By looking at the input-output-set combinations, we see possible state transitions. This, however, implies that all input-output-set combinations that are valid according to the domain model are considered to be valid according to the object behaviors. Thereby, we lose one option of refining the case model's behavior—the compiler is not fully feature complete.

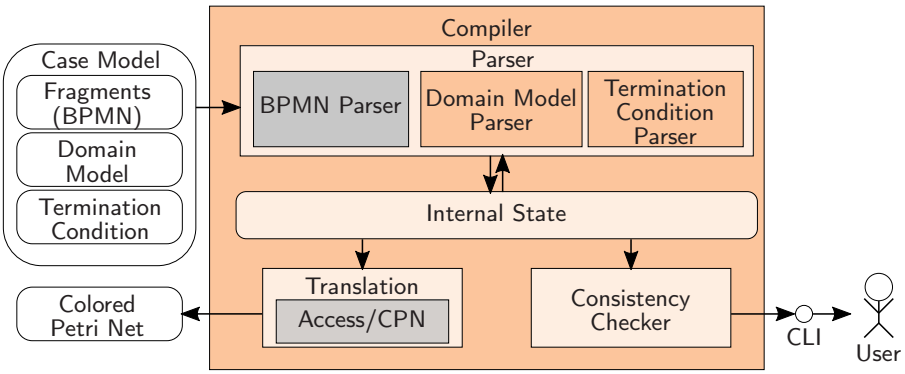


Figure 8.2: FMC [35] block diagram of the compiler's architecture—third party libraries are gray.

First, the compiler parses the model. Therefore, we use Camunda's BPMN parser[10] for fragments, a custom parser for the domain model, and the json-simple library[11] for the termination condition. Once the model has been parsed, the compiler checks structural consistency criteria. It reports violations to the user using the command line. Finally, the compiler use Access/CPN[12] to create a CPNTools compatible colored Petri net, which is written to a file.

[9] `https://github.com/bptlab/fcm2cpn/` (2021/11/06)
[10] `https://github.com/camunda/camunda-bpmn-model` (2021/11/06)
[11] `https://cliftonlabs.github.io/json-simple/` (2021/11/06)
[12] `http://cpntools.org/access-cpn/` (2021/11/06)

8.3.1 Structural Verification

The compiler checks that the case model is well-formed. Therefore, it investigates the fragments, the domain model, and the termination condition in isolation and in combination. We call this *structural verification*. For this purpose, the compiler has a dedicated component, the consistency checker. It reports violations via the command line to the user, who can resolve them manually.

To be a well-formed case model, the respective domain model, fragments, and termination condition must be well-formed and the consistency criteria—structural satisfiability, object behavior conformance, contextual object creation, and contextual batch processing—must be satisfied. However, since no object behavior is provided, object behavior conformance is assumed for all input-output-set combinations.

Regarding the domain model, the following questions must be answered. Is there at most one association between two classes? Are all associations existential one-to-one or one-to-many associations? Are multiplicity constraints consistent, i.e., are goal multiplicity constraints refinements of their global counterpart? If all questions are answered positively, the domain model is well-formed.

Fragments must be well-formed as well. The consistency checker investigates the control flow to verify that there is a clear entry point, that there is no uncontrolled flow, that it is acyclic, and that each gateway is either a split or a merge. Conditions, input sets, and output sets are verified as well. In the end, all but one of the properties defined in Definition 23 (p. 63) must be satisfied: Disconnected control flow is not considered to be an error. Since all fragments are contained in a single BPMN file, disconnected parts are treated as separate fragments.

Also, the termination condition must be well-formed. Each of its products terms must not have two elements with the same class.

To check *structural satisfiability*, the consistency checker extracts all local data object nodes used in the fragment or termination condition. It categorizes them into requirements (i.e., nodes used in conditions and input sets) and outputs (i.e., nodes used in output sets). Structural satisfiability requires the requirements to be a subset of the outputs.

Contextual object creation requires that the requirements posed by existential associations are satisfied during object creation. Therefore, verification investigates the input-output-set combinations in the fragments with regard to the associations and multiplicity constraints in the domain model. Not every combination must be valid, but for every input set must exist an output set and vice versa so that their combination is valid.

When data objects are processed in batches, *contextual batch processing* implies that there is one and only one reference object in the same input set. The consistency checker verifies whether this property holds by investigating input sets and product terms containing batches and the domain model to determine candidates for reference objects.

If no errors are detected by the consistency checker, the *Wickr* case model is well-formed. This does not imply that modeled behavior is

correct. There can be, among other things, deadlocks and livelocks. The structural verification has no global view on the behavior. This gap is filled by behavioral verification.

8.3.2 Behavioral Verification

To fully verify the process model, the full behavior—all possible states and state transitions—must be considered. This allows us, for example, to verify that the case goal can be reached or that constraints that are not explicitly modeled are not violated. Therefore, we have to derive all states and state transitions from the case model.

For this reason, we employ model checking [17, 44]. During model checking, the state space of a behavioral model is calculated to prove formal properties. The model is assumed to be complete: User choices and unspecified conditions are treated non-deterministically. Runtime changes to the model are not considered/supported. Properties can be formalized using temporal logics, such as Linear Temporal Logic (LTL) [5] and Computational Tree Logic (CTL) [6].

Figure 8.3 depicts the model checking process for *Wickr*. Case models are formalized as Petri nets and properties as temporal logic formulas. The model checker takes both formalizations, calculates the case model's state space, before verifying the properties. The result of the verification is then communicated to the user. For details on model checking, we refer to respective literature [17, 44].

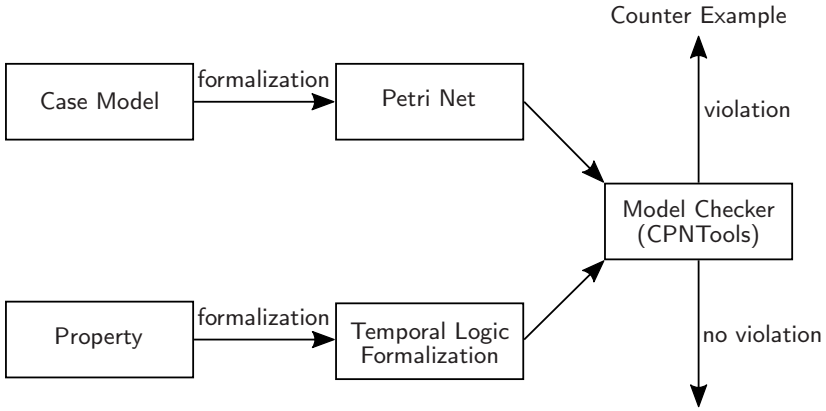


Figure 8.3: Schematic representation of the model checking method (cf. [113]).

State-Bounded Case Models

In the worst case, the model checker has to investigate the full state space. This implies that the state space has to be finite or that it has a finite approximation [21]. However, *Wickr* models have, in general, an infinite state space.[13] There are multiple potential sources of infinity: infinitely

[13] In fact, *Wickr* is Turing-complete, so model checking is, in general, undecidable. A proof is provided in Appendix A.

many data objects, infinitely many links, and infinitely many concurrent fragment instances (cf. [81, 114]). If the number of data objects, links, and concurrent fragment instances is bounded, the case model is *state-bounded* [81, 114]. We propose to preprocess case models to guarantee that they are state-bounded, i.e., that their Petri net formalization has a finite reachability graph.

To limit cases to a finite number of objects and links, we first require that all classes are associated directly or indirectly to the case class (cf. *navigationality* in [114]). Second, we forbid multiplicity constraints with unlimited upper bounds (cf. *case-width-boundedness* in [114]). Since there is exactly one case object for each case, and all objects are linked directly or indirectly to this case, and each object has a limited number of links, the case has a limited number of objects. However, the upper bounds of multiplicity constraints should not be set arbitrarily but chosen carefully by a domain expert.

Figure 8.4 depicts the updated domain model of the insurance example. We set the upper limit for assignments per reviewer to one because each reviewer should review the same claim at most once. Furthermore, there are at most ten assignments for each claim.

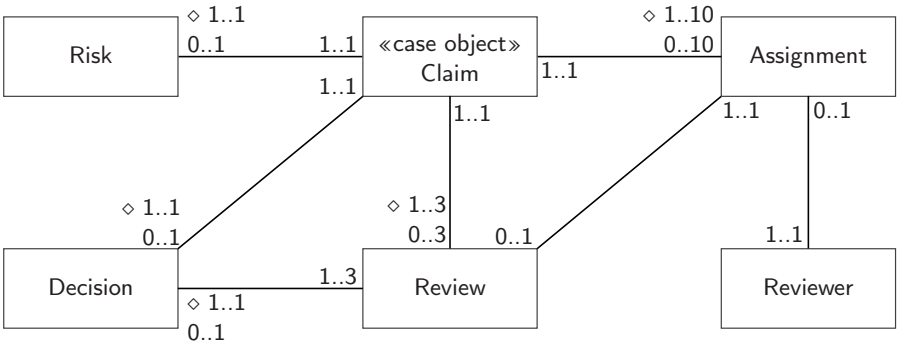


Figure 8.4: The insurance domain model without unbounded multiplicity constraints.

We must also prevent infinitely many concurrent fragment instances (cf. *case-width-boundedness* in [114]). Therefore, we augment fragments with a respective upper bound if necessary. This bound is again domain-specific. In the formalization, we add a place for each fragment. It contains as many tokens as concurrent instances are supported. The transitions starting a new fragment instance consume a token from the respective place. The transitions terminating a fragment instance produce a token into the place. A similar workaround has been used for checking fCM models [144].

The example does not need these artificial constraints. The three fragments with more than one activity cannot be executed arbitrarily often in parallel. Therefore, bounds are already part of the model: Fragment one is executed once; fragment two can be executed arbitrary often but only sequentially; and fragment six is limited to one instance as well. The formalization of the case model for the insurance example

with the adapted domain model (Figure 8.4) can be used for model checking without further adjustments.

Behavioral Properties of Case Models

Given a colored Petri net formalization that has been prepared for model checking, we can verify behavioral properties—generic ones and domain-specific ones. However, meaningful properties must be defined and selected. Soundness and its variants are properties verified on workflow nets, a subgroup of Petri nets that have a dedicated initial place and a dedicated final place [64]. To satisfy soundness, it must hold that each transition participates in at least one trace; that all traces end in the final state, which has a single token that is placed in the final place. The last requirement makes data-aware processes not sound because tokens representing data may remain in the net. However, we can define a variant of soundness that allows additional tokens on places for data, i.e., places for data object nodes and places for the registries (cf. [80, 83]). Initial and final marking may be valid even if they contain tokens representing data.

Yet, the Petri nets of case modes are often not sound. Due to the nature of knowledge-intensive processes, the models are purposely left underspecified. At runtime, knowledge-workers make decisions to steer the case towards a successful end. However, the model may allow states from which the goal cannot be reached. *Relaxed soundness* requires that every transition participates in one trace leading to a state with only a token in the final place [64]. A data-aware variant may furthermore allow tokens representing data. This is meaningful for many case models: It requires that each activity and each valid input-output-set combination may contribute to the goal and that the goal can be reached. At the same time, the case model can allow traces that do not result in the goal (underspecification).

For some case models, relaxed soundness may still be too strict. In general, a case can terminate even when fragment instances have not been completed. In this case, the final state may include control flow tokens, and relaxed soundness is not satisfied. This is not necessarily wrong. Therefore, we define *lazy relaxed soundness* (Definition 31).

Definition 31 (Lazy Relaxed Soundness)**.** The colored Petri net formalization of a case model is called *lazy relaxed sound*, iff each transition participates in a trace that ends in a marking with exactly *one* token in place *terminated*. ◇

Before model checking, properties must be formalized. Therefore, temporal logics, namely LTL and CTL, are commonly used in model checking and supported by Petri net model checkers, such as CPN-Tools [56] and LoLA [25]. Temporal logics have temporal operators and in the case of CTL path quantifiers. They can be used to make statements about sequences of states and state transitions.

For an example, we formalize the *lazy relaxed soundness* criteria. To satisfy *lazy relaxed soundness*, each transition must participate in a trace

whose final state has a token in place *terminated*. Since tokens in *terminated* cannot be removed, it is enough to find a trace in which such a state is reached after the transition has fired. Formally, for each transition t, the colored Petri net's reachability graph $(M, \xrightarrow{M})$ must have an arc $(m, (t, \mathit{binding}), m') \in \xrightarrow{M}$ and a path leading from m' to a state m_t so that $|m_t(\text{terminated})| = 1$. CPNTools allows us to verify a Petri net against temporal logic rules that are evaluated on states and transitions in the reachability graph. To check *lazy relaxed soundness*, we must verify the following CTL property for each transition t:

$$\mathbf{EF}(\mathit{fired}(t) \wedge \mathbf{EF}(\#\mathit{terminated} = 1))$$

Here, **E** refers to the existential path quantifier and **F** to the temporal operator *finally*. The formula reads "There exists a path on which we eventually reach a state by firing t, and from this state, we can reach another state with a token in state terminated." CPNTools ASK-CTL extension can evaluate temporal logic formulas on markings (states) as well as binding elements (state transitions). Therefore, we can investigate the transition that leads to a state (i.e., $\mathit{fired}(t)$).

Temporal logics can also be used to express domain-specific properties, i.e., compliance rules [52]. These rules may refer to and relate specific activities and data objects in the case model. In compliance checking, a model is verified against rules originating in law, guidelines, and best practices that are not explicitly included in the model.

Finally, we can also use behavioral verification to investigate the relationship between the domain model and the case behavior. A case does not necessarily cover the multiplicity constraints of the domain model fully. The state space contains lower and upper bounds for all association ends. If these multiplicity constraints equal the global constraints of the domain model, the case model satisfies *multiplicity coverage*.

With this approach, we can verify properties for the insurance example. The example is *lazy relaxed sound*, but it contains a deadlock. If all assignments are canceled, the case cannot continue as no reviews, which are required for a decision, can be created. However, in the original model (not adapted for model checking), it is always possible to create another assignment and the deadlock does not exist. Limiting the case to a finite number of objects is therefore not property preserving.

The adapted case model covers all multiplicity constraints if there are at least nine external and one internal reviewer in the initial state of the case. Furthermore, we can verify compliance rules such as the following: If the risk is high, multiple reviews are required. This can be expressed by the LTL formula:

$$\mathbf{G}(\text{risk}[\text{high}] \rightarrow \mathbf{F}(\#\text{review} \geq 2))$$

The formula reads as "It globally holds if the risk is high, the case eventually reaches a state with two or more reviews." This property is not satisfied—it is possible to close the case with a single review albeit

a high risk. This knowledge can, for example, be used to improve the model.

So far, we investigated the behavior of a single case. Therefore, it can be necessary to add cross-case data objects to the initial marking. However, the formalization also enables the behavioral verification of multiple cases, as long as the number of cases is finite. We can verify the same properties adapted for multiple cases and considering their interplay. One case may produce a data object required by another. Or there is a back and forth between two cases. In both scenarios, one isolated case model may not satisfy properties that are satisfied by multiple case models viewed together.

Experiments

Model checking explores the state space of a behavioral model and is computationally expensive. In fact, the state space grows exponentially with the degree of concurrency. This can be a problem for case models that describe flexible processes.

We calculated the state space for various variants of the insurance example (see Table 8.1). The data-aware BPMN (Figure 2.5) is highly structured, and the state space is small: 76 states and 82 transitions. The calculation took CPNTools three seconds.[14] If we consider the classical Petri net formalization of the *Wickr* model, limited to ten Assignments and three reviews, the state space is more than two orders of magnitude larger: 8,020 states and 14,927 transitions. The calculation of the state space takes eight seconds. We also tested the colored Petri net version, with a varying number of external reviewers: Three external reviewers (and one internal) results in 25,644 states, which takes 82 seconds to calculate. CPNTools fails to calculate the state space for four external reviewers.

CPNTools can handle state spaces with 20,000–200,000 states and 50,000–200,0000 transitions [37]. Due to the high degree of flexibility, the insurance example exceeds these limits for some configurations. However, we can approximate the state space: If we consider a configuration with three external reviewers—we call them Susan, Peter, and Maria—, they can be assigned in an arbitrary order. By prioritizing the order—e.g., first Susan will be assigned, then Peter, and finally Maria—we reduce the size of the state space drastically. The optimized version with three external reviewers has 894 states and 1,273 transitions. It takes five seconds to calculate. For the variant with four external reviewers, CPNTools calculates a state space with 1,544 states and 2,210 transitions in six seconds. Even if we have nine external reviewers and up to ten assignments, CPNtools can still calculate a state space with 7,494 states. Furthermore, this approximation does not affect the sequences of activities. We also checked lazy relaxed soundness for each of the models using CPNTools' ASK-CTL extension. Verification is the

[14] All performance measurements were taken on a computer with 12 GB RAM and an i7-7500U processor running CPNTools 4.0.1. The times are measured internally by CPNTools. The results are the average of three runs.

fastest for the BPMN's Petri net. It takes on average 17 ms. The slowest is the verification for the colored Petri net with three external reviewers that are not prioritized. It takes on average 1,459 ms. For each of the other nets, verification takes less than a second.

The size of the state space can hinder behavioral verification. By introducing additional assumptions, i.e., a limited number of objects or a prioritized correlation of cross-case data objects, a partial state space can be calculated. However, such assumptions must be carefully considered, as they may affect the results of analysis and verification: For example, if we want to determine whether all assignments are eventually in state *done* or in state *canceled*, limiting the number of assignments to less than four leads to a wrong result. If we assign one internal and at least three external reviewers, we can reach a state in which we have three reviews, but the assignment of the internal reviewer is still in state *due*. The internal assignment cannot be changed to *done* because three reviews already exist. Also, it cannot be *canceled* because only assignments of external reviewers can. Thus, the property is not satisfied, but if there were less than three external reviewers this situation would not occur, and the error would remain undetected.

However, the assumption that external reviewers are assigned in a fixed order does not limit the ordering of activities. Yet, it affects the sequence of actions, which depends on the exact combination of data objects that are read and written by each activity instance: Some combinations of data objects, such as the first assignment and the third external reviewer, cannot be linked albeit this being possible in the model without priorities. Therefore, the state space is an approximation of the original one. In conclusion, model checking is not always feasible, but careful manual tuning of the Petri can help for some case models.

Table 8.1: Experiments on calculating the state space of case models and checking lazy relaxed soundness (LRS).

Model	#States	#Transitions	Duration (s)	LRS (s)
BPMN PN	76	82	3	0.017
Wickr PN	8020	14927	8	0.417
Wickr CPN 3 ext. Reviewer	25644	38412	82	1.459
Wickr CPN 3 ext. Reviewer prioritized	894	1273	5	0.038
Wickr CPN 4 ext. Reviewer prioritized	1544	2210	6	0.063
Wickr CPN 9 ext. Reviewer prioritized	7494	10810	10	0.343

8.4 Execution

Engines can support knowledge workers in executing cases according to case models. However, the flexibility and complexity of knowledge-intensive processes can pose a challenge. We developed two prototypes to show the feasibility of *Wickr*: a case engine and a tool for goal modeling and semi-automated planning of cases.

A case engine tracks the state of cases and orchestrates activities based on a case model. Case engines determine the activities that can be executed. However, in decision-rich processes, such as knowledge-intensive ones, they cannot define which activities should be executed. For this purpose, goal modeling and planning can be used. During planning, actions are arranged into a sequence that conforms to the model and accomplishes a defined goal.

8.4.1 Case Execution Engine

An engine tracks the states of cases, invokes services, and is capable of receiving user inputs. Especially, users of complex processes can benefit from a case engine because they do not need to track the case state manually. Engines for knowledge-intensive processes have to satisfy special requirements, though [98, 147]. Knowledge workers consider versatile information to make decisions. Meanwhile, constraints of the model must not be violated. A balance between guidance and flexibility is required.

Our *Wickr* execution engine[15] uses CPNTools and the colored Petri net formalization of the case model. Instead of implementing the execution logic itself, the engine communicates with CPNTools to query the marking and fire transitions in the colored Petri net formalization of the case model. The architecture in Figure 8.5 shows that the case model's domain model and its colored Petri net formalization are provided as inputs to the engine. The engine communicates through *Access/CPN* with CPNTools and through a graphical user interface with the user. CPNTools tracks and updates the marking, and the engine stores additional information, e.g., attributes and their values.

The engine operates as follows: A user provides the colored Petri net formalization and the domain model of a case model. The engine parses the classes in the domain model and stores their structure. Furthermore, the engine passes the colored Petri net to CPNTools using Access/CPN. From CPNTools, the engine requests and receives the list of enabled transitions. It parses their labels to derive the corresponding control flow nodes. These are presented in a list of work items to the knowledge worker. When a knowledge worker selects a work item, it is presented with a set of available options: the input-output combinations in case of an activity, the outputs in case of an event, and the product terms in the case of the termination condition. When the user selects one of the options, the engine generates forms for the involved data objects (see Figure 8.6). By completing the action, the engine triggers the respective

[15] `https://github.com/bptlab/fCM-Engine` (2021/11/06)

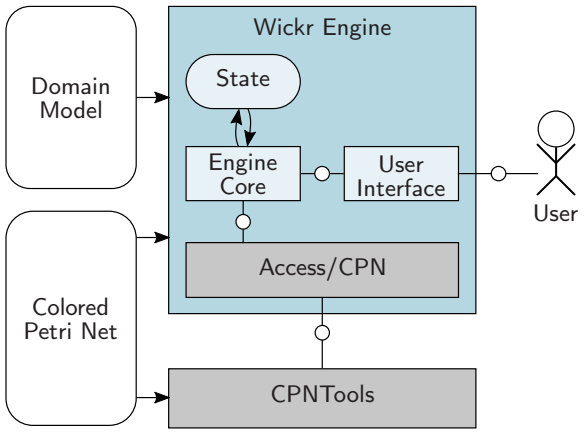


Figure 8.5: The execution engine operates on the colored Petri net representation. The abstract state is tracked via CPNTools, while attribute-level information are tracked in the engine.

binding element in CPNTools and stores the attribute values from the forms in the internal state. Afterwards, the view is reset, and the list of work items is updated.

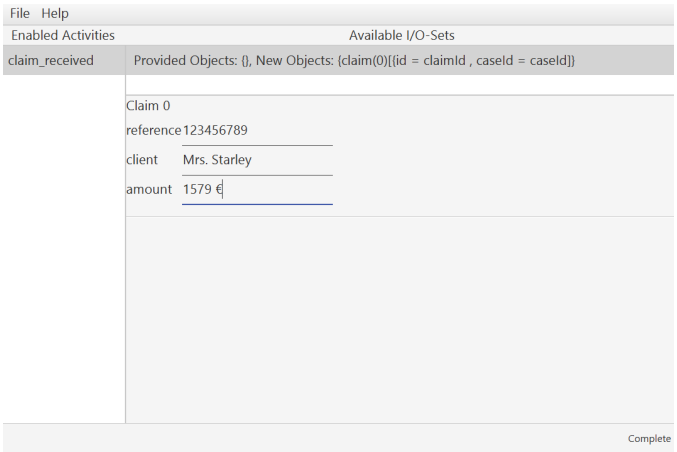


Figure 8.6: Screenshot of the execution engine. The work item list is shown on the left, the available input-output combinations on the top, and the forms for changing data objects in the center.

Since the Petri net models data objects only with an identity and an abstract state, detailed attribute-level information must be stored separately. The engine therefore has an internal state. It has a key-value store for each data object token in the Petri net. A key-value pair reflects a single attribute and its value. It can be updated by the knowledge worker through forms when activities are executed or start events are triggered. When a new object is created, a new key-value store will be associated with the novel identity.

The engine uses CPNTools as the backend that executes the case behavior. Therefore, the semantics presented in this thesis is sup-

ported straightforwardly. Smaller changes to the semantics can be implemented by adapting the Petri net without changing the engine at all. However, the lightweight implementation has its limits. The case state is not persisted and only one Petri net, which may contain multiple case models, is supported. Furthermore, the current implementation does not distinguish between read and write access to data objects, nor does it support multiple users. Finally, the case model cannot be adapted at runtime.

8.4.2 Goal Modeling and Planning

Typically, process engines guide users by listing the currently enabled actions. However, knowledge-intensive processes are multi-variant and require knowledge workers to choose one from many possible actions. Ideally, knowledge workers select an action that aligns with their goals, but in general, they have to make such decisions without support of the engine.

In this section, we present a method and a prototype for modeling goals and supporting knowledge-workers in planning actions accordingly. The work is based on a joint paper with Anjo Seidel [167].

According to the Miriam Webster dictionary, a goal is "the end toward which effort is directed."[16] Goals can be versatile. They can be functional or non-functional, concerned with actions, time, or data. They can be objective or subjective. However, we are only interested in goals supported by the case model. Therefore, the goals of interest may be concerned with objects, links, and actions.

Once a goal has been specified, decision support can be provided. Planning searches for a sequence of actions that accomplishes a goal. In *Wickr*, actions refer to activities and specific input-output combinations, i.e., a binding element in the Petri net. Simply put, planning searches the state space to find a path from the current state to a state satisfying the goal.

Goal Specification

In case management, a goal refers to the subject and the situation. Transferred to *Wickr*, a goal specification consists of constraints that describe a set of states: The goal has been accomplished if the case is in one of these states. The case state comprises data objects, links, and running fragment instances. More precisely, it is the marking of the corresponding colored Petri net. Given the state and the model, we can infer enabled actions (binding elements). This understanding leads to the following specification of goals.

Objects and Links. First, we revisit the representation of data within a case to specify respective goals. Data is encapsulated in a set of objects. In the formalism, objects have an identity, a state, and links to other

[16] `https://www.merriam-webster.com/dictionary/goal` (2021/11/06)

objects. A goal can, for example, make statements about particular objects, about all objects of a class, or about links among objects.

In a case of the insurance example, knowledge workers may want to achieve that

(g_1) All assignments are either done or canceled.

The goal makes a statement about all objects of the class *Assignment*. However, we can also require a specific object, for example

(g_2) A decision must exist.

We can also make statements about multiple objects of different classes.

(g_3) If the risk is regular, we require one review otherwise two or more.

And, we can define goals using links among objects:

(g_4) If the risk is high, only external reviewers are assigned.

Viewed formally, goals are defined as constraints for states. All goals that we consider are defined in the scope of a single case. Therefore, we can formalize the data state of a case by set O of objects and set L of links (see Definition 30, p. 70). Goals can be expressed using first-order logic with sets O and L.

Coming back to our previous examples, we can formalize them as follows:

$$
\begin{aligned}
g_1 \equiv & \forall o \in O : o.class = Assignment \Rightarrow \\
& (o.state = done \vee o.state = canceled) \\
g_2 \equiv & \exists o \in O : o.class = Decision \\
g_3 \equiv & \forall o \in O : (o.class = Risk \wedge o.state = regular \Rightarrow \\
& |\{o_r \in O | o_r.class = Review\}| = 1) \wedge \\
& (o.class = Risk \wedge o.state \neq regular \Rightarrow \\
& |\{o_r \in O | o_r.class = Review\}| \geq 2) \\
g_4 \equiv & \forall o_{hr}, o_a, o_r \in O : (o_{hr}.class = Risk \wedge o_{hr}.state = high \wedge \\
& o_a.class = Assignment \wedge o_r.class = Reviewer \wedge \{o_a, o_r\} \in L) \Rightarrow \\
& o_r.state = external
\end{aligned}
$$

These formulas can be evaluated in any given state. If it evaluates to true, the respective goal is satisfied.

Activities and I/O Operations. Goals drive the actions of knowledge workers. Like data objects, performing an action can be a goal. An action refers to the execution of an activity instance and involves concrete data objects that are read and written. With concrete data objects, we refer to objects with identities, states, and links and not to abstract data object nodes. Knowledge workers may aim to execute a specific activity for specific data objects.

Goals may be concerned with the state of activity instances/actions. From a marking in the colored Petri net, we can infer which activity

instances are *enabled*. However, we are unaware of other states—such as *running* and *terminated*—because the formalism abstracts from them.

We can specify goals concerned with actions for the insurance example. They may be concerned with the activity:

(g_5) Activity "pay reimbursement" is enabled.

Or they consider an activity instance including its input-output behavior. It is possible to make general statements about the inputs and outputs.

(g_6) Activity "decide on claim" is enabled and considers exactly three reviews.

It is also possible to make statements about specific objects that are read.

(g_7) Activity "cancel assignment" should be enabled for the reviewer with identity (Reviewer, 12).

Formally, each case has a set E containing enabled actions. Each action (O_r, n_a, O_w) consists of a set O_r of data objects that are read, an activity n_a, and a set O_w of data objects that are written. An action refers to a binding element of the colored Petri net, i.e., an arc in the reachability graph.

Based on the set E, we can formalize goals that are concerned with actions. The examples g_5–g_7 result in the following definitions:

$$
\begin{aligned}
g_5 \equiv & \exists e_{pr} \in E : e_{pr}.n_A = \textit{"pay reimbursement"} \\
g_6 \equiv & \exists e_{dc} \in E : e_{dc} = \textit{"decide on claim"} \wedge \\
& |\{o \in e_{dc}.O_r \mid o.\textit{class} = \textit{Review}\}| = 3 \\
g_7 \equiv & \exists e_{ca} \in E, \exists o \in e_{ca}.O_r : e_{ca}.n_A = \textit{"cancel assignment"} \wedge \\
& o.\textit{id} = (\textit{Reviewer}, 12)
\end{aligned}
$$

Composing Goals. Goals can be combined. A goal can involve both data and activities. We can, for example, make a statement about the links of objects read by an enabled activity. Thus, a single goal may access sets O, L, and E.

Consider the following goal for the insurance example:

(g_9) We want to execute "request external review" for a reviewer without assignments.

With our formalization of the case state, we can formulate g_9 formally.

$$
\begin{aligned}
g_9 \equiv & \exists e_{er} \in E, \exists o_r \in e_{er}.O_r, \forall o \in O : \\
& e_{er}.n_A = \textit{"request external review"} \\
& \wedge\, o_r.\textit{class} = \textit{Reviewer} \wedge o_r.\textit{state} = \textit{external} \wedge \\
& (o.class = \textit{Assignment} \Rightarrow \{o, o_r\} \notin L)
\end{aligned}
$$

Any goals that are specified refine the goal of the case. However, a case changes over time, but so far, goals are time-agnostic. Actual goals

of a knowledge worker may consist of multiple subgoals (also called objectives [148]) that must be achieved individually but not necessarily at the same time/in the same state. Therefore, we present composed goals that consist of a set of atomic goals (as specified before). The atomic parts of a composed goal can be ordered so that they must be achieved in the respective order.

Consider the insurance example. A knowledge worker may have three goals: First, three reviews are supposed to be created. Next, all incomplete assignments must be canceled, then "decide on paper" must be enabled. This can be expressed by the set $\{g_a, g_b, g_c\}$, where

$$\begin{aligned} g_a \equiv & |\{o \in O | o.\mathit{class} = \mathit{Review}\}| = 3 \\ g_b \equiv & \forall o_a \in O : o_a.\mathit{class} = \mathit{Assignment} \Rightarrow \\ & (o_a.\mathit{state} = \mathit{canceled} \vee o_a.\mathit{state} = \mathit{done}) \\ g_c \equiv & \exists e \in E : e.n_A = \text{“}\mathit{decide\ on\ paper}\text{”} \end{aligned}$$

The composed goal furthermore consists of a partial order $\leq_G$:

$$g_a <_G g_b <_G g_c$$

The composed goal is consequently defined as a tuple $G_{a;b;c}$:

$$G_{a;b;c} = (\{a, b, c\}, \leq_G)$$

Planning

Once knowledge workers have specified a goal, they can identify and take actions towards it. Finding a sequence of actions that realizes a goal is called planning, and the sequence is called a plan. Planning can be manual or automated [111].

Planning is common in knowledge work [31]. In *Wickr*, a plan is a sequence of actions. However, in knowledge-intensive processes, automated planning has limitations. Case models are often underspecified since the vast knowledge of experts cannot be modeled completely. Nevertheless, we present a method with which knowledge workers can still profit from automated planning.

To find a plan for a given goal, we must find a sequence of actions that implies a sequence of states satisfying the goal. Therefore, we can use the colored Petri net formalization of a case model that has been adapted for behavioral verification. The plan is a path in the reachability graph. The path starts in the current state and satisfies subgoals in the specified order until the composed goal has been accomplished. Since we are interested in both states and actions, the binding elements that lead from one state to another must be considered as well.

To find a plan, the state space has to be searched for respective paths. However, there can be multiple paths. Therefore, we propose a user-defined scoring function that assigns each plan a score. A user may prefer a plan with the least number of actions, or they may incorporate additional knowledge to score plans according to their duration. In any case, suboptimal plans are not discarded.

Instead of settling on one plan, we leave the knowledge workers in charge. All enabled actions are presented to the knowledge workers. If an action is part of a plan, this information is communicated together with the plan's score. If an action belongs to multiple plans, the highest score is shown. Thus, the knowledge workers can still decide freely. Yet additional information are provided, so knowledge workers know whether an action brings them closer to their goal.

Implementation

Under our supervision, Anjo Seidel developed a proof-of-concept for goal modeling and planning.[17] Goals can be specified via forms, and state space queries can be generated to support planning via CPNTools.

The case model's fragments and the domain model are provided as inputs. Both are parsed and stored internally. Through forms, the users interact with the application. They can specify goals (see Figure 8.7), view their definitions, and retrieve respective queries for CPNTools.

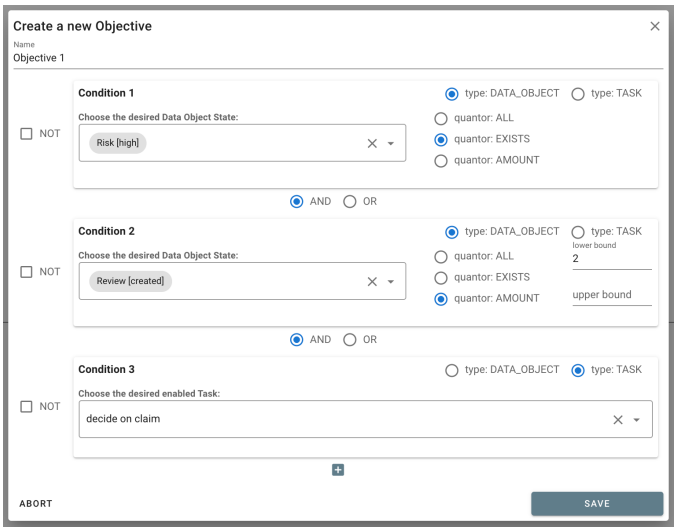


Figure 8.7: Screenshot of the form-based goal modeling.

Goals may involve activities and data objects, and they may include existential and universal quantifiers. However, the current implementation has some limitations: Links, data object identities, and input-output combinations for actions are not supported because this information depends on the actual case, but the implementation is not connected to the engine. Also, composed goals have not been implemented yet.

Based on the forms, state space queries that use CPNTools' ASK-CTL [11] extension are generated. Evaluated in CPNTools, the queries determine whether the goal can be accomplished starting in the current state. They do not return a path. However, the query can be evaluated for each possible successor state. If it returns true for a specific successor, the actions leading to this successor state are compliant with the goal. Although scoring functions have not been implemented, the result of

[17] `https://github.com/bptlab/fCM-query-generator` (2021/11/06)

the queries can support knowledge workers in the immediate decision of choosing the next action.

Limitations

Both the conceptual approach and the implementation presented above are limited. The decision support ranges only to the direct next activity, and the option to change the case model is not supported. Furthermore, there may be no plan at all for various reasons. Each of these points bears challenges and opportunities for future work.

A plan is a sequence of actions. But because a plan may be too restrictive for knowledge work, we use the entirety of plans to support knowledge workers. In the current implementation, however, knowledge workers lose the ability to investigate the plans, for example, to learn about long-term consequences of their decision: Choosing an action may impact future decisions. In the example, requesting an external review before a local review prevents a local review from ever being created. In the future, the implementation should enable knowledge workers to explore plans and/or the consequences of their decisions.

If no plan is found, no decision support can be provided. However, the absence of plans may have different causes, which may provide helpful insights. A model may simply be incapable of satisfying a goal, for example, when the goal requires a multiplicity outside the model's constraints. Similarly, a goal may contradict the current case state, e.g., if we aim at fewer objects than already exist. The goal may be in itself inconsistent, for example, if a single object is required in two different states at the same time. Conflicts within a goal, between a goal and the model, and between the current state and the model should be investigated in future work.

Finally, we may not find plans because the case model's state space may be too large to be searched exhaustively within time and memory constraints. Future work may develop techniques to reduce the size of the state space and heuristics to guide the search.

Currently, goals only use information from the case and the case model. However, more information may exist, for example, in event logs. By enriching the domain model, additional perspectives can be incorporated into goals. If each activity is equipped with an expected duration, temporal constraints can be included. Furthermore, planning may be used for more than just choosing the next action: Based on an existing model and a specified goal, planning might be used to adapt the case model semi-automatically similar to the SmartPM approach [121]. In summary, *Wickr* and planning can empower knowledge workers. Our work, however, is only a starting point and can be extended in many directions.

8.5 Runtime Extension

When an unforeseen situation occurs or when the knowledge on which the model is based changes, knowledge workers must have the freedom

to adapt the case model [98]—of course, within certain boundaries. Such changes apply to the running case. Currently, the case execution engine does not support adapting the case model. In this section, however, we discuss runtime adaptations and their effect on different tasks, such as verification and planning.

The fCM approach allows adding new fragments that adhere to the object lifecycles. *Wickr* is less restrictive: We can add new fragments, and we can add states to the object behaviors as well as state transitions that lead to or origin in an added state. Furthermore, new product terms can be added to the termination condition. With these changes, the behavior is extended, but previous behavior remains valid.

The adapted model must pass structural verification and may be behaviorally verified as well because properties that hold for the original model do not necessarily hold for the new version. It is not enough to start verification from the initial state. Instead, we have to consider the current states of all affected cases. This way, we can guarantee that the adapted model does not violate important properties. Therefore, behavioral verification can assist knowledge workers adapting the case.

If a case is running, the current state needs to be ported to the updated model. This can be done straightforwardly if extensions to the object behaviors and fragments are the only adaptations: No places are removed from the colored Petri net formalization, and the state/marking of the case remains valid. Also, existing plans can still be executed, but new plans with a better score may exist. So, planning should be repeated when the model is changed.

Using the formalization, we can allow all changes as long as the current case state remains part of the case model's state space (cf. state compliance [62]) and essential behavioral and structural properties are satisfied. Colored Petri-net-based semantics, structural and behavioral verification, and planning can assist knowledge workers in making respective changes. In the future, the work by Rinderle-Ma and Reichert [62] may be a starting point for supporting changes that violate state compliance.

Next, we consider the syntax presented in Chapter 4, the semantics presented in Chapters 5–7, and the prototypes and the discussion of run-time extensions in this chapter to evaluate whether *Wickr* is suited for knowledge-intensive and/or data-centric processes.

9 Conceptual Evaluation

Wickr is designed for knowledge-intensive and data-driven processes. It combines aspects from data-centric process modeling (e.g., data behavior and data-based constraints) with activity-centric process modeling (activities and imperative control flow). Therefore, it is a hybrid approach. Yet, including data is not necessarily enough to model knowledge-intensive and data-driven processes. In this chapter, we evaluate whether *Wickr* is suited for this task.

Our conceptual evaluation is based on existing frameworks. In their seminal work [98], Di Ciccio and his co-authors list characteristics and requirements for knowledge-intensive processes. The requirements address both the modeling language and the tooling. Similarly, DALEC [147] is an evaluation framework for data-centric process modeling. DALEC consists of 24 questions, which address different phases of the business process lifecycle and tool support. We use both frameworks to evaluate *Wickr* and to compare it against other approaches, such as fCM.

As a second part of the conceptual evaluation, we separate our contribution from *Wickr*. We combine domain models with BPMN process models and CMMN case models. Based on these combinations, we discuss links and multiplicity constraints and their influence on the process/case behavior.

9.1 Wickr for Knowledge-Intensive Processes

Knowledge work is human-centered, data-driven, unrepeatable, and unpredictable [82]. The involved knowledge is vast and changes quickly so that any model is condemned to be or become incomplete. Therefore, knowledge-intensive processes differ substantially from traditional ones. Di Ciccio et al. [98] present a catalog of characteristics and requirements for knowledge-intensive processes, corresponding modeling languages, and tools. The requirements are grouped into data, knowledge actions, rules and constraints, goals, process, knowledge workers, and environment. For each group, we briefly present the requirements, subsequently evaluate *Wickr*, and compare *Wickr* to fCM.

9.1.1 Data

Data describes the case. It is a foundation for knowledge workers' decisions. Hence, approaches must support data at design- and runtime. Yet, knowledge-intensive processes require a flexible and adaptable data management. This solidifies in the first group of requirements:

R1 Data modeling: Data entities and data relationships relevant to the case should be modeled structurally at design-time.

R2 Late data modeling: It must be possible to adapt the data model at runtime when requirements or knowledge change.

R3 Access to appropriate data: Data must be accessible to authorized knowledge workers independently of activities.

R4 Synchronized access to shared data: Data integrity must be guaranteed even if objects are accessed concurrently.

Comprehensive data support is *Wickr's* primary design goal. At design-time, data can be modeled by classes, associations, and multiplicity constraints. Classes describe data objects, and associations represent relationships among objects. Therefore, *Wickr* satisfies **R1**.

The support for runtime changes, however, is limited. While we may extend the behavior, novel associations and multiplicity constraints may invalidate the current state of a case and must therefore be considered carefully. While we sketched a method for allowing arbitrary changes to the case model, it needs a more thorough investigation. As of now, late data modeling (**R2**) is not supported.

Activities read and write data objects, which are grouped in input and output sets, respectively. This represents data requirements clearly. When a knowledge worker executes an activity, data objects for inputs and outputs can be accessed. Furthermore, *Wickr's* formalism includes global registries of objects and links. These can be used to implement a holistic view on case data that is decoupled from the execution of activities. Therefore, appropriate access to data can be provided (**R3**).

Different activities, fragment instances, and cases can access data objects concurrently. To prevent inconsistencies, *Wickr* does not support multiple simultaneous accesses to the same object. While not part of the presented semantics, objects are bound to an activity upon its beginning and released upon its termination [143]. Therefore, *Wickr* meets **R4**.

In comparison to fCM, *Wickr's* data models contain associations and multiplicity constraints. Therefore, fCM meets **R1** only partially. Late data modeling (**R2**) is supported by neither fCM nor *Wickr*. Data access semantics (**R3**) are comparable. Finally, fCM supports **R4** only partially because cases are isolated.

9.1.2 Knowledge Actions

The behavior of a case model is represented by actions that drive a case forward. Such actions are modeled as activities which may operate on data objects. In knowledge work, information need to be gathered, and decisions need to be made. These and similar tasks are knowledge actions and subject to a group of requirements:

R5 Represent data-driven actions: Data-driven actions must be modeled with data requirements and data operations.

R6 Late actions modeling: It must be possible to model novel actions at runtime.

From activity-centric process modeling approaches, *Wickr* inherits expressive means to model activities. Input sets represent data requirements. An input-output-set combination also encodes data operations, i.e., which objects are read, created, and updated. Furthermore, set data object nodes model batch operations. Implicitly defined are constraints and operations on the links between objects. In conclusion, *Wickr* satisfies **R5**. Since new fragments with novel activities and new states and state transitions can be added at runtime, **R6** is satisfied as well.

Concerning knowledge actions, *Wickr* is an improvement over fCM, since fCM does not support batch processing and links. Furthermore, more runtime extensions are possible because both fragments and object behaviors can be extended.

9.1.3 Rules and Constraints

Processes are subject to regulations, laws, best practices, and guidelines that define rules and constraints for the execution. Not all of them can be modeled appropriately using control flow and data flow. Yet, these rules and constraints are too important to be neglected. Respectively, Di Ciccio et al. define the following requirements:

R7 Formalize rules and constraints: Rules and constraints must be modeled formally to enable automated checks.

R8 Late constraints formalization: Because new requirements and constraints can arise at runtime, it must be possible to model and include them ad hoc.

Wickr has versatile means to express constraints: control flow, data flow, multiplicity constraints, and state transitions in the object behavior. Yet, the constraints apply primarily to individual activities. However, a case model can be verified against a set of temporal logic compliance rules. With these extensions, **R7** is supported.

Model checking can also be used at runtime. Instead of the initial state, we start in the current case state. Therefore, a running case can be verified against new temporal logic rules. This can also be used for monitoring, i.e., to prevent violations. Yet, these rules and constraints are not part of the case model itself. Therefore, we consider **R8** to be supported by *Wickr* when using such an extension.

The fCM approach has similar support for rules and constraints. While multiplicity constraints are not supported, rules can be verified using model checking. However, *Wickr's* rules can be more sophisticated and include links, multiplicity constraints, and object identities.

9.1.4 Goals

Each business process by definition serves a business goal [149]. All activities included in the process are directed towards this goal. Knowledge-

intensive processes are no exception. However, their goals can be highly case-specific, which leads to the following requirements:

R9 Goal modeling: It must be possible to model goals involving data.

R10 Late goal modeling: Goals may change. Knowledge workers must be capable of changing/adding goals.

Wickr case models have a goal specification, which is twofold. The termination condition is a data condition, which is evaluated on objects, their abstract states and links; and the goal multiplicity constraints are evaluated on the links. Both must be satisfied to close the case. *Wickr* therefore supports data-centric goal modeling (**R9**).

However, new requirements may arise at runtime and knowledge workers may subsequently refine the goal. For this reason, new product terms can be added to the termination condition. Besides changing the model, we presented an approach for modeling additional goals at runtime and planning knowledge workers' actions accordingly. Considering this, requirement **R10** is satisfied.

In comparison to fCM, *Wickr's* goal specification is more versatile as links, and multiplicities are not supported in fCM. Furthermore, runtime goal modeling has not been considered in fCM. Yet, a reduced version of our approach could be applied to fCM as well.

9.1.5 Processes

Knowledge-intensive processes have a set of requirements regarding modeling and execution. They should be flexible and adaptable, and the integration of and access to knowledge is crucial. Di Ciccio et al. present a respective group of requirements:

R11 Support for different modeling styles: It must be possible to model versatile constraints appropriately using a variety of modeling styles.

R12 Visibility of the process knowledge: At runtime, knowledge workers must have a holistic view of the case including actions, constraints, and data.

R13 Flexible process execution: Process execution must be flexible: Knowledge workers must be able to re-execute and skip activities.

R14 Deal with unanticipated exceptions: During execution, unanticipated exceptions may occur and must be handled so that the process can recover.

R15 Migration of process instances: Since models may change, it is necessary to migrate running cases to new models.

R16 Learning from event logs: Approaches must be able to learn from past executions that have been recorded in event logs.

R17 Learning from data sources: Approaches must consider other data sources besides event logs to gain/extract domain knowledge.

Wickr case models include process fragments, a data model, and state transition systems for data objects. The combination of modeling styles is suited to create a rich description of knowledge-intensive processes—a description that covers data and process-related aspects. Furthermore, fragments support both imperative control flow and declarative data flow. Therefore, *Wickr* satisfies **R11**.

At runtime, activities are executed, classes are instantiated, and goals are modeled and achieved. Requirement **R12** requires a holistic view on all case related information and constraints. The formalism (i.e., the Petri net) provides a starting point but is in most cases unsuited for knowledge workers. Yet, the structure of the net and its marking can be used to build a holistic view on the case. Therefore, *Wickr* supports **R12** only partially.

In comparison to traditional processes, knowledge-intensive ones must be flexible. In *Wickr*, this flexibility is accomplished through the combination of fragments at runtime. However, all flexibility is modeled. It is, in general, not possible to skip activities or re-execute past ones. Yet, it is not necessary to complete fragment instances, which is equivalent to skipping their execution. Thus, *Wickr* supports **R13** partially.

Furthermore, if an exception occurs during execution, *Wickr* has no build-in mechanism for recovery. Nevertheless, it is possible to adapt the model. In case of unanticipated exceptions, new behavior can be added to recover the case. Also, Andree et al. explore exception handling mechanisms for fCM, which can be ported to *Wickr* [153]. Exception handling is therefore partially supported (**R14**).

Runtime adaptation is part of *Wickr*. If a model is merely extended, running cases can easily be migrated. If other changes are made, migration is more sophisticated: First, it is checked whether the current state is valid for the new case model (state compliance [62]). Only if this is true, we can migrate the instance. Therefore, *Wickr* meets **R15**.

Finally, we consider requirements **R16** and **R17**. No methods utilizing process mining and data mining have been used in the context of *Wickr*. The respective requirements are not met.

Regarding processes, fCM and *Wickr* are comparable. However, *Wickr* models are more detailed. Furthermore, runtime adaptation in fCM is limited to adding fragments, while *Wickr* allows changes to the object behaviors as well. We even sketched a method to allow changing existing elements. In conclusion, *Wickr* improves fCM slightly in regard to the process related requirements.

9.1.6 Knowledge Workers

The execution of knowledge-intensive processes is driven by knowledge workers performing tasks and making decisions. Knowledge workers may have different rights and capabilities. Furthermore, processes

and/or tasks may require collaboration. Case models should include the knowledge worker perspective, respectively:

R18 Knowledge worker's modeling: Human resources with their roles and capabilities must be modeled.

R19 Formalize interactions between knowledge workers: Collaboration and communication among knowledge workers must be modeled.

R20 Define knowledge worker's privileges: Data access rights of knowledge workers and roles must be modeled.

R21 Late knowledge worker's modeling: It must be possible to model new knowledge workers for a running case.

R22 Late privileges modeling: Since rights can change, it must be possible to adapt and extend the model respectively at runtime.

R23 Capture knowledge workers' decision: The decisions of knowledge workers must be captured.

Currently, neither *Wickr* nor fCM support modeling knowledge workers. Yet, first efforts to extend *Wickr* with models for roles, rights, and responsibilities have been researched by Kerstin Andree in the context of a master seminar. Elaboration may be part of future work. The only requirement satisfied by both *Wickr* and fCM is **R23**: In both *Wick* and fCM, human decisions are essential for the case execution because the next activity needs to be selected and further decision outcomes may be captured by data objects and their states.

9.1.7 Environment

Processes are executed in an environment [149]. It includes other processes, organizations, and the technical context. Interactions between a process and its environment are commonly modeled as events. A process can raise events or react to external ones. Many knowledge-intensive processes are reactive. They receive external events and have to react to them swiftly. The last group of requirements addresses the integration of the process environment:

R24 Capture and model external events: External events must be explicitly included in the process model.

R25 Late modeling of external events: It must be possible, to include additional events at runtime.

So far, we did not consider non-start events. However, fragments may include intermediate events, which are similar to activities: Catching events have one input set and one output sets. When a respective event occurs, the input set is read, novel objects are created, and linked to the objects in the input set. Throwing events have one input set and an empty output set. However, catching events are triggered by the

environment and not by the knowledge worker. Thus, they cannot be controlled, and, during verification and planning, they must be treated differently than activities. Nevertheless, we consider *Wickr* to support events (**R24**). Since events are included in fragments, and fragments can be added at runtime, *Wickr* supports late modeling of events (**R25**).

In comparison to fCM, little changed. Events can be included in fragments. However, just as for activities, *Wickr* can support events with inputs and outputs. On the other hand, events have been integrated in Gryphon and Chimera [106], which are a modeling tool and an execution engine for fCM, respectively. In conclusion, we consider *Wickr* and fCM equally well-equipped to model events.

9.1.8 Comparing Wickr to fCM and Others

We use the previous evaluation to compare *Wickr* against fCM and the following approaches: YAWL [28], ADEPT2 [32], SmartPM [121], DECLARE [43], PHILharmonicFlows [66], GSM [60], MailOfMine [65]. We evaluated *Wickr* and fCM but reuse the evaluation results presented by Di Ciccio et al. [98] for the other approaches. Table 9.1 provides on overview of the comparison.

Wickr is strong at design-time. A case model specifies actions, data, and goals using different modeling styles. While excluded in the thesis, events may be modeled as well. In comparison, none of the other approaches support all these elements. But *Wickr* case models do not include knowledge workers nor additional constraints and rules: PHILharmonicFlows models process participants and their privileges; and various approaches model constraints and rules explicitly. Nevertheless, we proposed model checking to consider addition rules and constraints both during design and at runtime.

Compared to fCM, *Wickr* supports additional modifications at runtime. None of the approaches supports ad hoc changes fully: ADEPT and SmartPM, for example, can react to and recover from unanticipated exceptions, but lack the capability to model data and events at runtime.

In regard to the case execution, *Wickr* fares well against the other approaches. Changes to the model can be made; flexible execution is possible by combining fragments; and data is treated as a driving force. However, **R13** requires skipping and re-executing activities, which is not fully supported by *Wickr*. Other approaches, such as ADEPT, DECLARE, and MailOfMine fully satisfy the requirement.

Finally, we consider learning from data sources, such as event logs. MailOfMine and DECLARE support learning from data fully or partially. *Wickr* does not. In the future, event logs and other data may be used to enrich case models and to support knowledge workers.

In conclusion, *Wickr* fares well against the other approaches. However, there are major differences among all of them. Therefore, it is impossible to mark one as generally superior. *Wickr*, for example, lacks support for modeling knowledge worker, but its greatest strengths are the combination and integration of data and process models at design-time, as well as their flexible execution at runtime.

Table 9.1: Overview of the requirements for knowledge–intensive processes [98] and the evaluation for different approaches. – (not supported), ~(partially supported), + (supported), -/+ and -/ (not supported natively but respective extensions exist).

		YAWL	ADEPT2	SmartPM	DECLARE	PHILharmonicFlows	GSM	MailOfMine	fCM	Wickr
R1	Data modeling	~	~	+	~	+	+	–	~	+
R2	Late data modeling	–	–	–	–	–	–	–	–	–
R3	Access to appropriate data	–	–	–	~	+	+	~	+	+
R4	Synchronized access to shared data	–	+	–/+	–	+	–	–	–	+
R5	Represent data–driven actions	~	~	~	~	+	+	–	+	+
R6	Late actions modeling	~	+	~	–/~	–	–	–	+	+
R7	Formalize rules and constraints	–	–	+	+	+	+	–	–/+	–/+
R8	Late constraints formalization	–	–	+	–/~	–	–	–	–/+	–/+
R9	Goal modeling	–	–	–/+	–	–	+	–	+	+
R10	Late goal modeling	–	–	~	–	–	–	–	–	–/+
R11	Support for different modeling styles	–	–/+	+	–/~	+	+	~	~	+
R12	Visibility of process knowledge	–	~	–	~	+	+	~	–	~
R13	Flexible process execution	–	+	–	+	+	~	+	~	~
R14	Deal with unanticipated exceptions	–/+	+	+	–	~	–	–	~	~
R15	Migration of process instances	+	+	–	–/+	–	–	+	~	+
R16	Learning from event logs	–/+	–/+	–	–/+	–	–	+	–	–
R17	Learning from data sources	–	–	–	–/~	–	–	+	–	–
R18	Knowledge workers' modeling	+	+	+	–	+	+	–	–	–
R19	Formalize interaction between knowledge workers	–	–	–	–	–	–	–	–	–
R20	Define knowledge workers' privileges	–	–	–	–	+	+	–	–	–
R21	Late knowledge workers' modeling	+	–	–	–	–	–	–	–	–
R22	Late privileges modeling	–	–	–	–	–	–	–	–	–
R23	Capture knowledge worker's decisions	~	–	~	–	+	+	–	+	+
R24	Capture and model external events	–/+	–	+	–	–	+	–	+	+
R25	External events late modeling	–	–	–	–	–	–	–	+	+

9.2 Wickr for Data-Centric Processes

Wickr's primary concern are knowledge-intensive processes. Therefore, it combines data models and process models in one case model. And while it is not fully data-centric, *Wickr* includes many features of data-centric process models. In this chapter, we evaluate how well *Wickr* is suited for data-centric processes, which are not necessarily knowledge-intensive. We use the results to compare *Wickr* against data-centric process modeling approaches. Therefore, we use the DALEC framework [147], which defines criteria for the design, implementation and execution, diagnosis and optimization, and tooling for data-centric processes. We use the results to compare *Wickr* against PHILharmonicFlows [66] and GSM [60], two well-known and thoroughly researched data-centric process modeling approaches.

9.2.1 Design

During design, process and case models are created and subsequently used, e.g., for verification and validation. Data-centric process models must describe processes, data, and their relationship. Therefore, DALEC defines multiple criteria concerned with design-time:

D1 Modeling language: Which modeling languages are used?

D2 Specification of data representation constructs: Is it possible to model data? Is the specification formalized?

D3 Specification of behavior: Can the data behavior be specified? Is the specification formalized?

D4 Specification of interactions: Can the interactions among data entities be specified? Is the specification formalized?

D5 Process granularity: Can the process be specified on different granularity levels? If yes, are the levels enforced or recommended?

D6 Support for model verification: Are there properties that can be verified? Are they formally defined?

D7 Support of model validation: Is there support for process validation? If there is support, is validation automated?

D8 Specification of data access permissions: Can the access to data be restricted? Is it possible on the attribute level?

D9 Support for variants: Are variants supported? If yes, are they supported for both the process and the data model?

Wickr uses multiple modeling languages (**D1**). BPMN for process fragments. UML for domain models, which include data classes (data representation constructs) and which are formally specified. Therefore, *Wickr* fully satisfies **D2**. For each class, *Wickr* defines the object behavior through (formal) state transition systems (**D3**). Interactions among

objects are captured by associations in the domain model and realized through activities in the process fragment (**D4**). However, the model has no granularity levels (**D5**). A solution to this gap may be landscapes for case models, which provide an abstract view [142]. Furthermore, data access permissions are defined by input and output sets on the level of activities but not for users/roles. **D8** is therefore partially fulfilled. Variants are limited to alternative fragments (**D9**).

Case models can be verified structurally and behaviorally. Structural verification is fully automated. Behavioral verification is semi-automated because the model must be adapted manually for model checking. All in all, *Wickr* meets **D6**. However, it provides no explicit support for validation (**D7**).

9.2.2 Implementation and Execution

Processes that have been modeled can be implemented to be executed. The behavior of data-centric processes is data-driven. Therefore, it is important that the semantics of the process, data, and their combination are clearly defined. Furthermore, data-centric constructs, such as batch processing, should be supported. DALEC captures these by a set of requirements regarding the language and its tool support:

D10 Data-driven enactment: Can process-relevant data be viewed during execution? Is the state and sate progression captured as data? Is data mandatory for every process?

D11 Operational semantics for behavior: Is the execution semantics clearly defined?

D12 Operational semantics for interactions: Is the semantics of links clearly defined?

D13 Support for ad hoc changes and verification: Can the process model be changed at runtime? If yes, can these changes be verified?

D14 Support for monitoring: Can the process and its data be monitored?

D15 Batch execution: Can data be processed in batches?

D16 Support of error handling: Can unanticipated errors be handled?

D17 Support for versioning: Is versioning of the data and process models supported?

Wickr is data-driven (**D10**). Knowledge workers execute activities, during which they read and write data objects. Thereby, the state, consisting of all data objects and running fragment instances, evolves. Furthermore, there is no case without data because at least the case object must be instantiated. This is clearly defined by the formal execution semantics (**D11**).

During execution, links between objects are created and considered (**D12**): If an object is created in the context of other objects, links are created according to the associations in the domain model. If multiple objects are read, they must be linked. *Wickr* uses links furthermore for batch processing (**D15**).

Knowledge workers may also deviate from the predefined process. Therefore, they can extend the fragments and the object behaviors. With additional verification steps, it is even possible to change existing parts or to extend the domain model—the current state, however, must remain valid. Therefore, **D13** is satisfied.

The current tooling does not support monitoring (**D14**), error handling (**D16**), and versioning (**D17**). However, we consider monitoring and versioning limitations of the tools rather than the approach.

9.2.3 Diagnosis and Optimization

Ideally, processes are continually improved. The improvement manifests in a new model, and existing instances may need to be migrated. In data-centric processes, this improvement may apply to the data model:

D18 Data representation construct evolution: Can individual classes be changed? Can existing objects be migrated?

D19 Behavior schema evolution: Can the object behavior evolve and instances be migrated?

D20 Interaction schema evolution: Can new classes be added? Can associations be changed or added?

As described before, case models can be changed, and existing instances can be migrated. However, we have not considered changes to classes (**D18**). The object behavior can be extended, though. In this case, existing instances can be migrated (**D19**). Changes to the domain model often violate state compliance and must be considered carefully (**D20**). Requirements **D18**, **D20** are therefore not supported.

9.2.4 Tool Implementation and Practical Cases

Tool support is crucial for the success and usability of a process modeling approach. At design-time, tools support the creation of correct models; at runtime, models are interpreted by engines, and process instances are monitored. Furthermore, real-world use cases are important to evaluate the approach. Respectively, DALEC defines the following:

D21 Design: Is there a GUI-based modeling tool? Does it support the full language?

D22 Implementation and execution: Is there an implementation capturing the full semantics?

D23 Diagnosis and optimization: Can the process be monitored? If yes, does the monitoring tool provide real-time insights?

D24 Practical examples: Has the approach been used in real world use-cases?

In the previous chapter, we presented proof-of-concept implementations for *Wickr*, including tools for design (**D21**), verification, and execution (**D22**). All tools are publicly available under open-source licenses. However, instead of developing our tools from scratch, we reuse existing ones, most prominently CPNTools [56]. So far, no monitoring is implemented, but CPNTools' capabilities[1] can be used. Finally, case models for various domains have been created using *Wickr*. However, it has not been applied in real-world use cases, but fCM has been used for the SMILE Project [162] in the logistics domain.

9.2.5 Wickr vs. Data-Centric Approaches

We use DALEC to compare *Wickr* with the artifact-centric GSM [60] and the object-centric PHILharmonicFlows [66] (see Table 9.2), which are the most cited approaches in their respective category [147]. Therefore, we take the results from the DALEC publication [147]. *Wickr*—although not designed as a purely data-centric approach—can compete, as shown by the following discussion.

Wickr case models can represent almost all the data-related constructs present in PHILharmonicFlows and even more than GSM. However, *Wickr* lacks different granularity levels (**D5**) and permissions (**D9**)—two requirements which are satisfied by PHILharmonicFlows. *Wickr* and PHILharmonicFlows models can be verified, which GSM models, according to DALEC [147], cannot. All three approaches cannot be validated automatically and do not support variants.

During execution, *Wickr* features runtime adaptability (**D13**), which the other approaches do not. Furthermore, neither PHILharmonicFlows nor GSM support batch processing (**D16**), which *Wickr* does. However, PHILharmonicFlows has full support for evolving the behavioral schema and migrating instances (**D19**), where *Wickr* offers partial or no support. None of the approaches allows changes to the classes and the interaction schemata.

Tool support (**D21**–**D24**) is similar for all three approaches. All approaches offer tools for design and execution, but not for diagnosis and optimization. Yet, while tools for PHILharmonicFlows [85, 125, 126] have been presented in various papers, they are not publicly available. *Wickr's* tools are publicly available under open-source licenses.

Wickr combines the data-centric and activity-centric paradigms. The evaluation shows *Wickr's* feasibility for data-centric process modeling: Compared to GSM and PHILharmonicFlows, *Wickr* does not support the full-spectrum of data-centric aspects, but it integrates activity-centric fragments and data. Runtime adaptability and a clear depiction of dependencies among fragments are *Wickr's* strengths. Yet again, each approach has merits and demerits that need to be considered carefully when choosing a language.

[1] `http://cpntools.org/2018/01/12/monitors/` (2021/11/06)

Table 9.2: Comparison of GSM, PHILharmonicFlows, and Wickr using the DALEC framework for data-centric process management.

No.	Criterion	GSM	PHILharmonicFlows	Wickr
D1	Modeling language	GSM	Custom	BPMN/UML/Custom
D2	Specification of DRCs	+	+	+
D3	Specification of behavior	+	+	+
D4	Specification of interactions	+	+	+
D5	Support for process granularity	-	+	-
D6	Support for model verification	-	+	+
D7	Support for model validation	-	-	-
D8	Specification of data access permissions (read/write)	-	+	~
D9	Support for variants	-	-	~
D10	Data-driven enactment	+	+	+
D11	Operational semantics for behavior	+	+	+
D12	Operational Semantics for interactions	+	+	+
D13	Support for ad hoc changes and verification	-	-	+
D14	Support for monitoring	-	-	-
D15	Batch execution	-	~	+
D16	Support for error handling	-	-	-
D17	Support for versioning	-	-	-
D18	DRC schema evolution	-	~	–
D19	Behavior schema evolution	-	+	~
D20	Interaction schema evolution	-	~	–
D21	Design	+	+	+
D22	Implementation and execution	+	+	+
D23	Diagnosis and optimization	-	-	-
D24	Practical examples	Finance	Medical/HR	Logistics (fCM)

9.3 Transferring Insights to BPMN & CMMN

BPMN is established as the de facto standard for process modeling [107]. However, a lack of flexibility and runtime adaptation hinders BPMN's use for knowledge-intensive processes [82]. The CMMN standard is the OMG's case management standard, which is designed for flexible and knowledge-intensive processes. However, CMMN is not yet used widely in industry.

We presented a semantics for *Wickr* that combines flexible processes and data. However, insights gained during our work on *Wickr* also apply to other approaches, i.e., BPMN and CMMN.

9.3.1 Domain Models and BPMN

BPMN [92] is a language for fully defined and highly structured business processes. It defines the order of activities through control flow and supports data objects and data flow. Activities have input sets, output sets, and an input-output-set relation to refine which data is read and written, but no data model is defined. Furthermore, data in most process instances is limited to one object for each class. An exception is collection data objects, which comprise multiple objects, but their semantics are not further defined.

While similar in their constitution, a BPMN process can be more complex than a *Wickr* fragment. BPMN supports additional constructs, such as, parallel and inclusive gateways, loops, and subprocesses. From a data perspective, loops and multi-instance subprocesses are particularly interesting: If repeated, an activity may create multiple objects of the same class. However, BPMN is incapable of handling this situation properly. It does not define bounds, nor does BPMN specify how data objects are selected. However, all objects of one class can be subsumed in a collection data object, which allows parallel or sequential processing through multi-instance activities and subprocesses.

Figure 9.1 shows a BPMN model for the claim handling process. The claim is received, then assessed. For claims with a regular risk, one internal review is created. For claims with a high risk, one or multiple external reviews are gathered. A subprocess encapsulates the process for requesting and receiving external reviews. It can be executed multiple times concurrently and produces a collection of reviews. Based on the reviews, the claim is approved or declined.

Figure 9.2 depicts the subprocess. First, a review is requested. Afterwards, the review is either received within one week, or a reminder is sent, or the review is canceled. Multiple reminders may be sent if the reviewer misses the deadlines repeatedly.

Similar to *Wickr*, a data model with classes, associations, and multiplicity constraints can refine the semantics of BPMN models with loops and multi-instance subprocesses/activities. First, the activities semantics must be adapted so that they create links as they do in *Wickr*. Then, multiplicity constraints translate to upper and lower bounds for loops, multi-instance subprocesses/activities, and collection data objects. The

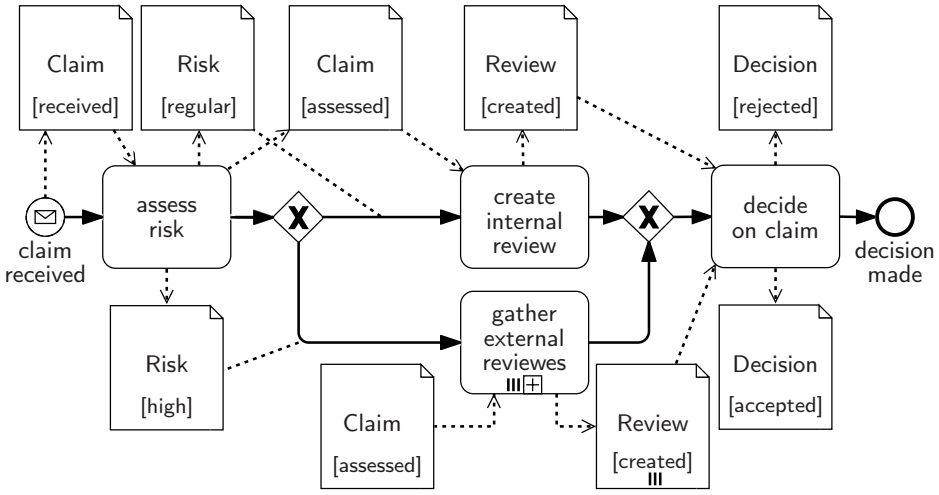


Figure 9.1: BPMN claim handling process, where external reviews are separated into a multi-instance subprocess.

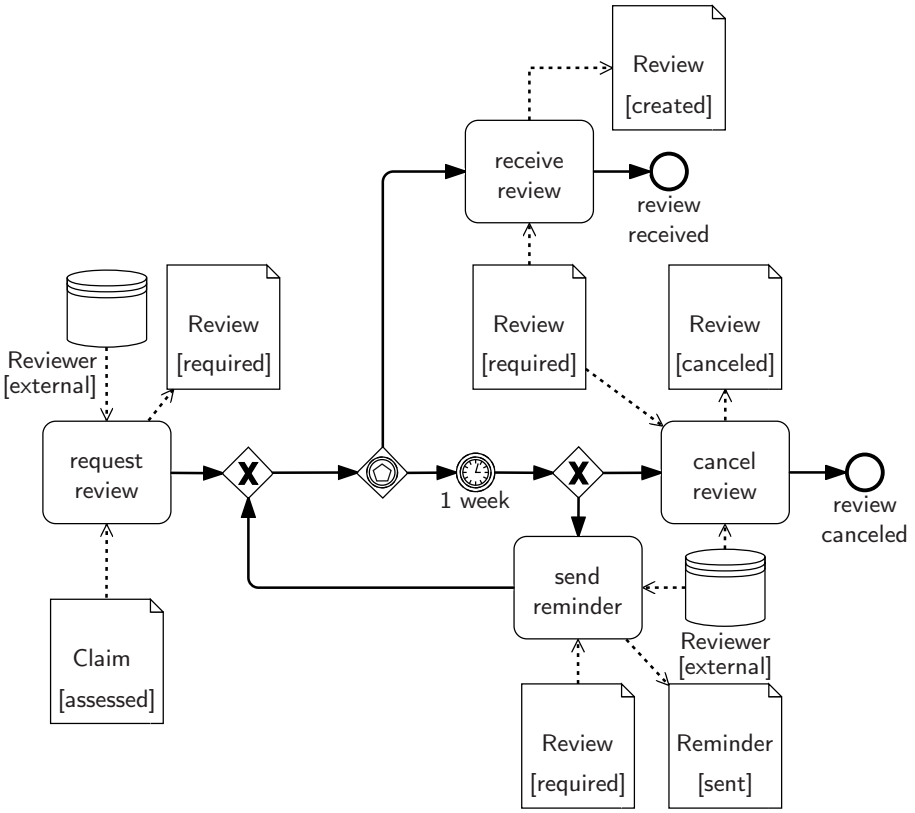


Figure 9.2: Subprocess for requesting and subsequently receiving external reviews.

links also group data objects, which refines loops/multi-instance activities that operate on multiple objects. Furthermore, traditional processes may include cross-case data objects, which can be handled analogously to *Wickr* [159]. Similar to *Wickr*, the refined semantics could be described using colored Petri nets.

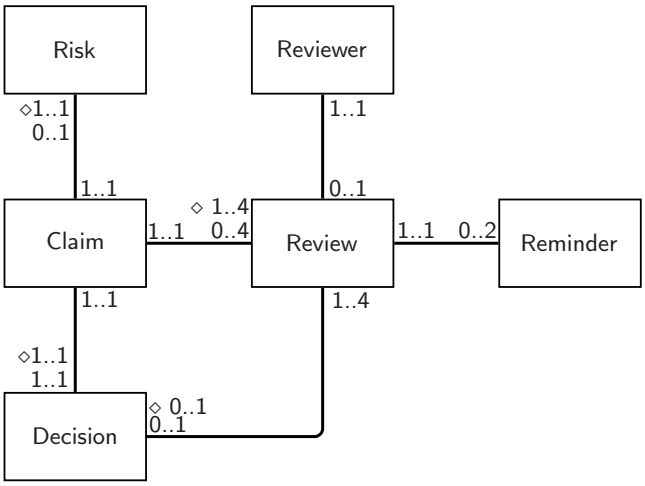


Figure 9.3: Domain model for the BPMN process models in Figures 9.1 and 9.2.

The domain model for the BPMN insurance example (Figure 9.3) shows that each claim has at most four reviews and that for each review there are up to two reminders. Consequently, at most four instances of the subprocess can be executed, and if a reviewer does not meet the deadline after receiving two reminders, the review is canceled.

In conclusion, a domain model can refine the behavior of a BPMN process model. It may define implicit bounds for multi-instance activities, subprocesses, and loops. In this regard, we can reuse ideas from *Wickr*. However, this applies mostly to processes with some flexibility. If the process model supports only a few variants (i.e., no loops, no multi-instance activities or subprocesses), domain models may not refine the process behavior.

9.3.2 Domain Models and CMMN

CMMN [115] is a case management approach favoring flexibility over imperative processes. Activities are arranged into stages. Each stage and each activity may have entry and exit criteria. They can define both events that need to occur and conditions that need to be satisfied to start or to end the respective element. Furthermore, activities and stages may have rules describing additional behavioral properties. These rules include the following:

RequiredRule If the rule evaluates to true, the element must be completed. It is marked by a "!"

RepetitionRule If the rule evaluates to true, the element may be repeated. It is marked by a "#".

Each case includes a case file that consists of one or multiple case file items. These items represent case-relevant data. Case file items can be inputs and outputs of activities. Furthermore, they may be accessed to evaluate rules and conditions. An information model describes the structure of the case file. However, the nature of the information model is not specified: "The structure, as well as the 'language' (or format) to define the structure, is defined by the associated *CaseFileItemDefinition*" [115, p. 22]. Here, we investigate the case that this definition refers to a domain model with global and goal multiplicity constraints. Therefore, we reuse the model in Figure 9.3.

By combining a CMMN model with a *Wickr* domain model, we can infer rules and criteria of the case model from the data structure: Exit criteria should include clauses that prevent global multiplicity constraints from being violated. Activities must be repeated until goal multiplicity constraints are satisfied (RepetitionRule).

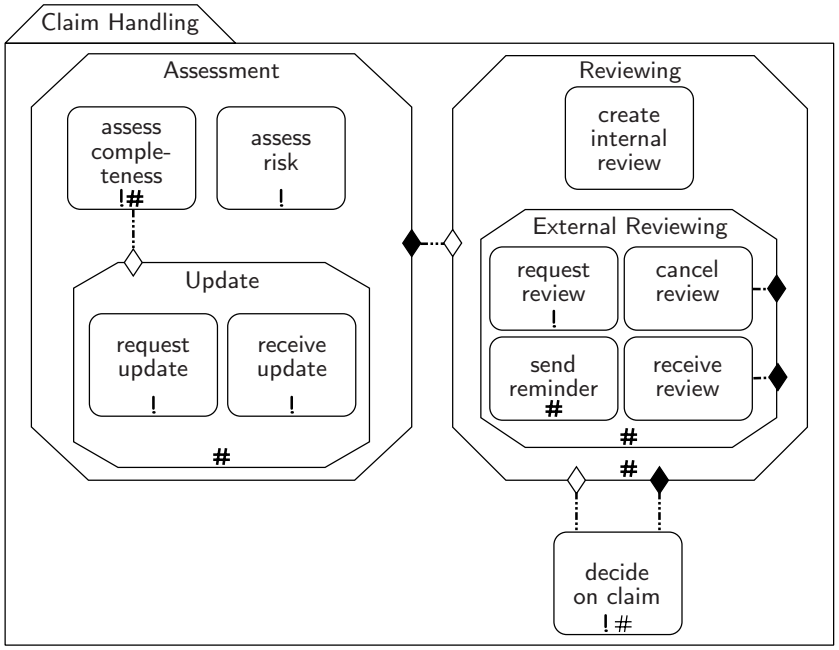


Figure 9.4: CMMN model for handling insurance claims.

Based on the domain model in Figure 9.3, we create the CMMN in Figure 9.4. On the top-level are two stages, "Assessment" and "Reviewing," and one activitiy, "decide on claim." Although connectors have no semantics [115, p. 124], we use them to indicate the general flow of a case: First, a case is assessed, then it is reviewed. Afterwards a decision is made, and if necessary, reviewing is repeated before the decision is remade. The input and outputs of the activities are equal to their counterpart in the BPMN model (Figures 9.1 and 9.2), and the creation of links works as it does in *Wickr*.

Some entry and exit criteria as well as the RepetitionRules and RequiredRules are directly related to the domain model. Criteria can have "on parts," which are required events, and "if parts," which are condi-

tions [115, p. 32]. The entry criterion of stage "Update" is defined as follows:

On Part "assess completeness" completed

If Part claim.state=incomplete

The stage's RepetitionRule is the same as the "if part," so that the "Update" stage can be repeated as long as the claim is incomplete. The same holds for activity "assess completeness." If we look at the "if part" of the exist criterion of stage "Assessment," we see that the goal multiplicity for the risk is reflected: The claim must be linked to a risk:

$$\text{claim.risk} \neq \text{NULL} \wedge \text{claim.state} = \text{complete}$$

After stage "Assessment" is completed, the first instance of stage "Reviewing" is started. Its elements are instantiated. Activity "create internal review" can be executed once, and stage "External Reviewing" can be executed repeatedly. Within the stage, "request review" must be executed; "send reminder" may be executed multiple times but at most twice (the global multiplicity constraint is used in the RepetitionRule); and either "cancel review" or "receive review" is executed to close the stage. As soon as one review exists (goal multiplicity constraint), the "Reviewing" stage can be terminated manually. This is specified by the "if part" of the corresponding exit criterion. Next, decide on claim can be executed. If no decision is created and if less than three reviews exist (global multiplicity constraint), stage "Reviewing" can be re-instantiated (second entry criterion of stage "Reviewing"). After termination, the decision is triggered again. Eventually, reviews have been created; a decision has been made; and the case can be closed.

We described how multiplicity constraints of a domain model are used in rules and criteria of CMMN case models. In contrast to BPMN, rules in CMMN may already operate on the links and multiplicities of data objects. If so, a domain model may contain redundant information, and consistency checks are important. These can be performed on a joint semantics. Alternatively, multiplicity constraints may not be considered in the CMMN model, then the domain model offers additional information which refine the case behavior. Therefore, a joint semantics is also required. As described in this section, *Wickr's* semantics may be a starting point for developing such a semantics for CMMN, i.e., by inferring additional rules from the domain model.

In conclusion, *Wickr* combines data and process modeling to support flexible business processes. Therefore, it is suited for both knowledge-intensive and data-driven processes. *Wickr* shows that the domain model can be used to constrain the process behavior. Yet, this does not only apply to *Wickr*: Highly structured process models, i.e., using BPMN, and other case management approach, i.e., CMMN, can benefit from the integration of a data model as well. Associations and multiplicity constraints can limit the processes' flexibility effectively.

10 Conclusion

Information systems gather, manage, and provide access to data. Often, they do this with regard to business processes, which include activities that read, create, and update data, and whose ordering is partly controlled by data. For both data and processes, models can act as specifications used for implementing and configuring information systems. Yet, process and data models are often handled separately, and many processes make strong assumptions about data: It is local to one process instance, and each data class is instantiated at most once per process instance. However, many processes violate these assumptions.

For example, knowledge-intensive processes, which are data-driven. They require a tight integration of process behavior and data. For this reason, we provide a joint description that shows how data is created by the process, and how the process is constrained by data. Furthermore, defining formal semantics of such a joint model enables us to verify and enact it. Henceforth, we can detect inconsistencies between data models and process models, and we can execute the processes while adhering to the data model.

10.1 Summary of the Contribution

We presented a novel case management approach which is called *Wickr*. It is based on fCM and combines flexible process models with data models. In detail, a *Wickr* case model consists of (i) a set of activity-centric process fragments; (ii) a domain model with classes, associations, and multiplicity constraints; (iii) a set of state transition systems including one system for each class; and (iv) a goal specification consisting of a data condition and a separate set of multiplicity constraints.

Wickr's semantics integrate data into the process. Fragments can be instantiated repeatedly and combined dynamically to execute one of many possible variants. However, data constrains this composition: Activities create and link data objects but must not violate multiplicity constraints. Activities also update data objects but must adhere to the corresponding object behavior. We formally specified *Wickr's* execution semantics by translating case models to colored Petri nets.

Wickr also supports sharing data among processes. So called *cross-case data objects* can be accessed by multiple cases concurrently. They can be used for inter-case communication and synchronization, but constraints must still be upheld. Furthermore, cross-case data objects may be correlated to cases.

In summary, we contributed a case management approach with a joint semantics for data and processes.

- Activities create, read, update, and link data objects.
- The case model is multi-variant and potentially highly concurrent.
- Despite the flexibility, the semantics assert compliance with the domain model and object behaviors.
- Fragment instances memorize data objects they access to assert consistent data access.
- Data can be in the scope of a single case or be shared among cases.
- The formal semantics can describe multiple concurrent cases of the same and different case models.

With this semantics, *Wickr* satisfies many requirements for modeling knowledge-intensive and data-centric processes. The semantics describes how the data model is instantiated by a process, and how the process behavior is constrained by the data model. Therefore, it answers both of our research questions.

Wickr's formal semantics can be used for different tasks. We explored some applications through conceptual work and in prototypes:

- We defined structural consistency criteria for *Wickr* case models.
- We applied model checking for verifying behavioral properties.
- We implemented an engine to assist knowledge-workers.
- We presented a framework for goal modeling and planning.

10.2 Limitations and Future Work

The focus of this thesis is on a joint semantics for flexible processes and data. Therefore, *Wickr* integrates activity-centric process fragments with a data model and object behaviors. Their combined semantics is described in a single colored Petri net. Yet, our work has limitations and opens opportunities for future work.

Conceptually, *Wickr* makes multiple assumptions and only supports a limited set of modeling elements. Furthermore, we only proposed tools to support the design and verification as well as the enactment phase of the BPM lifecycle. Lifting assumptions, extending *Wickr*, and improving its tooling are directions for future work. Finally, *Wickr* should be evaluated with users and real-world use cases. These studies can help to improve *Wickr's* understandability.

10.2.1 Conceptual Extensions

Case models are conceptual models of knowledge-intensive processes. Yet, not all concepts that may be used to describe data or processes are supported by *Wickr*. Hence, the domain model, fragments, and object behaviors can be extended to capture the domain in more detail. Subsequently, the semantics would need to be adapted.

Domain Model. Currently, domain models are limited to classes that are connected by binary, existential, one-to-one and one-to-many associations. Yet, languages, such as UML [123] and OntoUML [100], support more: n-ary, many-to-many associations; special associations, such as composition and aggregation; inheritance and class hierarchies; and much more. In future work, the relationship of such data modeling constructs and case behavior may be investigated.

Inheritance can be used for subtyping [123]: A subclass specializes its super-class.[1] The subclass inherits all attributes and associations but may define additional ones. Furthermore, the object behavior is inherited but can be refined as well. In the fragments, activities that operate on the super-class may also operate on any of its subclasses. However, when an object is created, it is not that clear which class gets instantiated. It may be required to explicitly define the class, e.g., if the super-class is specified, then the object is an instance of that class and none of its subclasses. Alternatively, the knowledge worker can choose from the super-class and all its subclasses, or the choice is deferred to a point at which the specific subclass matters.

Inheritance can also be non-rigid [88]. The phase pattern defines state-specific subclasses for a super-class. The class *Claim* may have the phases *ReceivedClaim*, *CompleteClaim*, *IncompleteClaim*, and so on. With the phase pattern, we can model associations and multiplicity constraints in respect to the state of data objects. A claim may have up to three reviews, but a *ReceivedClaim* has none, an *InReviewClaim* has up to three, and a *ReviewedClaim* has at least one. This must be reflected in both the fragments because activities may not link a *ReceivedClaim* to a Review, and in the object behavior because a claim must not transition from state *reviewed* to *received*. Alternatively, phases may be extracted from the behavior to enrich the domain model.

Currently, *Wickr* semantics exclude data attributes. Yet, business logic often depends on the internal state of objects. Therefore, some approach, e.g., PhilharmonicFlows [66], consider attributes and attribute-based operations. In database systems, integrity constraints, such as denial constraints, can be implemented on an attribute-level to prevent violations of business rules [108]. Such constraints affect the execution of business processes and may be considered in future work.

Process Fragments. In the current version of *Wickr*, events and exceptions are not fully supported. In fCM, both have been considered [106, 153]. Both concepts are important in knowledge-intensive processes [98]. Yet, their semantics can be challenging.

In fCM, events are implemented similarly to activities: Throwing events send data to the process environment. Catching events are triggered by the environment and may introduce data to the case. However, in *Wickr* the new data may be linked to existing objects. In this case, incoming events must be correlated to existing objects: A review that

[1] To keep the discussion of inheritance brief, we do not consider multi-inheritance.

is received must be correlated to the corresponding assignment and reviewer. Furthermore, multiplicity constraints should not be violated.

Events that cause a violation may be ignored. However, this naive approach contradicts the nature of events: An event is a relevant happening, and a process has to react to it. If a reviewer sends two reviews, it may be wrong to ignore the second one. Instead, such an event may be treated as an exception that leads to an inconsistent state. When it occurs, it must be handled, and the process must recover. The SmartPM approach [102, 121] uses automated planning techniques to propose actions to the knowledge workers that ultimately lead to a consistent state. A similar approach may be developed for *Wickr*.

Knowledge Workers. Knowledge-intensive processes are human-centered. Knowledge workers execute activities, make decisions, and plan future actions. However, *Wickr* does not model knowledge workers. Future extensions should include modeling knowledge workers.

It is important to capture knowledge worker's data access permissions. It is important to model what roles may execute an activity—some activities may even need collaboration. We may model delegation, supervision, and other advanced concepts. However, *Wickr* currently lacks the respective concepts.

10.2.2 Application & Tooling

Critical for using and adopting modeling languages are tools. They assist users during all phases of the BPM lifecycle, from creating and analyzing models, to executing cases, and analyzing the past. We provide some prototypes for *Wickr*, but they are neither mature nor complete.

An integrated modeling environment can support the creation of correct case models. It has knowledge about all parts of the case model and the relationships among them. Structural consistency criteria are checked, suggestions to fix violations are made. If an activity models a state transition missing in the object behavior, the tool may propose to add it. And, a simulator (token play) and a model checker may be integrated to further aid validation and verification [152].

Behavioral verification of *Wickr* models is, in general, undecidable. Yet, a special class of state-bounded case models can be verified, but verification is still computationally expensive. In future work, state space reduction techniques, efficient encoding, and heuristics may be introduced to analyze complex models more efficiently.

At runtime, cases are executed, but the model can still be adapted. In the future, an execution engine and the modeling environment should be integrated to allow quick and easy adaptation. Also, cases need to be migrated and verified. As explained, adding fragments or extending the object behavior allows migration of all instances. Furthermore, the tool for goal modeling and planning should become part of the engine. Goals are modeled ad hoc, and knowledge workers should be supported with making decisions in line with their goals.

There are, surely, more applications to be explored. *Wickr* has not yet been used for process mining. Yet, flexible and data-centric behavior is a topic of interest to the process mining community [120, 132, 150, 154]. A connection may be built in the future.

10.2.3 Case Studies & Usability

We focus on *Wickr's* semantics and not on the usability and comprehensibility of case models. A *Wickr* model contains hidden dependencies, which lead to cognitive load and make models harder to understand [12]. However, as a case modeling approach, one of *Wickr's* goals is bridging the business-IT gap—comprehensibility is essential. To improve the understandability, future work may change the visual modeling language and the tooling that is based on it.

Wickr case models contain details about the case behavior and the involved data. Future user studies can explore, which aspects of the case model are relevant to users in different roles. A business user may not be as interested in multiplicity constraints as a software developer.

In the context of fCM, research has been conducted on case model elicitation [161] and landscape models [142]. Hewelt et al. [161] explore methods for eliciting case models starting with the goal, the object life cycles, or the fragments and found differences between them. Furthermore, [142] presents landscape models that aim at providing a broad overview of a given case model. The applicability of these approaches to *Wickr* needs to be investigated.

Similarly, *Wickr* should be evaluated with real-world use cases. Possible research question can address the feasibility, perceived usability and comprehensibility, as well as technical limitations. Are models error-prone? Is behavioral verification feasibly? Are there requirements that cannot be modeled? These and additional questions need to be answered to evaluate the current approach and to direct future efforts.

10.3 Final Remarks

We presented *Wickr*, a modeling approach for flexible processes that integrate data and are driven by human decisions. Hence, *Wickr* is suited for knowledge-intensive processes. *Wickr's* semantics describe how associations and multiplicity constraints influence the process execution: If the process is flexible, constraints in the data model may play an important role during process execution.

Furthermore, *Wickr* allows us to check the coverability of multiplicity constraints based on the logic defined in the process fragments. It tells us the activities that are the root cause for existential associations. In other words, techniques used in data modeling become relevant to the process model because they can express business logic.

All in all, *Wickr* shows the overlap between process and data models. The combination of both can create a synergy that helps to describe flexible, data-centric behavior and to design respective systems.

Appendix

A Wickr is Turing Complete

Wickr is Turing-complete. Therefore, behavioral verification and planning is in general undecidable. To proof that *Wickr* is Turing-complete, we show that case models can simulate a 2-counter machine, which are known to be Turing-complete [1]. The proof is similar to the one for *data-centric dynamic systems* by Montali and Calvanese [114]. The proof shows that *Wickr* can model a 2-counter machine, where

- each counter can be incremented and decremented,
- we can check whether a counter is zero, and
- based on this check, we can jump to a specific instruction.

The domain model (see Figure A.1) contains three classes: the class of the case object and one class for each counter. The case object is associated to arbitrarily many objects of each counter.

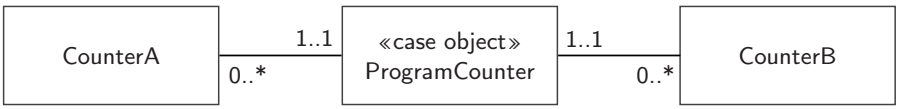


Figure A.1: Domain model for simulating a 2-counter machine.

The state of the case object is used as the program counter. The corresponding state transition system may allow arbitrary state changes because the program behavior can be defined by the fragments.

$$Q_{pc} = \{q_0, q_1, \ldots, q_n\}$$
$$b_{pc} = (Q_{pc}, Q_{pc} \times Q_{pc})$$

Each of the counter classes has a behavior b_{count} with two states, *available* and *deleted*. An object can only transition from state *available* to *deleted*.

$$b_{count} = (\{\mathit{available}, \mathit{deleted}\}, \{(\mathit{available}, \mathit{deleted})\})$$

Since data objects and links in *Wickr* cannot be deleted, the current value of the counter is the number of respective objects in state *available*.

A counter is incremented by created a new object of the corresponding class in state *available*. A counter is decremented by changing the state of a such an object from *available* to *deleted*. All these objects, are linked to the case object. If all the objects of one counter are in state *deleted*, the counter's value is 0. Therefore, we can check whether a counter is 0 or not by modeling an activity that requires all objects for the counter to be in state *deleted*. Therefore, it reads the set of respective

counter objects that are associated to the case object. However, this check only works if there exists at least one object in state *deleted*. Therefore, the start event creates an instance of the case object with a state pointing to the first instruction, and an instance of each of the counter class in state *deleted* (see Figure A.2).

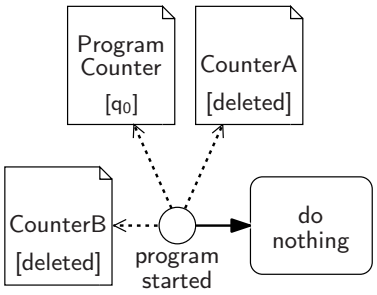


Figure A.2: Initial fragment setting the program counter to q_0 and the two counters CounterA and CounterB to 0.

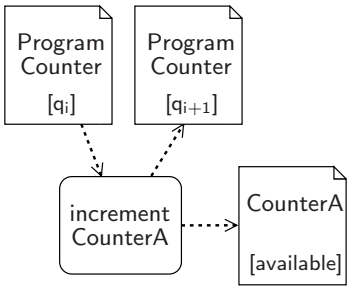


Figure A.3: Fragment for an instruction q_i incrementing CounterA.

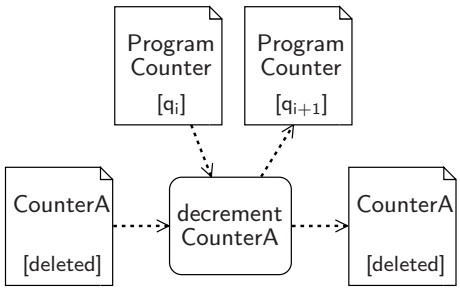


Figure A.4: Fragment for an instruction q_i decrementing CounterA.

Each increment and decrement instruction is translated to a fragment with a single activity (see Figures A.3 and A.4). Each check is translated to two fragments (see Figure A.5): one checks whether the counter is 0, the other one checks whether it is unequal to 0. The two activities can perform different changes to the state of the case object so that a different instruction is executed next.

The termination condition should always be false to prevent an abortion of the modeled program. The fragments above can be used to construct arbitrary 2-counter-machines. Therefore, *Wickr* is Turing-complete.

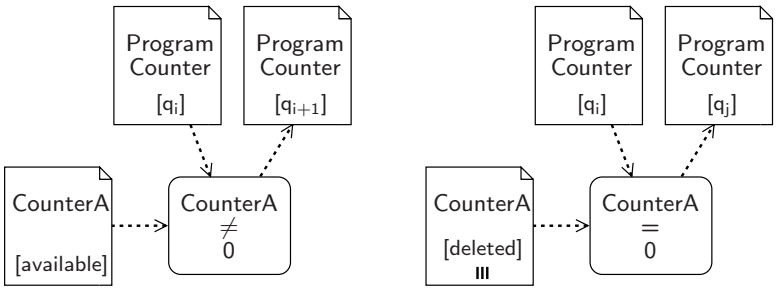


Figure A.5: Fragments for a instruction q_i testing whether CounterA is 0 (go to instruction q_j) or not (continue with instruction q_{i+1}).

Bibliography

[1] Marvin L. Minsky. "Recursive Unsolvability of Post's Problem of "Tag" and other Topics in Theory of Turing Machines". In: *Anals of Mathematics* 74.3 (Nov. 1961), pp. 437–455.

[2] Carl Adam Petri. *Kommunikation mit Automaten*. PhD Thesis. 1962.

[3] Herbert Stachowiak. *Allgemeine Modelltheorie*. Springer, 1973. ISBN: 3-211-81106-0.

[4] Peter P. Chen. "The Entity-Relationship Model: Toward a Unified View of Data". In: *Proceedings of the International Conference on Very Large Data Bases, September 22-24, 1975, Framingham, Massachusetts, USA*. Ed. by Douglas S. Kerr. ACM, 1975, p. 173. URL: https://doi.org/10.1145/1282480.1282492.

[5] Amir Pnueli. "The Temporal Logic of Programs". In: *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977*. IEEE Computer Society, 1977, pp. 46–57. URL: https://doi.org/10.1109/SFCS.1977.32.

[6] Edmund M. Clarke and E. Allen Emerson. "Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic". In: *Logics of Programs, Workshop, Yorktown Heights, New York, USA, May 1981*. Ed. by Dexter Kozen. Vol. 131. Lecture Notes in Computer Science. Springer, 1981, pp. 52–71. ISBN: 3-540-11212-X. URL: https://doi.org/10.1007/BFb0025774.

[7] Heinz Oswald, Rob Esser, and R. Mattmann. "An Environment for Specifying and Executing Hierarchical Petri Nets". In: *Proceedings of the 12th International Conference on Software Engineering, Nice, France, March 26-30, 1990*. Ed. by François-Régis Valette, Peter A. Freeman, and Marie-Claude Gaudel. IEEE Computer Society, 1990, pp. 164–172. ISBN: 0-8186-2026-9. URL: http://dl.acm.org/citation.cfm?id=100319.

[8] W.M. Zuberek. "Timed Petri nets definitions, properties, and applications". In: *Microelectronics Reliability* 31.4 (1991), pp. 627–644. ISSN: 0026-2714. URL: https://doi.org/10.1016/0026-2714(91)90007-T.

[9] Eike Best, Raymond R. Devillers, and Jon G. Hall. "The box calculus: a new causal algebra with multi-label communication". In: *Advances in Petri Nets 1992, The DEMON Project*. Ed. by Grzegorz Rozenberg. Vol. 609. Lecture Notes in Computer Science. Springer, 1992, pp. 21–69. ISBN: 3-540-55610-9. URL: https://doi.org/10.1007/3-540-55610-9_167.

[10] Amit P. Sheth and Marek Rusinkiewicz. "On Transactional Workflows". In: *IEEE Data Eng. Bull.* 16.2 (1993), pp. 37–40. URL: http://sites.computer.org/debull/93JUN-CD.pdf.

[11] Søren Christensen and Kjeld H. Mortensen. *Design/CPN ASK-CTL Manual*. Version 0.9. 1996. URL: http://cpntools.org/wp-content/uploads/2018/01/askctlmanual.pdf.

[12] Thomas R. G. Green and Marian Petre. "Usability Analysis of Visual Programming Environments: A 'Cognitive Dimensions' Framework". In: *J. Vis. Lang. Comput.* 7.2 (1996), pp. 131–174. URL: https://doi.org/10.1006/jvlc.1996.0009.

[13] Manfred Reichert and Peter Dadam. "A Framework for Dynamic Changes in Workflow Management Systems". In: *Eighth International Workshop on Database and Expert Systems Applications, DEXA '97, Toulouse, France, September 1-2, 1997, Proceedings*. Ed. by Roland R. Wagner. IEEE Computer Society, 1997, pp. 42–48. ISBN: 0-8186-8147-0. URL: https://doi.org/10.1109/DEXA.1997.617231.

[14] Arthur H. M. ter Hofstede and Henderik Alex Proper. "How to formalize it?: Formalization principles for information system development methods". In: *Inf. Softw. Technol.* 40.10 (1998), pp. 519–540. URL: https://doi.org/10.1016/S0950-5849(98)00078-0.

[15] Wil MP Van Der Aalst. "Three good reasons for using a Petri-net-based workflow management system". In: *Information and Process Integration in Enterprises*. Springer, 1998, pp. 161–182. URL: https://doi.org/10.1007/978-1-4615-5499-8_10.

[16] Wil M. P. van der Aalst. "Formalization and verification of event-driven process chains". In: *Inf. Softw. Technol.* 41.10 (1999), pp. 639–650. URL: https://doi.org/10.1016/S0950-5849(99)00016-6.

[17] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model checking*. MIT Press, 1999. ISBN: 978-0-262-03270-4. URL: https://mitpress.mit.edu/books/model-checking.

[18] Richard Hull, François Llirbat, Eric Simon, Jianwen Su, Guozhu Dong, Bharat Kumar, and Gang Zhou. "Declarative workflows that support easy modification and dynamic browsing". In: *Proceedings of the international joint conference on Work activities coordination and collaboration 1999, San Francisco, California, USA, February 22-25, 1999*. ACM, 1999, pp. 69–78. URL: https://doi.org/10.1145/295665.295674.

[19] Robin Milner. *Communicating and mobile systems - the Pi-calculus*. Cambridge University Press, 1999. ISBN: 978-0-521-65869-0.

[20] Margaret T. O'Hara, Richard T. Watson, and C. Bruce Kavan. "Managing the three Levels of Change". In: *Inf. Syst. Manag.* 16.3 (1999), pp. 63–70. URL: https://doi.org/10.1201/1078/43197.16.3.19990601/31317.9.

[21] Karsten Schmidt. "Model-Checking with Coverability Graphs". In: *Formal Methods Syst. Des.* 15.3 (1999), pp. 239–254. URL: https://doi.org/10.1023/A:1008753219837.

[22] Frank Wolter and Michael Zakharyaschev. "Temporalizing description logics". In: *Frontiers of Combining Systems* 2 (1999), pp. 379–402.

[23] Wil M. P. van der Aalst, Paulo Barthelmess, Clarence A. Ellis, and Jacques Wainer. "Workflow Modeling Using Proclets". In: *Cooperative Information Systems, 7th International Conference, CoopIS 2000, Eilat, Israel, September 6-8, 2000, Proceedings*. Ed. by Opher Etzion and Peter Scheuermann. Vol. 1901. Lecture Notes in Computer Science. Springer, 2000, pp. 198–209. ISBN: 3-540-41021-X. URL: https://doi.org/10.1007/10722620_20.

[24] August-Wilhelm Scheer. *ARIS—business process modeling*. Springer, 2000. ISBN: 978-3-642-97998-9. URL: https://doi.org/10.1007/978-3-642-97998-9.

[25] Karsten Schmidt. "LoLA: A Low Level Analyser". In: *Application and Theory of Petri Nets 2000, 21st International Conference, ICATPN 2000, Aarhus, Denmark, June 26-30, 2000, Proceeding*. Ed. by Mogens Nielsen and Dan Simpson. Vol. 1825. Lecture Notes in Computer Science. Springer, 2000, pp. 465–474. ISBN: 3-540-67693-7. URL: https://doi.org/10.1007/3-540-44988-4_27.

[26] Paul W. P. J. Grefen, Jochem Vonk, and Peter M. G. Apers. "Global transaction support for workflow management systems: from formal specification to practical implementation". In: *VLDB J.* 10.4 (2001), pp. 316–333. URL: https://doi.org/10.1007/s007780100056.

[27] Wil M. P. van der Aalst, Lachlan Aldred, Marlon Dumas, and Arthur H. M. ter Hofstede. "Design and Implementation of the YAWL System". In: *Advanced Information Systems Engineering, 16th International Conference, CAiSE 2004, Riga, Latvia, June 7-11, 2004, Proceedings*. Ed. by Anne Persson and Janis Stirna. Vol. 3084. Lecture Notes in Computer Science. Springer, 2004, pp. 142–159. ISBN: 3-540-22151-4. URL: https://doi.org/10.1007/978-3-540-25975-6_12.

[28] Wil M. P. van der Aalst and Arthur H. M. ter Hofstede. "YAWL: yet another workflow language". In: *Inf. Syst.* 30.4 (2005), pp. 245–275. URL: https://doi.org/10.1016/j.is.2004.02.002.

[29] Wil M. P. van der Aalst, Mathias Weske, and Dolf Grünbauer. "Case handling: a new paradigm for business process support". In: *Data Knowl. Eng.* 53.2 (2005), pp. 129–162. URL: https://doi.org/10.1016/j.datak.2004.07.003.

[30] Frank Puhlmann and Mathias Weske. "Using the *pi*-Calculus for Formalizing Workflow Patterns". In: *Business Process Management, 3rd International Conference, BPM 2005, Nancy, France, September 5-8, 2005, Proceedings*. Ed. by Wil M. P. van der Aalst, Fabio Casati, and Francisco Curbera. Vol. 3649. 2005, pp. 153–168. URL: https://doi.org/10.1007/11538394_11.

[31] Pasi Pyöriä. "The concept of knowledge work revisited". In: *J. Knowl. Manag.* 9.3 (2005), pp. 116–127. URL: https://doi.org/10.1108/13673270510602818.

[32] Manfred Reichert, Stefanie Rinderle, Ulrich Kreher, and Peter Dadam. "Adaptive Process Management with ADEPT2". In: *Proceedings of the 21st International Conference on Data Engineering, ICDE 2005, 5-8 April 2005, Tokyo, Japan*. Ed. by Karl Aberer, Michael J. Franklin, and Shojiro Nishio. IEEE Computer Society, 2005, pp. 1113–1114. ISBN: 0-7695-2285-8. URL: https://doi.org/10.1109/ICDE.2005.17.

[33] Wil MP Van der Aalst. "Pi calculus versus Petri nets: Let us eat "humble pie" rather than further inflate the "Pi hype"". In: *BPTrends* 3.5 (2005), pp. 1–11. URL: http://www.workflowpatterns.com/documentation/documents/bptrendsPiHype.pdf.

[34] Benkt Wangler and Alexander Backlund. "Information Systems Engineering: What Is It?" In: *Advanced Information Systems Engineering, 17th International Conference, CAiSE 2005, Porto, Portugal, June 13-17, 2005, Proceedings of the CAiSE'05 Workshops, Vol. 2*. Ed. by Jaelson Castro and Ernest Teniente. FEUP Edições, Porto, 2005, pp. 427–437. ISBN: 972-752-077-4. URL: http://www.kybele.etsii.urjc.es/PHISE05/papers/sesionI/WanglerBacklund.pdf.

[35] Peter Tabeling Andreas Knopfel Bernhard Grone. *Fundamental Modeling Concepts: Effective Communication of IT Systems*. Wiley, 2006. ISBN: 978-0-470-02710-3. URL: http://eu.wiley.com/WileyCDA/WileyTitle/productCd-047002710X.html.

[36] Paul W. P. J. Grefen and Jochem Vonk. "A Taxonomy of Transactional Workflow Support". In: *Int. J. Cooperative Inf. Syst.* 15.1 (2006), pp. 87–118. URL: https://doi.org/10.1142/S021884300600130X.

[37] Kurt Jensen, Søren Christensen, and Lars M. Kristensen. *CPN Tools State Space Manual*. Version 0.9. Jan. 2006. URL: http://cpntools.org/wp-content/uploads/2018/01/manual.pdf.

[38] Alan R. Hevner. "The Three Cycle View of Design Science". In: *Scand. J. Inf. Syst.* 19.2 (2007), p. 4. URL: http://aisel.aisnet.org/sjis/vol19/iss2/4.

[39] Nataliya Mulyar, Maja Pesic, Wil M. P. van der Aalst, and Mor Peleg. "Declarative and Procedural Approaches for Modelling Clinical Guidelines: Addressing Flexibility Issues". In: *Business Process Management Workshops, BPM 2007 International Workshops, BPI, BPD, CBP, ProHealth, RefMod, semantics4ws, Brisbane, Australia, September 24, 2007, Revised Selected Papers*. Ed. by Arthur H. M. ter Hofstede, Boualem Benatallah, and Hye-Young Paik. Vol. 4928. Lecture Notes in Computer Science. Springer, 2007, pp. 335–346. ISBN: 978-3-540-78237-7. URL: https://doi.org/10.1007/978-3-540-78238-4_35.

[40] OASIS. *Web Services Business Process Execution Language(WS-BPEL*. Version 2.0. Apr. 2007. URL: http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html.

[41] Object Management Group. *Business Process Modeling and Notation (BPMN)*. Version 1.0.0. Mar. 2007. URL: https://www.omg.org/spec/BPMN/1.0.

[42] Antoni Olivé. *Conceptual modeling of information systems*. Springer, 2007. URL: https://doi.org/10.1007/978-3-540-39390-0.

[43] Maja Pesic, Helen Schonenberg, and Wil M. P. van der Aalst. "DECLARE: Full Support for Loosely-Structured Processes". In: *11th IEEE International Enterprise Distributed Object Computing Conference (EDOC 2007), 15-19 October 2007, Annapolis, Maryland, USA*. IEEE Computer Society, 2007, pp. 287–300. ISBN: 0-7695-2891-0. URL: https://doi.org/10.1109/EDOC.2007.14.

[44] Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT Press, 2008. ISBN: 978-0-262-02649-9. URL: https://mitpress.mit.edu/books/principles-model-checking.

[45] Gero Decker, Hagen Overdick, and Mathias Weske. "Oryx - An Open Modeling Platform for the BPM Community". In: *Business Process Management, 6th International Conference, BPM 2008, Milan, Italy, September 2-4, 2008. Proceedings*. Ed. by Marlon Dumas, Manfred Reichert, and Ming-Chien Shan. Vol. 5240. Lecture Notes in Computer Science. Springer, 2008, pp. 382–385. ISBN: 978-3-540-85757-0. URL: https://doi.org/10.1007/978-3-540-85758-7_29.

[46] Remco M. Dijkman, Marlon Dumas, and Chun Ouyang. "Semantics and analysis of business process models in BPMN". In: *Inf. Softw. Technol.* 50.12 (2008), pp. 1281–1294. URL: https://doi.org/10.1016/j.infsof.2008.02.006.

[47] Richard Hull. "Artifact-Centric Business Process Models: Brief Survey of Research Results and Challenges". In: *On the Move to Meaningful Internet Systems: OTM 2008, OTM 2008 Confederated International Conferences, CoopIS, DOA, GADA, IS, and ODBASE 2008, Monterrey, Mexico, November 9-14, 2008, Proceedings, Part II*. Ed. by Robert Meersman and Zahir Tari. Vol. 5332. Lecture Notes in Computer Science. Springer, 2008, pp. 1152–1163. URL: https://doi.org/10.1007/978-3-540-88873-4_17.

[48] Davide Prandi, Paola Quaglia, and Nicola Zannone. "Formal Analysis of BPMN Via a Translation into COWS". In: *Coordination Models and Languages, 10th International Conference, COORDINATION 2008, Oslo, Norway, June 4-6, 2008. Proceedings*. Ed. by Doug Lea and Gianluigi Zavattaro. Vol. 5052. Lecture Notes in Computer Science. Springer, 2008, pp. 249–263. ISBN: 978-3-540-68264-6. URL: https://doi.org/10.1007/978-3-540-68265-3_16.

[49] Tsukasa Takemura. "Formal Semantics and Verification of BPMN Transaction and Compensation". In: *Proceedings of the 3rd IEEE Asia-Pacific Services Computing Conference, APSCC 2008, Yilan, Taiwan, 9-12 December 2008*. IEEE Computer Society, 2008, pp. 284–290. ISBN: 978-0-7695-3473-2. URL: https://doi.org/10.1109/APSCC.2008.208.

[50] Peter Y. H. Wong and Jeremy Gibbons. "A Process Semantics for BPMN". In: *Formal Methods and Software Engineering, 10th International Conference on Formal Engineering Methods, ICFEM 2008, Kitakyushu-City, Japan, October 27-31, 2008. Proceedings*. Ed. by Shaoying Liu, T. S. E. Maibaum, and Keijiro Araki. Vol. 5256. Lecture Notes in Computer Science. Springer, 2008, pp. 355–374. ISBN: 978-3-540-88193-3. URL: https://doi.org/10.1007/978-3-540-88194-0_22.

[51] Wil M. P. van der Aalst, Maja Pesic, and Helen Schonenberg. "Declarative workflows: Balancing between flexibility and support". In: *Comput. Sci. Res. Dev.* 23.2 (2009), pp. 99–113. URL: https://doi.org/10.1007/s00450-009-0057-9.

[52] Ahmed Awad, Matthias Weidlich, and Mathias Weske. "Specification, Verification and Explanation of Violation for Data Aware Compliance Rules". In: *Service-Oriented Computing, 7th International Joint Conference, ICSOC-ServiceWave 2009, Stockholm, Sweden, November 24-27, 2009. Proceedings*. Ed. by Luciano Baresi, Chi-Hung Chi, and Jun Suzuki. Vol. 5900. Lecture Notes in Computer Science. 2009, pp. 500–515. ISBN: 978-3-642-10382-7. URL: https://doi.org/10.1007/978-3-642-10383-4_37.

[53] BPTrends. *Case Management: Combining Knowledge With Process*. July 2009. URL: https://www.bptrends.com/case-management-combining-knowledge-with-process/.

[54] David Cohn and Richard Hull. "Business Artifacts: A Data-centric Approach to Modeling Business Operations and Processes". In: *IEEE Data Eng. Bull.* 32.3 (2009), pp. 3–9. URL: http://sites.computer.org/debull/A09sept/david.pdf.

[55] Dirk Fahland, Daniel Lübke, Jan Mendling, Hajo A. Reijers, Barbara Weber, Matthias Weidlich, and Stefan Zugal. "Declarative versus Imperative Process Modeling Languages: The Issue of Understandability". In: *Enterprise, Business-Process and Information Systems Modeling, 10th International Workshop, BPMDS 2009, and 14th International Conference, EMMSAD 2009, held at CAiSE 2009, Amsterdam, The Netherlands,*

June 8-9, 2009. Proceedings. Ed. by Terry A. Halpin, John Krogstie, Selmin Nurcan, Erik Proper, Rainer Schmidt, Pnina Soffer, and Roland Ukor. Vol. 29. Lecture Notes in Business Information Processing. Springer, 2009, pp. 353–366. ISBN: 978-3-642-01861-9. URL: https://doi.org/10.1007/978-3-642-01862-6_29.

[56] Kurt Jensen and Lars Michael Kristensen. *Coloured Petri Nets - Modelling and Validation of Concurrent Systems*. Springer, 2009. ISBN: 978-3-642-00283-0. URL: https://doi.org/10.1007/b95112.

[57] Daniel L. Moody. "The "Physics" of Notations: Toward a Scientific Basis for Constructing Visual Notations in Software Engineering". In: *IEEE Trans. Software Eng.* 35.6 (2009), pp. 756–779. URL: https://doi.org/10.1109/TSE.2009.67.

[58] Remco M. Dijkman and Pieter Van Gorp. "BPMN 2.0 Execution Semantics Formalized as Graph Rewrite Rules". In: *Business Process Modeling Notation - Second International Workshop, BPMN 2010, Potsdam, Germany, October 13-14, 2010. Proceedings*. Ed. by Jan Mendling, Matthias Weidlich, and Mathias Weske. Vol. 67. Lecture Notes in Business Information Processing. Springer, 2010, pp. 16–30. ISBN: 978-3-642-16297-8. URL: https://doi.org/10.1007/978-3-642-16298-5_4.

[59] Thomas T. Hildebrandt and Raghava Rao Mukkamala. "Declarative Event-Based Workflow as Distributed Dynamic Condition Response Graphs". In: *Proceedings Third Workshop on Programming Language Approaches to Concurrency and communication-cEntric Software, PLACES 2010, Paphos, Cyprus, 21st March 2010*. Ed. by Kohei Honda and Alan Mycroft. Vol. 69. EPTCS. 2010, pp. 59–73. URL: https://doi.org/10.4204/EPTCS.69.5.

[60] Richard Hull, Elio Damaggio, Fabiana Fournier, Manmohan Gupta, Fenno F. Terry Heath III, Stacy Hobson, Mark H. Linehan, Sridhar Maradugu, Anil Nigam, Piyawadee Sukaviriya, and Roman Vaculín. "Introducing the Guard-Stage-Milestone Approach for Specifying Business Entity Lifecycles". In: *Web Services and Formal Methods - 7th International Workshop, WS-FM 2010, Hoboken, NJ, USA, September 16-17, 2010. Revised Selected Papers*. Ed. by Mario Bravetti and Tevfik Bultan. Vol. 6551. Lecture Notes in Computer Science. Springer, 2010, pp. 1–24. URL: https://doi.org/10.1007/978-3-642-19589-1_1.

[61] David Knuplesch, Linh Thao Ly, Stefanie Rinderle-Ma, Holger Pfeifer, and Peter Dadam. "On Enabling Data-Aware Compliance Checking of Business Process Models". In: *Conceptual Modeling - ER 2010, 29th International Conference on Conceptual Modeling, Vancouver, BC, Canada, November 1-4, 2010. Proceedings*. Ed. by Jeffrey Parsons, Motoshi Saeki, Peretz Shoval, Carson C. Woo, and Yair Wand. Vol. 6412. Lecture Notes in Computer Science. Springer, 2010, pp. 332–346. ISBN: 978-3-642-16372-2. URL: https://doi.org/10.1007/978-3-642-16373-9_24.

[62] Stefanie Rinderle-Ma and Manfred Reichert. "Advanced Migration Strategies for Adaptive Process Management Systems". In: *12th IEEE Conference on Commerce and Enterprise Computing, CEC 2010, Shanghai, China, November 10-12, 2010*. Ed. by Kuo-Ming Chao, Christian Huemer, Birgit Hofreiter, Yinsheng Li, and Nazaraf Shah. IEEE Computer Society, 2010, pp. 56–63. ISBN: 978-1-4244-8433-1. URL: https://doi.org/10.1109/CEC.2010.18.

[63] William N. Robinson and Yi Ding. "A survey of customization support in agent-based business process simulation tools". In: *ACM Trans. Model. Comput. Simul.* 20.3 (2010), 14:1–14:29. URL: https://doi.org/10.1145/1842713.1842717.

[64] 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 Thandar Wynn. "Soundness of workflow nets: classification, decidability, and analysis". In: *Formal Aspects Comput.* 23.3 (2011), pp. 333–363. URL: https://doi.org/10.1007/s00165-010-0161-4.

[65] Claudio Di Ciccio, Massimo Mecella, Monica Scannapieco, Diego Zardetto, and Tiziana Catarci. "MailOfMine - Analyzing Mail Messages for Mining Artful Collaborative Processes". In: *Data-Driven Process Discovery and Analysis - First International Symposium, SIMPDA 2011, Campione d'Italia, Italy, June 29 - July 1, 2011, Revised Selected Papers*. Ed. by Karl Aberer, Ernesto Damiani, and Tharam S. Dillon. Vol. 116. Lecture Notes in Business Information Processing. Springer, 2011, pp. 55–81. ISBN: 978-3-642-34043-7. URL: https://doi.org/10.1007/978-3-642-34044-4_4.

[66] Vera Künzle and Manfred Reichert. "PHILharmonicFlows: towards a framework for object-aware process management". In: *J. Softw. Maintenance Res. Pract.* 23.4 (2011), pp. 205–244. URL: https://doi.org/10.1002/smr.524.

[67] Stephen W. Liddle. "Model-Driven Software Development". In: *Handbook of Conceptual Modeling - Theory, Practice, and Research Challenges*. Ed. by David W. Embley and Bernhard Thalheim. Springer, 2011, pp. 17–54. URL: https://doi.org/10.1007/978-3-642-15865-0_2.

[68] Fabrizio Maria Maggi, Michael Westergaard, Marco Montali, and Wil M. P. van der Aalst. "Runtime Verification of LTL-Based Declarative Process Models". In: *Runtime Verification - Second International Conference, RV 2011, San Francisco, CA, USA, September 27-30, 2011, Revised Selected Papers*. Ed. by Sarfraz Khurshid and Koushik Sen. Vol. 7186. Lecture Notes in Computer Science. Springer, 2011, pp. 131–146. ISBN: 978-3-642-29859-2. URL: https://doi.org/10.1007/978-3-642-29860-8%5C_11.

[69] Paul Pichler, Barbara Weber, Stefan Zugal, Jakob Pinggera, Jan Mendling, and Hajo A. Reijers. "Imperative versus Declarative Process Modeling Languages: An Empirical Investigation". In: *Business Process Management Workshops - BPM 2011 International Workshops, Clermont-Ferrand, France, August 29, 2011, Revised Selected Papers, Part I*. Ed. by Florian Daniel, Kamel Barkaoui, and Schahram Dustdar. Vol. 99. Lecture Notes in Business Information Processing. Springer, 2011, pp. 383–394. ISBN: 978-3-642-28107-5. URL: https://doi.org/10.1007/978-3-642-28108-2_37.

[70] Natalia Sidorova, Christian Stahl, and Nikola Trcka. "Soundness verification for conceptual workflow nets with data: Early detection of errors with the most precision possible". In: *Inf. Syst.* 36.7 (2011), pp. 1026–1043. URL: https://doi.org/10.1016/j.is.2011.04.004.

[71] Elio Damaggio, Alin Deutsch, and Victor Vianu. "Artifact systems with data dependencies and arithmetic". In: *ACM Trans. Database Syst.* 37.3 (2012), 22:1–22:36. URL: https://doi.org/10.1145/2338626.2338628.

[72] Keith D. Swenson. "Position: BPMN Is Incompatible with ACM". In: *Business Process Management Workshops - BPM 2012 International Workshops, Tallinn, Estonia, September 3, 2012. Revised Papers*. Ed. by Marcello La Rosa and Pnina Soffer. Vol. 132. Lecture Notes in Business Information Processing. Springer, 2012, pp. 55–58. ISBN: 978-3-642-36284-2. URL: https://doi.org/10.1007/978-3-642-36285-9_7.

[73] Zhiqiang Yan, Remco M. Dijkman, and Paul W. P. J. Grefen. "Business process model repositories - Framework and survey". In: *Inf. Softw. Technol.* 54.4 (2012), pp. 380–395. URL: https://doi.org/10.1016/j.infsof.2011.11.005.

[74] Pieter Van Gorp and Remco M. Dijkman. "A visual token-based formalization of BPMN 2.0 based on in-place transformations". In: *Inf. Softw. Technol.* 55.2 (2013), pp. 365–394. URL: https://doi.org/10.1016/j.infsof.2012.08.014.

[75] Massimiliano de Leoni and Wil M. P. van der Aalst. "Data-aware process mining: discovering decisions in processes using alignments". In: *Proceedings of the 28th Annual ACM Symposium on Applied Computing, SAC '13, Coimbra, Portugal, March 18-22, 2013*. Ed. by Sung Y. Shin and José Carlos Maldonado. ACM, 2013, pp. 1454–1461. ISBN: 978-1-4503-1656-9. URL: https://doi.org/10.1145/2480362.2480633.

[76] Andreas Meyer, Luise Pufahl, Dirk Fahland, and Mathias Weske. "Modeling and Enacting Complex Data Dependencies in Business Processes". In: *Business Process Management - 11th International Conference, BPM 2013, Beijing, China, August 26-30, 2013. Proceedings*. Ed. by Florian Daniel, Jianmin Wang, and Barbara Weber. Vol. 8094. Lecture Notes in Computer Science. Springer, 2013, pp. 171–186. ISBN: 978-3-642-40175-6. URL: https://doi.org/10.1007/978-3-642-40176-3_14.

[77] Marco Montali, Federico Chesani, Paola Mello, and Fabrizio Maria Maggi. "Towards data-aware constraints in declare". In: *Proceedings of the 28th Annual ACM Symposium on Applied Computing, SAC '13, Coimbra, Portugal, March 18-22, 2013*. Ed. by Sung Y. Shin and José Carlos Maldonado. ACM, 2013, pp. 1391–1396. ISBN: 978-1-4503-1656-9. URL: https://doi.org/10.1145/2480362.2480624.

[78] Marco Montali, Fabrizio Maria Maggi, Federico Chesani, Paola Mello, and Wil M. P. van der Aalst. "Monitoring business constraints with the event calculus". In: *ACM Trans. Intell. Syst. Technol.* 5.1 (2013), 17:1–17:30. URL: https://doi.org/10.1145/2542182.2542199.

[79] Sung Y. Shin and José Carlos Maldonado, eds. *Proceedings of the 28th Annual ACM Symposium on Applied Computing, SAC '13, Coimbra, Portugal, March 18-22, 2013*. ACM, 2013. ISBN: 978-1-4503-1656-9. URL: http://dl.acm.org/citation.cfm?id=2480362.

[80] Natalia Sidorova and Christian Stahl. "Soundness for Resource-Constrained Workflow Nets Is Decidable". In: *IEEE Trans. Syst. Man Cybern. Syst.* 43.3 (2013), pp. 724–729. URL: https://doi.org/10.1109/TSMCA.2012.2210415.

[81] Dmitry Solomakhin, Marco Montali, Sergio Tessaris, and Riccardo De Masellis. "Verification of Artifact-Centric Systems: Decidability and Modeling Issues". In: *Service-Oriented Computing - 11th International Conference, ICSOC 2013, Berlin, Germany, December 2-5, 2013, Proceedings*. Ed. by Samik Basu, Cesare Pautasso, Liang Zhang, and Xiang Fu. Vol. 8274. Lecture Notes in Computer Science. Springer, 2013, pp. 252–

266. ISBN: 978-3-642-45004-4. URL: https://doi.org/10.1007/978-3-642-45005-1_18.

[82] Keith Swenson. *State of the art in case management*. 2013. URL: http://kswenson.purplehillsbooks.com/2013/State-of-the-Art-In-Case-Management_2013.pdf.

[83] Vladimir A. Bashkin and Irina A. Lomazova. "Decidability of k - Soundness for Workflow Nets with an Unbounded Resource". In: *Trans. Petri Nets Other Model. Concurr.* Lecture Notes in Computer Science 9 (2014). Ed. by Maciej Koutny, Serge Haddad, and Alex Yakovlev, pp. 1–18. URL: https://doi.org/10.1007/978-3-662-45730-6_1.

[84] Diego Calvanese, Marco Montali, Montserrat Estañol, and Ernest Teniente. "Verifiable UML Artifact-Centric Business Process Models". In: *Proceedings of the 23rd ACM International Conference on Conference on Information and Knowledge Management, CIKM 2014, Shanghai, China, November 3-7, 2014*. Ed. by Jianzhong Li, Xiaoyang Sean Wang, Minos N. Garofalakis, Ian Soboroff, Torsten Suel, and Min Wang. ACM, 2014, pp. 1289–1298. ISBN: 978-1-4503-2598-1. URL: https://doi.org/10.1145/2661829.2662050.

[85] Carolina Ming Chiao, Vera Künzle, and Manfred Reichert. "A Tool for Supporting Object-Aware Processes". In: *18th IEEE International Enterprise Distributed Object Computing Conference Workshops and Demonstrations, EDOC Workshops 2014, Ulm, Germany, September 1-2, 2014*. Ed. by Georg Grossmann, Sylvain Hallé, Dimka Karastoyanova, Manfred Reichert, and Stefanie Rinderle-Ma. IEEE Computer Society, 2014, pp. 410–413. URL: https://doi.org/10.1109/EDOCW.2014.69.

[86] Montserrat Estañol, Anna Queralt, Maria-Ribera Sancho, and Ernest Teniente. "Specifying Artifact-Centric Business Process Models in UML". In: *Business Modeling and Software Design - 4th International Symposium, BMSD 2014, Luxembourg, Luxembourg, June 24-26, 2014, Revised Selected Papers*. Ed. by Boris Shishkov. Vol. 220. Lecture Notes in Business Information Processing. Springer, 2014, pp. 62–81. URL: https://doi.org/10.1007/978-3-319-20052-1_4.

[87] Georg Grossmann, Sylvain Hallé, Dimka Karastoyanova, Manfred Reichert, and Stefanie Rinderle-Ma, eds. *18th IEEE International Enterprise Distributed Object Computing Conference Workshops and Demonstrations, EDOC Workshops 2014, Ulm, Germany, September 1-2, 2014*. IEEE Computer Society, 2014. URL: https://ieeexplore.ieee.org/xpl/conhome/6971861/proceeding.

[88] Giancarlo Guizzardi. "Ontological Patterns, Anti-Patterns and Pattern Languages for Next-Generation Conceptual Modeling". In: *Conceptual Modeling - 33rd International Conference, ER 2014, Atlanta, GA, USA, October 27-29, 2014. Proceedings*. Ed. by Eric S. K. Yu, Gillian Dobbie, Matthias Jarke, and Sandeep Purao. Vol. 8824. Lecture Notes in Computer Science. Springer, 2014, pp. 13–27. ISBN: 978-3-319-12205-2. URL: https://doi.org/10.1007/978-3-319-12206-9_2.

[89] Matheus Hauder, Simon Pigat, and Florian Matthes. "Research Challenges in Adaptive Case Management: A Literature Review". In: *18th IEEE International Enterprise Distributed Object Computing Conference Workshops and Demonstrations, EDOC Workshops 2014, Ulm, Germany, September 1-2, 2014*. Ed. by Georg Grossmann, Sylvain Hallé, Dimka Karastoyanova, Manfred Reichert, and Stefanie Rinderle-Ma. IEEE

Computer Society, 2014, pp. 98–107. URL: https://doi.org/10.1109/EDOCW.2014.24.

[90] Andreas Meyer, Nico Herzberg, Frank Puhlmann, and Mathias Weske. "Implementation Framework for Production Case Management: Modeling and Execution". In: *18th IEEE International Enterprise Distributed Object Computing Conference, EDOC 2014, Ulm, Germany, September 1-5, 2014*. Ed. by Manfred Reichert, Stefanie Rinderle-Ma, and Georg Grossmann. IEEE Computer Society, 2014, pp. 190–199. ISBN: 978-1-4799-5470-4. URL: https://doi.org/10.1109/EDOC.2014.34.

[91] Andreas Meyer and Mathias Weske. "Weak Conformance between Process Models and Synchronized Object Life Cycles". In: *Service-Oriented Computing - 12th International Conference, ICSOC 2014, Paris, France, November 3-6, 2014. Proceedings*. Ed. by Xavier Franch, Aditya K. Ghose, Grace A. Lewis, and Sami Bhiri. Vol. 8831. Lecture Notes in Computer Science. Springer, 2014, pp. 359–367. ISBN: 978-3-662-45390-2. URL: https://doi.org/10.1007/978-3-662-45391-9_25.

[92] Object Management Group. *Business Process Model and Notation (BPMN)*. Version 2.0.2. Jan. 2014. URL: https://www.omg.org/spec/BPMN.

[93] Object Management Group. *Object Constraint Language (OCL)*. Version 2.4. Feb. 2014. URL: https://www.omg.org/spec/OCL.

[94] Monique Snoeck. *Enterprise Information Systems Engineering - The MERODE Approach*. The Enterprise Engineering Series. Springer, 2014. ISBN: 978-3-319-10144-6. URL: https://doi.org/10.1007/978-3-319-10145-3.

[95] Kimon Batoulis, Andreas Meyer, Ekaterina Bazhenova, Gero Decker, and Mathias Weske. "Extracting Decision Logic from Process Models". In: *Advanced Information Systems Engineering - 27th International Conference, CAiSE 2015, Stockholm, Sweden, June 8-12, 2015, Proceedings*. Ed. by Jelena Zdravkovic, Marite Kirikova, and Paul Johannesson. Vol. 9097. Lecture Notes in Computer Science. Springer, 2015, pp. 349–366. ISBN: 978-3-319-19068-6. URL: https://doi.org/10.1007/978-3-319-19069-3_22.

[96] Anne Baumgrass, Claudio Di Ciccio, Remco M. Dijkman, Marcin Hewelt, Jan Mendling, Andreas Meyer, Shaya Pourmirza, Mathias Weske, and Tsun Yin Wong. "GET Controller and UNICORN: Event-driven Process Execution and Monitoring in Logistics". In: *Proceedings of the BPM Demo Session 2015 Co-located with the 13th International Conference on Business Process Management (BPM 2015), Innsbruck, Austria, September 2, 2015*. Ed. by Florian Daniel and Stefan Zugal. Vol. 1418. CEUR Workshop Proceedings. CEUR-WS.org, 2015, pp. 75–79. URL: http://ceur-ws.org/Vol-1418/paper16.pdf.

[97] Riad Boussetoua, Hammadi Bennoui, Allaoua Chaoui, Khaled Khalfaoui, and Elhillali Kerkouche. "An automatic approach to transform BPMN models to Pi-Calculus". In: *12th IEEE/ACS International Conference of Computer Systems and Applications, AICCSA 2015, Marrakech, Morocco, November 17-20, 2015*. IEEE Computer Society, 2015, pp. 1–8. URL: https://doi.org/10.1109/AICCSA.2015.7507176.

[98] Claudio Di Ciccio, Andrea Marrella, and Alessandro Russo. "Knowledge-Intensive Processes: Characteristics, Requirements and Analysis of Contemporary Approaches". In: *J. Data Semant.* 4.1 (2015), pp. 29–57. URL: https://doi.org/10.1007/s13740-014-0038-4.

[99] Florian Daniel and Stefan Zugal, eds. *Proceedings of the BPM Demo Session 2015 Co-located with the 13th International Conference on Business Process Management (BPM 2015), Innsbruck, Austria, September 2, 2015*. Vol. 1418. CEUR Workshop Proceedings. CEUR-WS.org, 2015. URL: http://ceur-ws.org/Vol-1418.

[100] Giancarlo Guizzardi. *Ontological Foundations for Structural Conceptual Models*. Telematica Instituut / CTIT, Oct. 2015. ISBN: 90-75176-81-3. URL: https://research.utwente.nl/en/publications/ontological-foundations-for-structural-conceptual-models.

[101] Stephan Haarmann, Nikolai Podlesny, Marcin Hewelt, Andreas Meyer, and Mathias Weske. "Production Case Management: A Prototypical Process Engine to Execute Flexible Business Processes". In: *Proceedings of the BPM Demo Session 2015 Co-located with the 13th International Conference on Business Process Management (BPM 2015), Innsbruck, Austria, September 2, 2015*. Ed. by Florian Daniel and Stefan Zugal. Vol. 1418. CEUR Workshop Proceedings. CEUR-WS.org, 2015, pp. 110–114. URL: http://ceur-ws.org/Vol-1418/paper23.pdf.

[102] Andrea Marrella, Patris Halapuu, Massimo Mecella, and Sebastian Sardiña. "SmartPM: An Adaptive Process Management System for Executing Processes in Cyber-Physical Domains". In: *Proceedings of the BPM Demo Session 2015 Co-located with the 13th International Conference on Business Process Management (BPM 2015), Innsbruck, Austria, September 2, 2015*. Ed. by Florian Daniel and Stefan Zugal. Vol. 1418. CEUR Workshop Proceedings. CEUR-WS.org, 2015, pp. 115–119. URL: http://ceur-ws.org/Vol-1418/paper24.pdf.

[103] Juliana Baptista dos Santos França, Joanne Manhães Netto, Juliana do E. Santo Carvalho, Flávia Maria Santoro, Fernanda Araujo Baião, and Mariano Gomes Pimentel. "KIPO: the knowledge-intensive process ontology". In: *Softw. Syst. Model.* 14.3 (2015), pp. 1127–1157. URL: https://doi.org/10.1007/s10270-014-0397-1.

[104] Wil M. P. van der Aalst. *Process Mining - Data Science in Action, Second Edition*. Springer, 2016. ISBN: 978-3-662-49850-7. URL: https://doi.org/10.1007/978-3-662-49851-4.

[105] Heiko Beck, Marcin Hewelt, and Luise Pufahl. "Extending Fragment-Based Case Management with State Variables". In: *Business Process Management Workshops - BPM 2016 International Workshops, Rio de Janeiro, Brazil, September 19, 2016, Revised Papers*. Ed. by Marlon Dumas and Marcelo Fantinato. Vol. 281. Lecture Notes in Business Information Processing. 2016, pp. 227–238. ISBN: 978-3-319-58456-0. URL: https://doi.org/10.1007/978-3-319-58457-7_17.

[106] Jonas Beyer, Patrick Kuhn, Marcin Hewelt, Sankalita Mandal, and Mathias Weske. "Unicorn meets Chimera: Integrating External Events into Case Management". In: *Proceedings of the BPM Demo Track 2016 Co-located with the 14th International Conference on Business Process Management (BPM 2016), Rio de Janeiro, Brazil, September 21, 2016*. Ed. by Leonardo Azevedo and Cristina Cabanillas. Vol. 1789. CEUR Workshop Proceedings. CEUR-WS.org, 2016, pp. 67–72. URL: http://ceur-ws.org/Vol-1789/bpm-demo-2016-paper13.pdf.

[107] Marlon Dumas and Dietmar Pfahl. "Modeling Software Processes Using BPMN: When and When Not?" In: *Managing Software Process Evolution*. Ed. by Marco Kuhrmann, Jürgen Münch, Ita Richardson, Andreas Rausch, and He Zhang. Springer, Sept. 2016, pp. 165–183.

ISBN: 978-3-319-31545-4. URL: https://doi.org/10.1007/978-3-319-31545-4_9.

[108] Ramez Elmasri and Shamkant B. Navathe. *Fundamentals of Database Systems, 7th Edition*. Pearson, 2016. ISBN: 978-0-133-97077-7.

[109] European Parliament and Council. *Regulation (EU) 2016/679 of the European Parliament and of the Council of 27 April 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing Directive 95/46/EC (General Data Protection Regulation)*. Apr. 2016. URL: http://data.europa.eu/eli/reg/2016/679/oj.

[110] Walid Fdhila, Manuel Gall, Stefanie Rinderle-Ma, Juergen Mangler, and Conrad Indiono. "Classification and Formalization of Instance-Spanning Constraints in Process-Driven Applications". In: *Business Process Management - 14th International Conference, BPM 2016, Rio de Janeiro, Brazil, September 18-22, 2016. Proceedings*. Ed. by Marcello La Rosa, Peter Loos, and Oscar Pastor. Vol. 9850. Lecture Notes in Computer Science. Springer, 2016, pp. 348–364. ISBN: 978-3-319-45347-7. URL: https://doi.org/10.1007/978-3-319-45348-4_20.

[111] Malik Ghallab, Dana S. Nau, and Paolo Traverso. *Automated Planning and Acting*. Cambridge University Press, 2016. ISBN: 978-1-107-03727-4. URL: http://www.cambridge.org/de/academic/subjects/computer-science/artificial-intelligence-and-natural-language-processing/automated-planning-and-acting?format=HB.

[112] Marcin Hewelt and Mathias Weske. "A Hybrid Approach for Flexible Case Modeling and Execution". In: *Business Process Management Forum - BPM Forum 2016, Rio de Janeiro, Brazil, September 18-22, 2016, Proceedings*. Ed. by Marcello La Rosa, Peter Loos, and Oscar Pastor. Vol. 260. Lecture Notes in Business Information Processing. Springer, 2016, pp. 38–54. ISBN: 978-3-319-45467-2. URL: https://doi.org/10.1007/978-3-319-45468-9_3.

[113] Matthias Kunze and Mathias Weske. *Behavioural Models - From Modelling Finite Automata to Analysing Business Processes*. Springer, 2016. ISBN: 978-3-319-44958-6. URL: https://doi.org/10.1007/978-3-319-44960-9.

[114] Marco Montali and Diego Calvanese. "Soundness of data-aware, case-centric processes". In: *Int. J. Softw. Tools Technol. Transf.* 18.5 (2016), pp. 535–558. URL: https://doi.org/10.1007/s10009-016-0417-2.

[115] Object Management Group. *Case Management Model and Notation (CMMN*. Version 1.1. Dec. 2016. URL: https://www.omg.org/spec/CMMN.

[116] Object Management Group. *Meta Object Facility (MOF)*. Version 2.5.1. Oct. 2016. URL: https://www.omg.org/spec/MOF.

[117] Tim Sporleder. *Fragmentbasiertes Case-Management: Spezifikation und translationale Semantik*. Master Thesis. Sept. 2016.

[118] Wil M. P. van der Aalst, Guangming Li, and Marco Montali. "Object-Centric Behavioral Constraints". In: *CoRR* abs/1703.05740 (2017). arXiv: 1703.05740. URL: http://arxiv.org/abs/1703.05740.

[119] Kimon Batoulis, Stephan Haarmann, and Mathias Weske. "Various Notions of Soundness for Decision-Aware Business Processes". In: *Conceptual Modeling - 36th International Conference, ER 2017, Valencia, Spain, November 6-9, 2017, Proceedings*. Ed. by Heinrich C. Mayr, Giancarlo Guizzardi, Hui Ma, and Oscar Pastor. Vol. 10650. Lecture Notes in Computer Science. Springer, 2017, pp. 403–418. ISBN: 978-3-319-69903-5. URL: https://doi.org/10.1007/978-3-319-69904-2_31.

[120] Guangming Li, Renata Medeiros de Carvalho, and Wil M. P. van der Aalst. "Automatic Discovery of Object-Centric Behavioral Constraint Models". In: *Business Information Systems - 20th International Conference, BIS 2017, Poznan, Poland, June 28-30, 2017, Proceedings*. Ed. by Witold Abramowicz. Vol. 288. Lecture Notes in Business Information Processing. Springer, 2017, pp. 43–58. ISBN: 978-3-319-59335-7. URL: https://doi.org/10.1007/978-3-319-59336-4_4.

[121] Andrea Marrella, Massimo Mecella, and Sebastian Sardiña. "Intelligent Process Adaptation in the SmartPM System". In: *ACM Trans. Intell. Syst. Technol.* 8.2 (2017), 25:1–25:43. URL: https://doi.org/10.1145/2948071.

[122] Marco Montali and Andrey Rivkin. "DB-Nets: On the Marriage of Colored Petri Nets and Relational Databases". In: *Trans. Petri Nets Other Model. Concurr.* Lecture Notes in Computer Science 12 (2017). Ed. by Maciej Koutny, Jetty Kleijn, and Wojciech Penczek, pp. 91–118. URL: https://doi.org/10.1007/978-3-662-55862-1_5.

[123] Object Management Group. *Unified Modeling Language (UML)*. Version 2.5.1. Dec. 2017. URL: https://www.omg.org/spec/UML.

[124] Shaya Pourmirza, Sander Peters, Remco M. Dijkman, and Paul Grefen. "A systematic literature review on the architecture of business process management systems". In: *Inf. Syst.* 66 (2017), pp. 43–58. URL: https://doi.org/10.1016/j.is.2017.01.007.

[125] Sebastian Steinau, Kevin Andrews, and Manfred Reichert. "A Modeling Tool for PHILharmonicFlows Objects and Lifecycle Processes". In: *Proceedings of the BPM Demo Track and BPM Dissertation Award co-located with 15th International Conference on Business Process Modeling (BPM 2017), Barcelona, Spain, September 13, 2017*. Ed. by Robert Clarisó, Henrik Leopold, Jan Mendling, Wil M. P. van der Aalst, Akhil Kumar, Brian T. Pentland, and Mathias Weske. Vol. 1920. CEUR Workshop Proceedings. CEUR-WS.org, 2017. URL: http://ceur-ws.org/Vol-1920/BPM_2017_paper_196.pdf.

[126] Kevin Andrews, Sebastian Steinau, and Manfred Reichert. "A Tool for Supporting Ad-Hoc Changes to Object-Aware Processes". In: *22nd IEEE International Enterprise Distributed Object Computing Workshop, EDOC Workshops 2018, Stockholm, Sweden, October 16-19, 2018*. IEEE Computer Society, 2018, pp. 220–223. ISBN: 978-1-5386-4141-5. URL: https://doi.org/10.1109/EDOCW.2018.00041.

[127] Carlo Combi, Barbara Oliboni, Mathias Weske, and Francesca Zerbato. "Conceptual Modeling of Processes and Data: Connecting Different Perspectives". In: *Conceptual Modeling - 37th International Conference, ER 2018, Xi'an, China, October 22-25, 2018, Proceedings*. Ed. by Juan Trujillo, Karen C. Davis, Xiaoyong Du, Zhanhuai Li, Tok Wang Ling, Guoliang Li, and Mong-Li Lee. Vol. 11157. Lecture Notes in Computer Science. Springer, 2018, pp. 236–250. ISBN: 978-3-030-00846-8. URL: https://doi.org/10.1007/978-3-030-00847-5_18.

[128] Marlon Dumas, Marcello La Rosa, Jan Mendling, and Hajo A. Reijers. *Fundamentals of Business Process Management, Second Edition*. Springer, 2018. ISBN: 978-3-662-56508-7. URL: https://doi.org/10.1007/978-3-662-56509-4.

[129] Montserrat Estañol, Maria-Ribera Sancho, and Ernest Teniente. "Ensuring the semantic correctness of a BAUML artifact-centric BPM". In: *Inf. Softw. Technol.* 93 (2018), pp. 147–162. URL: https://doi.org/10.1016/j.infsof.2017.09.003.

[130] Stephan Haarmann, Kimon Batoulis, and Mathias Weske. "Compliance Checking for Decision-Aware Process Models". In: *Business Process Management Workshops - BPM 2018 International Workshops, Sydney, NSW, Australia, September 9-14, 2018, Revised Papers*. Ed. by Florian Daniel, Quan Z. Sheng, and Hamid Motahari. Vol. 342. Lecture Notes in Business Information Processing. Springer, 2018, pp. 494–506. ISBN: 978-3-030-11640-8. URL: https://doi.org/10.1007/978-3-030-11641-5_39.

[131] Marcin Hewelt, Felix Wolff, Sankalita Mandal, Luise Pufahl, and Mathias Weske. "Towards a Methodology for Case Model Elicitation". In: *Enterprise, Business-Process and Information Systems Modeling - 19th International Conference, BPMDS 2018, 23rd International Conference, EMMSAD 2018, Held at CAiSE 2018, Tallinn, Estonia, June 11-12, 2018, Proceedings*. Ed. by Jens Gulden, Iris Reinhartz-Berger, Rainer Schmidt, Sérgio Guerreiro, Wided Guédria, and Palash Bera. Vol. 318. Lecture Notes in Business Information Processing. Springer, 2018, pp. 181–195. ISBN: 978-3-319-91703-0. URL: https://doi.org/10.1007/978-3-319-91704-7_12.

[132] Guangming Li, Eduardo González López de Murillas, Renata Medeiros de Carvalho, and Wil M. P. van der Aalst. "Extracting Object-Centric Event Logs to Support Process Mining on Databases". In: *Information Systems in the Big Data Era - CAiSE Forum 2018, Tallinn, Estonia, June 11-15, 2018, Proceedings*. Ed. by Jan Mendling and Haralambos Mouratidis. Vol. 317. Lecture Notes in Business Information Processing. Springer, 2018, pp. 182–199. ISBN: 978-3-319-92900-2. URL: https://doi.org/10.1007/978-3-319-92901-9_16.

[133] Sebastian Steinau, Kevin Andrews, and Manfred Reichert. "The Relational Process Structure". In: *Advanced Information Systems Engineering - 30th International Conference, CAiSE 2018, Tallinn, Estonia, June 11-15, 2018, Proceedings*. Ed. by John Krogstie and Hajo A. Reijers. Vol. 10816. Lecture Notes in Computer Science. Springer, 2018, pp. 53–67. ISBN: 978-3-319-91562-3. URL: https://doi.org/10.1007/978-3-319-91563-0_4.

[134] Amine Abbad Andaloussi, Jon Buch-Lorentsen, Hugo A. López, Tijs Slaats, and Barbara Weber. "Exploring the Modeling of Declarative Processes Using a Hybrid Approach". In: *Conceptual Modeling - 38th International Conference, ER 2019, Salvador, Brazil, November 4-7, 2019, Proceedings*. Ed. by Alberto H. F. Laender, Barbara Pernici, Ee-Peng Lim, and José Palazzo M. de Oliveira. Vol. 11788. Lecture Notes in Computer Science. Springer, 2019, pp. 162–170. ISBN: 978-3-030-33222-8. URL: https://doi.org/10.1007/978-3-030-33223-5_14.

[135] Amine Abbad Andaloussi, Andrea Burattin, Tijs Slaats, Anette Chelina Møller Petersen, Thomas T. Hildebrandt, and Barbara Weber. "Exploring the Understandability of a Hybrid Process Design Artifact Based on DCR Graphs". In: *Enterprise, Business-Process and Information*

Systems Modeling - 20th International Conference, BPMDS 2019, 24th International Conference, EMMSAD 2019, Held at CAiSE 2019, Rome, Italy, June 3-4, 2019, Proceedings. Ed. by Iris Reinhartz-Berger, Jelena Zdravkovic, Jens Gulden, and Rainer Schmidt. Vol. 352. Lecture Notes in Business Information Processing. Springer, 2019, pp. 69–84. ISBN: 978-3-030-20617-8. URL: https://doi.org/10.1007/978-3-030-20618-5_5.

[136] Alessandro Artale, Alisa Kovtunova, Marco Montali, and Wil M. P. van der Aalst. "Modeling and Reasoning over Declarative Data-Aware Processes with Object-Centric Behavioral Constraints". In: *Business Process Management - 17th International Conference, BPM 2019, Vienna, Austria, September 1-6, 2019, Proceedings*. Ed. by Thomas T. Hildebrandt, Boudewijn F. van Dongen, Maximilian Röglinger, and Jan Mendling. Vol. 11675. Lecture Notes in Computer Science. Springer, 2019, pp. 139–156. ISBN: 978-3-030-26618-9. URL: https://doi.org/10.1007/978-3-030-26619-6_11.

[137] Kimon Batoulis. *Checking Compliance for Fragment-based Case Management*. PhD Thesis. Nov. 2019. URL: https://doi.org/10.25932/publishup-43738.

[138] Diego Calvanese, Silvio Ghilardi, Alessandro Gianola, Marco Montali, and Andrey Rivkin. "From Model Completeness to Verification of Data Aware Processes". In: *Description Logic, Theory Combination, and All That - Essays Dedicated to Franz Baader on the Occasion of His 60th Birthday*. Ed. by Carsten Lutz, Uli Sattler, Cesare Tinelli, Anni-Yasmin Turhan, and Frank Wolter. Vol. 11560. Lecture Notes in Computer Science. Springer, 2019, pp. 212–239. ISBN: 978-3-030-22101-0. URL: https://doi.org/10.1007/978-3-030-22102-7_10.

[139] Diego Calvanese, Marco Montali, Fabio Patrizi, and Andrey Rivkin. "Modeling and In-Database Management of Relational, Data-Aware Processes". In: *Advanced Information Systems Engineering - 31st International Conference, CAiSE 2019, Rome, Italy, June 3-7, 2019, Proceedings*. Ed. by Paolo Giorgini and Barbara Weber. Vol. 11483. Lecture Notes in Computer Science. Springer, 2019, pp. 328–345. ISBN: 978-3-030-21289-6. URL: https://doi.org/10.1007/978-3-030-21290-2_21.

[140] Montserrat Estañol, Jorge Munoz-Gama, Josep Carmona, and Ernest Teniente. "Conformance checking in UML artifact-centric business process models". In: *Softw. Syst. Model.* 18.4 (2019), pp. 2531–2555. URL: https://doi.org/10.1007/s10270-018-0681-6.

[141] Dirk Fahland. "Describing Behavior of Processes with Many-to-Many Interactions". In: *Application and Theory of Petri Nets and Concurrency - 40th International Conference, PETRI NETS 2019, Aachen, Germany, June 23-28, 2019, Proceedings*. Ed. by Susanna Donatelli and Stefan Haar. Vol. 11522. Lecture Notes in Computer Science. Springer, 2019, pp. 3–24. ISBN: 978-3-030-21570-5. URL: https://doi.org/10.1007/978-3-030-21571-2_1.

[142] Fernanda Gonzalez-Lopez and Luise Pufahl. "A Landscape for Case Models". In: *Enterprise, Business-Process and Information Systems Modeling - 20th International Conference, BPMDS 2019, 24th International Conference, EMMSAD 2019, Held at CAiSE 2019, Rome, Italy, June 3-4, 2019, Proceedings*. Ed. by Iris Reinhartz-Berger, Jelena Zdravkovic, Jens Gulden, and Rainer Schmidt. Vol. 352. Lecture Notes in Business Information Processing. Springer, 2019, pp. 87–102. ISBN: 978-3-030-20617-8. URL: https://doi.org/10.1007/978-3-030-20618-5_6.

[143] Adrian Holfter. *Checking Compliance for Fragment-based Case Management*. Master Thesis. July 2019.

[144] Adrian Holfter, Stephan Haarmann, Luise Pufahl, and Mathias Weske. "Checking Compliance in Data-Driven Case Management". In: *Business Process Management Workshops - BPM 2019 International Workshops, Vienna, Austria, September 1-6, 2019, Revised Selected Papers*. Ed. by Chiara Di Francescomarino, Remco M. Dijkman, and Uwe Zdun. Vol. 362. Lecture Notes in Business Information Processing. Springer, 2019, pp. 400–411. ISBN: 978-3-030-37452-5. URL: https://doi.org/10.1007/978-3-030-37453-2_33.

[145] Guosheng Kang, Liqin Yang, and Liang Zhang. "Verification of behavioral soundness for artifact-centric business process model with synchronizations". In: *Future Gener. Comput. Syst.* 98 (2019), pp. 503–511. URL: https://doi.org/10.1016/j.future.2019.03.012.

[146] Iris Reinhartz-Berger, Jelena Zdravkovic, Jens Gulden, and Rainer Schmidt, eds. *Enterprise, Business-Process and Information Systems Modeling - 20th International Conference, BPMDS 2019, 24th International Conference, EMMSAD 2019, Held at CAiSE 2019, Rome, Italy, June 3-4, 2019, Proceedings*. Vol. 352. Lecture Notes in Business Information Processing. Springer, 2019. ISBN: 978-3-030-20617-8. URL: https://doi.org/10.1007/978-3-030-20618-5.

[147] Sebastian Steinau, Andrea Marrella, Kevin Andrews, Francesco Leotta, Massimo Mecella, and Manfred Reichert. "DALEC: a framework for the systematic evaluation of data-centric approaches to process management software". In: *Softw. Syst. Model.* 18.4 (2019), pp. 2679–2716. URL: https://doi.org/10.1007/s10270-018-0695-0.

[148] Sheila Katherine Venero, Júlio Cesar dos Reis, Leonardo Montecchi, and Cecília Mary Fischer Rubira. "Towards a metamodel for supporting decisions in knowledge-intensive processes". In: *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing, SAC 2019, Limassol, Cyprus, April 8-12, 2019*. Ed. by Chih-Cheng Hung and George A. Papadopoulos. ACM, 2019, pp. 75–84. ISBN: 978-1-4503-5933-7. URL: https://doi.org/10.1145/3297280.3297290.

[149] Mathias Weske. *Business Process Management—Concepts, Languages, Architectures, Third Edition*. Springer, 2019. ISBN: 978-3-662-59431-5. URL: https://doi.org/10.1007/978-3-662-59432-2.

[150] Wil M. P. van der Aalst and Alessandro Berti. "Discovering Object-centric Petri Nets". In: *Fundam. Informaticae* 175.1-4 (2020), pp. 1–40. URL: https://doi.org/10.3233/FI-2020-1946.

[151] Witold Abramowicz and Gary Klein, eds. *Business Information Systems - 23rd International Conference, BIS 2020, Colorado Springs, CO, USA, June 8-10, 2020, Proceedings*. Vol. 389. Lecture Notes in Business Information Processing. Springer, 2020. ISBN: 978-3-030-53336-6. URL: https://doi.org/10.1007/978-3-030-53337-3.

[152] Amine Abbad Andaloussi, Andrea Burattin, Tijs Slaats, Ekkart Kindler, and Barbara Weber. "On the declarative paradigm in hybrid business process representations: A conceptual framework and a systematic literature study". In: *Inf. Syst.* 91 (2020), p. 101505. URL: https://doi.org/10.1016/j.is.2020.101505.

[153] Kerstin Andree, Sven Ihde, and Luise Pufahl. "Exception Handling in the Context of Fragment-Based Case Management". In: *Enterprise, Business-Process and Information Systems Modeling - 21st International Conference, BPMDS 2020, 25th International Conference, EMMSAD 2020, Held at CAiSE 2020, Grenoble, France, June 8-9, 2020, Proceedings*. Ed. by Selmin Nurcan, Iris Reinhartz-Berger, Pnina Soffer, and Jelena Zdravkovic. Vol. 387. Lecture Notes in Business Information Processing. Springer, 2020, pp. 20–35. ISBN: 978-3-030-49417-9. URL: https://doi.org/10.1007/978-3-030-49418-6_2.

[154] Dorina Bano and Mathias Weske. "Discovering Data Models from Event Logs". In: *Conceptual Modeling - 39th International Conference, ER 2020, Vienna, Austria, November 3-6, 2020, Proceedings*. Ed. by Gillian Dobbie, Ulrich Frank, Gerti Kappel, Stephen W. Liddle, and Heinrich C. Mayr. Vol. 12400. Lecture Notes in Computer Science. Springer, 2020, pp. 62–76. ISBN: 978-3-030-62521-4. URL: https://doi.org/10.1007/978-3-030-62522-1_5.

[155] Diego Calvanese, Silvio Ghilardi, Alessandro Gianola, Marco Montali, and Andrey Rivkin. "SMT-based verification of data-aware processes: a model-theoretic approach". In: *Math. Struct. Comput. Sci.* 30.3 (2020), pp. 271–313. URL: https://doi.org/10.1017/S0960129520000067.

[156] Silvio Ghilardi, Alessandro Gianola, Marco Montali, and Andrey Rivkin. "Petri Nets with Parameterised Data - Modelling and Verification". In: *Business Process Management - 18th International Conference, BPM 2020, Seville, Spain, September 13-18, 2020, Proceedings*. Ed. by Dirk Fahland, Chiara Ghidini, Jörg Becker, and Marlon Dumas. Vol. 12168. Lecture Notes in Computer Science. Springer, 2020, pp. 55–74. ISBN: 978-3-030-58665-2. URL: https://doi.org/10.1007/978-3-030-58666-9_4.

[157] Stephan Haarmann, Marco Montali, and Mathias Weske. "Technical Report: Refining Case Models Using Cardinality Constraints". In: *CoRR* abs/2012.02245 (2020). arXiv: 2012.02245. URL: https://arxiv.org/abs/2012.02245.

[158] Stephan Haarmann and Mathias Weske. "Correlating Data Objects in Fragment-Based Case Management". In: *Business Information Systems - 23rd International Conference, BIS 2020, Colorado Springs, CO, USA, June 8-10, 2020, Proceedings*. Ed. by Witold Abramowicz and Gary Klein. Vol. 389. Lecture Notes in Business Information Processing. Springer, 2020, pp. 197–209. ISBN: 978-3-030-53336-6. URL: https://doi.org/10.1007/978-3-030-53337-3_15.

[159] Stephan Haarmann and Mathias Weske. "Cross-Case Data Objects in Business Processes: Semantics and Analysis". In: *Business Process Management Forum - BPM Forum 2020, Seville, Spain, September 13-18, 2020, Proceedings*. Ed. by Dirk Fahland, Chiara Ghidini, Jörg Becker, and Marlon Dumas. Vol. 392. Lecture Notes in Business Information Processing. Springer, 2020, pp. 3–17. ISBN: 978-3-030-58637-9. URL: https://doi.org/10.1007/978-3-030-58638-6_1.

[160] Stephan Haarmann and Mathias Weske. "Data Object Cardinalities in Flexible Business Processes". In: *Business Process Management Workshops - BPM 2020 International Workshops, Seville, Spain, September 13-18, 2020, Revised Selected Papers*. Ed. by Adela del-Río-Ortega, Henrik Leopold, and Flávia Maria Santoro. Vol. 397. Lecture Notes in Business Information Processing. Springer, 2020, pp. 380–391. ISBN: 978-3-030-66497-8. URL: https://doi.org/10.1007/978-3-030-66498-5_28.

[161] Marcin Hewelt, Luise Pufahl, Sankalita Mandal, Felix Wolff, and Mathias Weske. "Toward a methodology for case modeling". In: *Softw. Syst. Model.* 19.6 (2020), pp. 1367–1393. URL: https://doi.org/10.1007/s10270-019-00766-5.

[162] Luise Pufahl, Sven Ihde, Michael Glöckner, Bogdan Franczyk, Björn Paulus, and Mathias Weske. "Countering Congestion: A White-Label Platform for the Last Mile Parcel Delivery". In: *Business Information Systems - 23rd International Conference, BIS 2020, Colorado Springs, CO, USA, June 8-10, 2020, Proceedings*. Ed. by Witold Abramowicz and Gary Klein. Vol. 389. Lecture Notes in Business Information Processing. Springer, 2020, pp. 210–223. ISBN: 978-3-030-53336-6. URL: https://doi.org/10.1007/978-3-030-53337-3_16.

[163] Fernanda Gonzalez-Lopez, Luise Pufahl, Jorge Munoz-Gama, Valeria Herskovic, and Marcos Sepúlveda. "Case model landscapes: toward an improved representation of knowledge-intensive processes using the fCM-language". In: *Softw. Syst. Model.* 20.5 (2021), pp. 1353–1377. URL: https://doi.org/10.1007/s10270-021-00885-y.

[164] Stephan Haarmann. "Fragment-Based Case Management Models: Metamodel, Consistency, & Correctness". In: *Proceedings of the 13th European Workshop on Services and their Composition (ZEUS 2021), Bamberg, Germany, February 25-26, 2021*. Ed. by Johannes Manner, Stephan Haarmann, Stefan Kolb, Nico Herzberg, and Oliver Kopp. Vol. 2839. CEUR Workshop Proceedings. CEUR-WS.org, 2021, pp. 1–8. URL: http://ceur-ws.org/Vol-2839/paper1.pdf.

[165] Stephan Haarmann, Adrian Holfter, Luise Pufahl, and Mathias Weske. "Formal Framework for Checking Compliance of Data-Driven Case Management". In: *J. Data Semant.* 10.1 (2021), pp. 143–163. URL: https://doi.org/10.1007/s13740-021-00120-3.

[166] Stephan Haarmann, Marco Montali, and Mathias Weske. "Refining Case Models Using Cardinality Constraints". In: *Advanced Information Systems Engineering - 33rd International Conference, CAiSE 2021, Melbourne, VIC, Australia, June 28 - July 2, 2021, Proceedings*. Ed. by Marcello La Rosa, Shazia W. Sadiq, and Ernest Teniente. Vol. 12751. Lecture Notes in Computer Science. Springer, 2021, pp. 296–310. ISBN: 978-3-030-79381-4. URL: https://doi.org/10.1007/978-3-030-79382-1_18.

[167] Stephan Haarmann, Anjo Seidel, and Mathias Weske. "Modeling Objectives of Knowledge Workers". In: *Business Process Management Workshops - BPM 2021 International Workshops, Rome, Italy, September 6-10, 2021, Accepted for Publication*. 2021.

[168] Massimiliano de Leoni, Paolo Felli, and Marco Montali. "Integrating BPMN and DMN: Modeling and Analysis". In: *J. Data Semant.* 10.1 (2021), pp. 165–188. URL: https://doi.org/10.1007/s13740-021-00132-z.

[169] Object Management Group. *Decision Model and Notation (DMN*. Version 1.3. Feb. 2021. URL: https://www.omg.org/spec/DMN.

[170] Monique Snoeck, Johannes De Smedt, and Jochen De Weerdt. "Supporting Data-Aware Processes with MERODE". In: *Enterprise, Business-Process and Information Systems Modeling - 22nd International Conference, BPMDS 2021, and 26th International Conference, EMMSAD 2021, Held at CAiSE 2021, Melbourne, VIC, Australia, June 28-29, 2021, Proceedings*. Ed. by Adriano Augusto, Asif Gill, Selmin Nurcan, Iris Reinhartz-Berger,

Rainer Schmidt, and Jelena Zdravkovic. Vol. 421. Lecture Notes in Business Information Processing. Springer, 2021, pp. 131–146. URL: https://doi.org/10.1007/978-3-030-79186-5_9.

[171] Claes Wohlin and Per Runeson. "Guiding the selection of research methodology in industry-academia collaboration in software engineering". In: *Inf. Softw. Technol.* 140 (2021), p. 106678. URL: https://doi.org/10.1016/j.infsof.2021.106678.

All links were last accessed on 2021/11/06.

Declaration

I hereby confirm that I have authored this thesis independently and without use of other than the indicated sources. All passages which are literally or in general matter taken out of publications or other sources are marked as such. I am aware of the examination regulations and this thesis has not been previously submitted elsewhere.

Potsdam, Germany
November 2021

Stephan Haarmann