

EVENT HANDLING IN BUSINESS PROCESSES
FLEXIBLE EVENT SUBSCRIPTION FOR BUSINESS PROCESS ENACTMENT

SANKALITA MANDAL

BUSINESS PROCESS TECHNOLOGY GROUP
HASSO PLATTNER INSTITUTE
DIGITAL ENGINEERING FACULTY
UNIVERSITY OF POTSDAM
POTSDAM, GERMANY

DISSERTATION
ZUR ERLANGUNG DES AKADEMISCHEN GRADES EINES
"DOCTOR RERUM NATURALIUM"
– DR. RER. NAT. –

DATE OF DEFENSE: 17/12/2019

December 2019

This work is licensed under a Creative Commons License:
Attribution – 4.0 International.

This does not apply to quoted content from other authors.

To view a copy of this license visit

<https://creativecommons.org/licenses/by/4.0/>

Supervisor: Prof. Dr. Mathias Weske, University of Potsdam

Reviewers: Prof. Dr. Matthias Weidlich, HU Berlin, and

Dr. Remco Dijkman, Eindhoven University of Technology

Sankalita Mandal: Event Handling in Business Processes,

© December 2019

Published online at the

Institutional Repository of the University of Potsdam:

<https://doi.org/10.25932/publishup-44170>

<https://nbn-resolving.org/urn:nbn:de:kobv:517-opus4-441700>

ABSTRACT

Business process management (BPM) deals with modeling, executing, monitoring, analyzing, and improving business processes. During execution, the process communicates with its environment to get relevant contextual information represented as *events*. Recent development of big data and the Internet of Things (IoT) enables sources like smart devices and sensors to generate tons of events which can be filtered, grouped, and composed to trigger and drive business processes.

The industry standard Business Process Model and Notation (BPMN) provides several event constructs to capture the interaction possibilities between a process and its environment, e.g., to instantiate a process, to abort an ongoing activity in an exceptional situation, to take decisions based on the information carried by the events, as well as to choose among the alternative paths for further process execution. The specifications of such interactions are termed as *event handling*. However, in a distributed setup, the event sources are most often unaware of the status of process execution and therefore, an event is produced irrespective of the process being ready to consume it. BPMN semantics does not support such scenarios and thus increases the chance of processes getting delayed or getting in a deadlock by missing out on event occurrences which might still be relevant.

The work in this thesis reviews the challenges and shortcomings of integrating real-world events into business processes, especially the *subscription management*. The basic integration is achieved with an architecture consisting of a process modeler, a process engine, and an event processing platform. Further, *points of subscription and unsubscription* along the process execution timeline are defined for different BPMN event constructs. Semantic and temporal dependencies among event subscription, event occurrence, event consumption and event unsubscription are considered. To this end, an *event buffer* with policies for updating the buffer, retrieving the most suitable event for the current process instance, and reusing the event has been discussed that supports issuing of early subscription.

The Petri net mapping of the event handling model provides our approach with a translation of semantics from a business process perspective. Two applications based on this formal foundation are presented to support the significance of different event handling configurations on correct process execution and reachability of a process path. Prototype implementations of the approaches show that realizing flexible event handling is feasible with minor extensions of off-the-shelf process engines and event platforms.

ZUSAMMENFASSUNG

Das Prozessmanagement befasst sich mit der Modellierung, Ausführung, Überwachung, Analyse und Verbesserung von Geschäftsprozessen. Während seiner Ausführung kommuniziert der Prozess mit seiner Umgebung, um relevante Kontextinformationen in Form von *Ereignissen* zu erhalten. Der jüngste Fortschritt im Bereich Big Data und dem Internet der Dinge ermöglicht Smart Devices und Sensoren eine Vielzahl von Ereignissen zu generieren, welche gefiltert, gruppiert und kombiniert werden können, um Geschäftsprozesse zu triggern und voranzutreiben.

Der Industriestandard Business Process Model and Notation (BPMN) stellt mehrere Ereigniskonstrukte bereit, um die Interaktionsmöglichkeiten eines Prozesses mit seiner Umgebung zu erfassen. Beispielsweise können Prozesse durch Ereignisse gestartet, laufende Aktivitäten in Ausnahmefällen abgebrochen, Entscheidungen auf Basis der Ereignisinformationen getroffen, und alternative Ausführungspfade gewählt werden. Die Spezifikation solcher Interaktionen wird als *Event Handling* bezeichnet. Allerdings sind sich insbesondere in verteilten Systemen die Ereignisquellen des Zustands des Prozesses unbewusst. Daher werden Ereignisse unabhängig davon produziert, ob der Prozess bereit ist sie zu konsumieren. Die BPMN-Semantik sieht solche Situationen jedoch nicht vor, sodass die Möglichkeit besteht, dass das Auftreten von relevanten Ereignissen versäumt wird. Dies kann zu Verzögerungen oder gar Deadlocks in der Prozessausführung führen.

Die vorliegende Dissertation untersucht die Mängel und Herausforderungen der Integration von Ereignissen und Geschäftsprozessen, insbesondere in Bezug auf das *Subscription Management*. Die grundlegende Integration wird durch eine Architektur erreicht, die aus einer Prozessmodellierungskomponente, einer Ausführungskomponente und einer Ereignisverarbeitungskomponente besteht. Ferner werden *Points of Subscription and Unsubscription* für verschiedene BPMN-Ereigniskonstrukte entlang der Zeitachse der Prozessausführung definiert. Semantische und temporale Abhängigkeiten zwischen der Subscription, dem Auftreten, dem Konsumieren und der Unsubscription eines Ereignisses werden betrachtet. In dieser Hinsicht wird ein *Event Buffer* diskutiert, welcher mit Strategien zum Update des Puffers, zum Abruf der geeigneten Ereignisse für den laufenden Prozess, sowie zur Wiederverwendung von Ereignissen ausgestattet ist.

Die Abbildung des Event Handling Modells in ein Petri-Netz versieht den beschriebenen Ansatz mit einer eindeutigen Semantik. Basierend auf diesem Formalismus werden zwei Anwendungen demonstriert, die die Relevanz verschiedener Konfigurationen des Event Handlings für

eine korrekte Prozessausführung aufzeigen. Eine prototypische Implementierung des Ansatzes beweist dessen Umsetzbarkeit durch geringe Erweiterungen bestehender Software zur Prozessausführung und Ereignisverarbeitung.

PUBLICATIONS

The supporting publications for the research work presented in this thesis are:

- Sankalita Mandal and Mathias Weske. “A Flexible Event Handling Model for Business Process Enactment.” In: *22nd IEEE International Enterprise Distributed Object Computing Conference, EDOC*. IEEE Computer Society, 2018, pp. 68–74. DOI: [10.1109/EDOC.2018.00019](https://doi.org/10.1109/EDOC.2018.00019). URL: <https://doi.org/10.1109/EDOC.2018.00019>.
- Sankalita Mandal, Matthias Weidlich, and Mathias Weske. “Events in Business Process Implementation: Early Subscription and Event Buffering.” In: *Business Process Management Forum - BPM Forum*. Vol. 297. Lecture Notes in Business Information Processing. Springer, 2017, pp. 141–159. DOI: [10.1007/978-3-319-65015-9_9](https://doi.org/10.1007/978-3-319-65015-9_9). URL: https://doi.org/10.1007/978-3-319-65015-9_9.
- Sankalita Mandal, Marcin Hewelt, and Mathias Weske. “A Framework for Integrating Real-World Events and Business Processes in an IoT Environment.” In: *On the Move to Meaningful Internet Systems. OTM 2017 Conferences - Confederated International Conferences: CoopIS, C&TC, and ODBASE, Proceedings, Part I*. Vol. 10573. Lecture Notes in Computer Science. Springer, 2017, pp. 194–212. DOI: [10.1007/978-3-319-69462-7_13](https://doi.org/10.1007/978-3-319-69462-7_13). URL: https://doi.org/10.1007/978-3-319-69462-7_13.
- Sankalita Mandal. “A Flexible Event Handling Model for Using Events in Business Processes.” In: *Proceedings of the 9th International Workshop on Enterprise Modeling and Information Systems Architectures*. Vol. 2097. CEUR Workshop Proceedings. CEUR-WS.org, 2018, pp. 11–15. URL: <http://ceur-ws.org/Vol-2097/paper2.pdf>
- Luise Pufahl, Sankalita Mandal, Kimon Batoulis, and Mathias Weske. “Re-evaluation of Decisions Based on Events.” In: *Enterprise, Business-Process and Information Systems Modeling - 18th International Conference, BPMDS*. Vol. 287. Lecture Notes in Business Information Processing. Springer, 2017, pp. 68–84. DOI: [10.1007/978-3-319-59466-8_5](https://doi.org/10.1007/978-3-319-59466-8_5). URL: https://doi.org/10.1007/978-3-319-59466-8_5.
- Sankalita Mandal, Marcin Hewelt, Maarten Östreich, and Mathias Weske. “A Classification Framework for IoT Scenarios.” In: *Business Process Management Workshops - BPM 2018 International Workshops*. Vol. 342. Lecture Notes in Business Information Processing.

Springer, 2018, pp. 458–469. DOI: [10.1007/978-3-030-11641-5_36](https://doi.org/10.1007/978-3-030-11641-5_36). URL: https://doi.org/10.1007/978-3-030-11641-5_36.

- Sankalita Mandal. “Events in BPMN: The Racing Events Dilemma.” In: *Proceedings of the 9th Central European Workshop on Services and their Composition (ZEUS)*. Vol. 1826. CEUR Workshop Proceedings. CEUR-WS.org, 2017, pp. 23–30. URL: <http://ceur-ws.org/Vol-1826/paper5.pdf>.
- Maximilian Völker, Sankalita Mandal, and Marcin Hewelt. “Testing Event-driven Applications with Automatically Generated Events.” In: *Proceedings of the BPM Demo Track and BPM Dissertation Award co-located with 15th International Conference on Business Process Management*. Vol. 1920. CEUR Workshop Proceedings. CEUR-WS.org, 2017. URL: http://ceur-ws.org/Vol-1920/BPM_2017_paper_182.pdf.
- 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*. Vol. 1789. CEUR Workshop Proceedings. CEUR-WS.org, 2016, pp. 67–72. <http://ceur-ws.org/Vol-1789/bpm-demo-2016-paper13.pdf>.

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

- 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*. Vol. 318. Lecture Notes in Business Information Processing. Springer, 2018, pp.181–195. DOI: [10.1007/978-3-319-91704-7_12](https://doi.org/10.1007/978-3-319-91704-7_12). URL: https://doi.org/10.1007/978-3-319-91704-7_12.
- Adriatik Nikaj, Sankalita Mandal, Cesare Pautasso, and Mathias Weske. “From Choreography Diagrams to RESTful Interactions.” In: *Service-Oriented Computing - ICSOC 2015 Workshops - WESOA*. Vol. 9586. Lecture Notes in Computer Science. Springer, 2015, pp. 3–14. DOI: [10.1007/978-3-662-50539-7_1](https://doi.org/10.1007/978-3-662-50539-7_1). URL: https://doi.org/10.1007/978-3-662-50539-7_1.

ACKNOWLEDGMENTS

The journey of a PhD thesis takes lot more than just the scientific contributions promised in it. The past four years made me stronger as a researcher, as a colleague, and also as a person. I always see my life as an expanding collage of experiences, and now that I look back, I realize there are so many people who added colors to it!

Speaking of the researcher life, I am so glad that I ended up being a PhD student under Prof. Mathias Weske's supervision. I appreciate how you give us the freedom of finding our interest areas while guiding us to take the right decisions. I have been privileged to be a part of the BPT ecosystem that you drive so efficiently. At the same time, being part of the HPI research school was a great exposure to the work being done in other streams of computer science.

I am grateful that Prof. Matthias Weidlich and Dr. Remco Dijkman have agreed to be my reviewers. Both of their work influenced my research significantly. Matthias, you have always been my favorite. Thank you for all the long valuable discussions, collaborating with you has been truly a nice learning experience.

A special credit goes to Anne Baumgraß for triggering my interest in complex event processing and being a skilled mentor for the initial months. Later, I shared the major part of my research and teaching activities with Marcin Hewelt. Thanks Marcin for always answering my questions with a smile. This brings me to express my gratitude to all the previous BPT members who have shared some part of their academic stay with me and gave me a warm welcome to the group.

A big applause to my current colleagues for making time to proof-read my thesis chapters. Luise, Stephan, Kiarash, Adrian, Sven, Jan, Marcin, Simon – thanks for all the thoughtful feedback that enriched the content of this work. You guys are amazing, not only for being smart researchers, but also for being so humorous and supportive and patient with my random crazy stories. I would like to give additional courtesy to my 'Allround Enhancement Executive' Stephan, who has always been there to help, be it checking a formalism or correcting an email in German; and my 'Chief Aesthetics Counselor' Jan, who volunteered to make this thesis look prettier and also designed the cover.

And then there are Adriatik and Kimon, coinciding the writing phase with whom has been absolutely blissful for sharing the excitement of finally finishing the PhD, and more than that, for sharing the huge amount of stress that suppresses that excitement! Kimon, thanks a lot for all those long hours of working together, for all the help with scientific and paper work, and for all your unique ways of motivating when I felt low.

The journey of completing the PhD would not have been so accomplishing without being able to bring my parents over for the first time in Europe and witness them being proud while visiting my workplace. My musical family including my guitarists and co-artists, thanks for being my refuge. Lastly, all my wonderful friends across Germany and back in India, thanks for being on board through good and bad times.

CONTENTS

| | | |
|-----------|---|-----------|
| I | INTRODUCTION & FOUNDATIONS | 1 |
| 1 | INTRODUCTION | 3 |
| 1.1 | Problem Statement | 4 |
| 1.2 | Research Objectives | 6 |
| 1.3 | Contributions | 8 |
| 1.4 | Structure of Thesis | 10 |
| 2 | BUSINESS PROCESS MANAGEMENT | 13 |
| 2.1 | BPM Life Cycle | 13 |
| 2.2 | Business Process Model and Notation | 15 |
| 2.2.1 | Core Elements | 15 |
| 2.2.2 | Events in Business Processes | 18 |
| 2.3 | Petri Nets | 20 |
| 2.4 | BPMN to Petri Net Mapping | 22 |
| 3 | COMPLEX EVENT PROCESSING | 27 |
| 3.1 | Basic Concepts | 27 |
| 3.2 | Event Distribution | 29 |
| 3.3 | Event Abstraction | 31 |
| 3.4 | Event Processing Techniques | 34 |
| 4 | RELATED WORK | 41 |
| 4.1 | Overview | 41 |
| 4.2 | External Events in BPM | 42 |
| 4.3 | Integrated Applications | 44 |
| 4.4 | Flexible Event Subscription & Buffering | 47 |
| 4.5 | Summary | 49 |
| II | CONCEPTUAL FRAMEWORK | 51 |
| 5 | INTEGRATING REAL-WORLD EVENTS INTO BUSINESS PROCESS EXECUTION | 53 |
| 5.1 | Motivation & Overview | 53 |
| 5.2 | Requirements Analysis | 54 |
| 5.2.1 | R1: Separation of Concerns | 55 |
| 5.2.2 | R2: Representation of Event Hierarchies | 56 |
| 5.2.3 | R3: Implementation of Integration | 57 |
| 5.3 | System Architecture | 57 |
| 5.3.1 | Distribution of Logic | 58 |
| 5.3.2 | Use of Event Abstraction | 59 |
| 5.3.3 | Implementation Concepts | 61 |
| 5.4 | Summary & Discussion | 63 |
| 6 | FLEXIBLE EVENT HANDLING MODEL | 65 |
| 6.1 | Motivation & Overview | 65 |
| 6.2 | Event Handling Notions | 69 |

| | | |
|-------|--|-----|
| 6.2.1 | Business Process View | 69 |
| 6.2.2 | Event Processing View | 73 |
| 6.3 | Flexible Subscription Management | 74 |
| 6.3.1 | Points of Subscription | 75 |
| 6.3.2 | Points of Unsubscription | 77 |
| 6.3.3 | Event Buffering | 78 |
| 6.3.4 | Semantic Interdependencies | 80 |
| 6.4 | Summary & Discussion | 81 |
| 7 | FORMAL EXECUTION SEMANTICS | 83 |
| 7.1 | Motivation & Overview | 83 |
| 7.2 | Petri Net Mapping | 85 |
| 7.2.1 | Event Handling Notions | 85 |
| 7.2.2 | Points of Subscription | 86 |
| 7.2.3 | Points of Unsubscription | 91 |
| 7.2.4 | Event Buffering | 93 |
| 7.3 | Summary & Discussion | 96 |
| III | EVALUATION & CONCLUSIONS | 99 |
| 8 | APPLICATION OF CONCEPTS | 101 |
| 8.1 | Execution Trace Analysis | 101 |
| 8.1.1 | Correctness Constraints | 101 |
| 8.1.2 | Impact of Event Handling | 105 |
| 8.1.3 | Discussion | 106 |
| 8.2 | Reachability Analysis | 107 |
| 8.2.1 | Communication Model | 108 |
| 8.2.2 | Impact of Event Handling | 110 |
| 8.2.3 | Discussion | 112 |
| 8.3 | Summary | 114 |
| 9 | PROOF-OF-CONCEPT IMPLEMENTATION | 115 |
| 9.1 | Basic Event Interaction | 115 |
| 9.1.1 | Unicorn Event Processing Platform | 116 |
| 9.1.2 | Gryphon Case Modeler | 117 |
| 9.1.3 | Chimera Process Engine | 119 |
| 9.1.4 | Event Integration Sequence | 120 |
| 9.2 | Flexible Event Subscription with Buffering | 122 |
| 9.2.1 | BPMN Extension | 123 |
| 9.2.2 | Unicorn Extension | 127 |
| 9.2.3 | Camunda Extension | 129 |
| 9.3 | Summary | 132 |
| 10 | CONCLUSIONS | 133 |
| 10.1 | Summary of Thesis | 133 |
| 10.2 | Limitations and Future Research | 135 |
| | BIBLIOGRAPHY | 139 |

LIST OF FIGURES

| | | |
|-----------|--|----|
| Figure 1 | Overview of research steps for the thesis. | 7 |
| Figure 2 | Relevance of Information Systems research [19]. | 8 |
| Figure 3 | Summary of research contributions. | 11 |
| Figure 4 | Business process management (BPM) lifecycle. | 14 |
| Figure 5 | State transition diagram for activity lifecycle. | 16 |
| Figure 6 | BPMN tasks. | 16 |
| Figure 7 | BPMN gateways. | 17 |
| Figure 8 | BPMN events, as presented in BPMN 2.0 Poster by BPM Offensive Berlin (cf. http://www.bpmb.de/index.php/BPMNPoster). | 19 |
| Figure 9 | Mapping of BPMN task to Petri net modules. | 23 |
| Figure 10 | Mapping of BPMN gateways to Petri net modules. | 23 |
| Figure 11 | Mapping of BPMN event constructs to Petri net modules. | 24 |
| Figure 12 | Mapping of BPMN boundary event construct to Petri net module. | 24 |
| Figure 13 | Event processing network. | 28 |
| Figure 14 | Data in rest vs. data in motion. | 29 |
| Figure 15 | Communication models for event distribution. | 30 |
| Figure 16 | Applications of event abstraction in context of process execution. | 32 |
| Figure 17 | Event abstraction for order tracking in an online shop. | 33 |
| Figure 18 | Stream processing operations on events. | 33 |
| Figure 19 | State changes for an event query in an event processing platform. | 34 |
| Figure 20 | An example EPN showing the different components. | 35 |
| Figure 21 | Event hierarchy showing Transaction and events derived from it. | 36 |
| Figure 22 | Screenshot from Unicorn showing generation of complex events following pre-defined rules. | 40 |
| Figure 23 | Overview of discussed research areas as related work. | 41 |
| Figure 24 | Usecase from the logistics domain | 55 |

| | |
|-----------|---|
| Figure 25 | Proposed system architecture for event-process integration. 58 |
| Figure 26 | Event subscription and correlation within a process engine and an event platform. 62 |
| Figure 27 | Collaboration diagram motivating the need for flexible subscription. 66 |
| Figure 28 | Dependencies among the event handling notions: event subscription, occurrence, consumption, and unsubscription. 69 |
| Figure 29 | Event construct classification put in example process models. 71 |
| Figure 30 | Lifecycle of an event w.r.t. requisite for a process execution. 73 |
| Figure 31 | Interaction from event processing platform's perspective. 74 |
| Figure 32 | Process execution timeline. 75 |
| Figure 33 | Points of (Un)-Subscription. 78 |
| Figure 34 | Interdependencies among the aspects of flexible event handling. 80 |
| Figure 35 | Event handling notions represented by Petri net. 86 |
| Figure 36 | Scenarios considering the separate notions of event occurrence and event consumption. 86 |
| Figure 37 | Petri net modules for mandatory event construct. 87 |
| Figure 38 | Petri net module for boundary event construct. 88 |
| Figure 39 | Petri net modules for racing event construct. 90 |
| Figure 40 | Petri net modules for exclusive event construct. 91 |
| Figure 41 | Petri net modules for points of unsubscription. 92 |
| Figure 42 | Petri net showing event matching, correlation, and buffering. 94 |
| Figure 43 | Petri net showing a <i>mandatory</i> event construct with <i>subscription at process instantiation</i> and <i>unsubscription at event consumption</i> . 102 |
| Figure 44 | Excerpt from motivating example presented in Figure 27 in Chapter 6 . 106 |
| Figure 45 | Petri net with <i>subscription at event enablement</i> and <i>unsubscription at event consumption</i> . 106 |
| Figure 46 | Petri net with <i>subscription at process instantiation</i> and <i>unsubscription at event consumption</i> . 107 |
| Figure 47 | Transition diagram for a process model (\mathfrak{M}) and corresponding environment models ($\mathfrak{E}, \mathfrak{E}'$). 109 |
| Figure 48 | Different subscription configurations for receiving event z . 110 |

- Figure 49 Process Model \mathfrak{M} communicating with environment. 110
- Figure 50 A chosen path in process and the set of corresponding paths in environment with varying *subscription configuration*. 111
- Figure 51 A chosen path in process and the set of corresponding paths in environment with varying *consumption configuration*. 111
- Figure 52 A chosen path in the process and the set of corresponding paths in environment with *early subscription* for z and w . 113
- Figure 53 BPMN process model with boundary event and the corresponding transition system. 113
- Figure 54 Detailed system architecture showing specific components and the sequence of integrating events into processes. 115
- Figure 55 Architecture of event processing platform Unicorn. 116
- Figure 56 Modeling of subscription queries with extended field (in right) for event annotation in Gryphon. 118
- Figure 57 Event data from LongDelay is written into a newly created data object Delay using a JSON path expression. 119
- Figure 58 Architecture of the Chimera process engine. 120
- Figure 59 The sequence of communication between Gryphon, Chimera, and Unicorn for the integration of events into processes. 121
- Figure 60 BPMN+X model showing extension of BPMN *Message* element. 124
- Figure 61 An example process with embedded subscription definition for the intermediate catching message event. 126
- Figure 62 Extended architecture for flexible event handling. 127
- Figure 63 UML Class diagram of Camunda process engine plugin. 130

LISTINGS

| | | |
|------------|---|-----|
| Listing 1 | Event type definition of Transaction | 35 |
| Listing 2 | Example of Project EPA | 37 |
| Listing 3 | Example of Translate EPA | 37 |
| Listing 4 | Example of Enrich EPA | 37 |
| Listing 5 | Example of Aggregation EPA | 38 |
| Listing 6 | Event types Withdrawal and FraudAlert | 38 |
| Listing 7 | Example of Compose EPA | 39 |
| Listing 8 | Event type definitions for motivating example | 60 |
| Listing 9 | Event abstraction pattern for LongDelay | 60 |
| Listing 10 | Example of event patterns | 74 |
| Listing 11 | XML interpretation of BPMN+X model | 124 |
| Listing 12 | Excerpt from the XML interpretation of the BPMN process modeled in Fig. 59 showing process structure and extended elements enabling flexible subscription | 126 |

ACRONYMS

| | |
|------|-------------------------------------|
| BPM | Business Process Management |
| CEP | Complex Event Processing |
| IoT | Internet of Things |
| BPEL | Business Process Execution Language |
| BPMN | Business Process Model and Notation |
| EPC | Event-driven Process Chain |
| EPA | Event Processing Agents |
| EPN | Event Processing Network |
| EPS | Event Processing Systems |
| DBMS | Database Management Systems |
| SEP | Simple Event Processing |
| ESP | Event Stream Processing |
| PN | Petri Net |
| SQL | Structured Query Language |
| JMS | Java Message Service |
| REST | Representational State Transfer |
| CQL | Continuous Query Language |
| IS | Information Systems |
| CPN | Coloured Petri Nets |
| EPP | Event Processing Platform |
| ERP | Enterprise Resource Planning |
| UUID | Universally Unique Identifier |
| EIP | Enterprise Integration Patterns |

Part I

INTRODUCTION & FOUNDATIONS

INTRODUCTION

“The event concept is simple yet powerful.”
– Etzion & Niblett, *Event Processing in Action* [46]

Medical science says, *every second of every day, our senses bring in way too much data than we can possibly process in our brains*¹. The situation for software systems is similar in the era of the Internet of Things (IoT), since now we have the technological advancements to translate the physical senses into digital signals [9]. Millions of sensors are producing a huge amount of events that carry data or information which can be interpreted to gain insight about the environmental occurrences. We are already familiar with the concepts of “big data explosion” [65] and “data being the new oil” [124]. However, there is a subtle yet very significant paradigm shift in terms of processing data. Instead of the static data stored in large databases, the last decade focused more on the dynamic data or data in motion. Rather than using the data afterwards for analyzing what happened in the past, the events give us insight about the current situation or even predict the future, so that we can react to the environmental occurrences in a timely manner [74]. But while the amount of available information is rapidly increasing, the time window during which the information is relevant, i.e., the self-life of the information is decreasing, so is the reaction time [102].

Research fields such as data science, data engineering, and Complex Event Processing (CEP) explore concepts and technicalities for *extracting insight from data*. The event processing platforms connect to event sources to receive streams of events, and perform range of operations to have a meaningful interpretation of the data carried by the events [56]. Often, put together a few events occurred in a specific pattern, higher level business information are derived that influence organizational processes. For instance, a sudden fall in stock market prices might demand a company to postpone a release of their new product. Another example could be an airlines offering the passengers alternative connections after there is a strike at a specific airport. When it comes to organizational processes, Business Process Management (BPM) is the field of research that deals with the overall support for modeling, executing, monitoring, evaluating, and improving them [134].

Given the situation, it is highly important to get access to the relevant information that can influence a business process, and to take the necessary steps as soon as a certain circumstance has occurred. Such external influences are represented as *event constructs* in a business process. Business Process Model and Notation (BPMN) [94] is the industry

¹ https://www.brainyquote.com/quotes/peter_diamandis_488485

standard for modeling and executing business processes. BPMN is a highly expressive language and provides several event constructs to capture the interaction possibilities between a process and its environment. According to the standard, events can be used to instantiate a process, to abort an ongoing activity in an exceptional situation, to take decisions based on the information carried by the events, as well as to choose among the alternative paths for further process execution. On one hand, a process can receive these events from the event sources distributed in the environment. On the other hand, the process can also produce events to send a message or a signal to the environment [27]. The specifications of such interactions are termed as *event handling*.

Despite having the provisions for integrating external information into processes, so far there is no clear guideline for event handling from a business process perspective, such as how to receive a business level event from the raw event sources, when to start listening to an external occurrence, and how long a particular event is relevant for process execution. Research gaps for an efficient communication between events and processes are outlined in detail in [Section 1.1](#). This brings the opportunity to explain how those questions are addressed in this work, described as the research objectives in [Section 1.2](#). Next, the contributions of this thesis are listed in [Section 1.3](#). Lastly, the structure of the thesis is sketched in [Section 1.4](#).

1.1 PROBLEM STATEMENT

Business process models incorporate environmental happenings in the form of external events. These events play a significant role in process enactment — they can drive the process execution flow, and can provide information for decision making. Standard notations and languages for modeling and executing business processes provide event constructs to capture the communication with environment. Business Process Execution Language (BPEL) [92] has been popular in the BPM community for last two decades. In BPEL, events are either messages received (or sent) by activities from (to) other webservices, or timers. BPEL's *<receive>* activity defines an Endpoint (URL, Port, operation) that needs to be invoked to receive an event. *<onAlarm>* activity waits for a duration of time or until a specified point in time [72].

While BPEL has a narrow range of events (message, receive, and timer), the de facto standard BPMN offers a much wider understanding of events [130]. Happenings in the world might impact a business process in various ways, rather than just a message sent to the process instance, or a timer based alarm. To accommodate that, BPMN defines events that can be placed as a start event to instantiate a process, as an intermediate event to represent communication with other process participants, and as an end event to signify the completion of the process. Also, several event types are specified to capture different semantics of

the communication, e.g., broadcasting a news to all process participants and representing an error state. These event constructs are visualized as dedicated nodes in process models, abstracting from the technicalities such as subscribing to the event source, receiving it during process execution, and consuming the information carried by it.

However, BPEL and BPMN both completely ignore the details about the event sources and event abstraction hierarchy that are significant for successfully receiving the required information in the processes. Other languages such as UML Activity Diagrams [95] also neglect the above issues. Standard process engines like Camunda [24] support BPMN events for starting a process instance and for selecting between alternative paths following a gateway. Nevertheless, even in the execution level, the engines do not care about the receiving part of the message event. For instance, Camunda has interfaces that can be connected to a Java Message Service (JMS) queue² or a Representational State Transfer (REST) interface [105]; but the reception of messages is not implemented.

Lack of support for event handling in BPM

Event handling details the specification of how a process interacts with its environment and how the environmental occurrences impact the process execution. Considering the semantics for event handling, especially the guidelines with respect to subscription for intermediate catching events, BPMN specification [94] states:

‘For Intermediate Events, the handling consists of waiting for the Event to occur. Waiting starts when the Intermediate Event is reached. Once the Event occurs, it is consumed. Sequence Flows leaving the Event are followed as usual.’ [BPMN 2.0, Sect. 13.4.2]

That is, when the control-flow reaches the event construct, it is enabled and a process instance starts waiting for the event to happen. Once it happens, the control-flow is passed down to next activities. As a consequence, a process instance may not react to an event that occurred *earlier* than its control-flow reached the respective event construct. The assumption that “an event occurs only when the process is ready to consume it” is severely restricting in various real business scenarios. Especially in a distributed setup or in an IoT environment, where the event sources are not aware of the current execution status of the process, it might cause the process to get delayed by waiting for the next occurrence of the event. Even worse, if the event is not published regularly then once the process misses the event occurrence, the process execution gets in a deadlock.

Taking into account the fact that creation of event by the environment is decoupled from process execution status, existing event handling semantics raise the following research questions:

- *RQ1: When to subscribe to an event source?*

This of course depends on the data availability that is needed

² <https://www.oracle.com/technetwork/articles/java/introjms-1577110.html>

to create a subscription. But the necessary information can be available at different points of time during process execution, earlier than enablement of the event construct. Also, instance specific data might not be needed for subscribing to certain events. Therefore, flexible yet unambiguous semantics enabling early subscription are needed to specify when it is feasible and when it is recommended to start listening to a particular event occurrence.

- *RQ2: For how long to keep the subscription?*

After an event is consumed, the process instance does not need it any more. But there might be other points in time when the event loses the relevance to the process instance even before it is consumed. Hence, possibilities and need for an unsubscription is worth discussing in the context of early subscription.

- *RQ3: How to store and retrieve the relevant events for a specific process execution?*

Early subscription calls for storing the events until the process is ready to consume it. But if by the time there are more than one occurrences of a certain event type, then the following aspects have to be decided.

- How many events should be stored of the same type?
- Which one of them is the most relevant for a specific instance?
- Is the event information reusable?

To summarize, most process specification languages share the same limitations described above. The lack of flexibility in event handling semantics limits the interaction scope between a process and its environment. The research objectives to address these issues are described next.

1.2 RESEARCH OBJECTIVES

The goal of the thesis is to provide a formal event handling model for business process enactment. The event handling model shall facilitate flexible event subscription to fit the need of real-world distributed environment where event sources send information to business process management systems without knowing the internal process execution status.

Essentially, the event handling model defines the notions of a business process creating a *subscription* to start listening to an event, the *occurrence* of the event that is relevant for the process, *consumption* of the event following the process control-flow, and *unsubscription* to stop receiving the event notification. The subscription and unsubscription can be done at several milestones along the process execution timeline. The business-level event can either directly be produced as a raw event

by an event source, or can be an aggregated one generated using event abstraction hierarchies by an event processing platform. The process can consume the event as soon as it occurs, given there is a subscription issued before, or the event can be temporarily stored in a buffer until the process execution is ready to consume it. The event handling notions and their interdependencies shall be defined explicitly.

The aim is to come up with a platform independent formal model that considers the separate event handling notions instead of abstracting the details in an event construct. This gives unambiguous and clear semantics for correct process execution while considering event-process interactions. The concepts of *early subscription* and *event buffering* are intended to leverage an extended timespan of using relevant events for a process and thereby mitigating the possibility of the process getting delayed or getting stuck while waiting for the event occurrence. The objective of highlighting different possible *points of (un)-subscription* is to enhance the flexibility of the event-process interplay.

The flow of building up concepts and technicalities during the thesis journey follows an explorative path, as shown in [Figure 1](#). The goal of having a detailed event handling model raised the demand to understand the role of complex events in business processes. Therefore, extensive literature study in the area of using complex event processing in business process management has been carried out. Along with building up the knowledge base, some hands on applications are also developed. For instance, using complex event processing to enable re-evaluation of decisions based on context information is explored in [101]. Integration challenges such as impact of probable latency between the occurrence time of an event and detection time of that event in the process execution engine has been discussed in [78].



Figure 1: Overview of research steps for the thesis.

The investigation of state-of-the-art revealed that business process management (BPM) and complex event processing (CEP) are well established fields as individual research areas; however, an integration framework of the two worlds is missing. This led to basic and advanced understanding of the challenges of integrating external events into business processes, both in process design level, and process execution level. Thus, the next research phase has been dedicated to build an end-to-end solution that integrates external events into business processes [80]. While the integrated architecture met the basic requirements, it was not enough to capture the flexibility needed by real-world scenarios present in distributed setups. In the next research phase, concepts for early event subscription and event buffering [82] are introduced to induce the needed flexibility to communicate with the event sources. The

final phase, flexible event handling with Petri net mapping, extends the concept of early subscription and adds formal semantics to the event handling notions from a business process perspective.

1.3 CONTRIBUTIONS

In Information Systems (IS) research, it is important for a contribution to be relevant. According to Benbasat and Zmud [19], a research content is relevant when it fulfills the following three criteria:

- *Interesting*: whether the research address the challenges that are of concern to IS professionals,
- *Current*: whether the research considers state-of-the-art technologies and business issues, and
- *Applicable*: whether the research is utilizable by practitioners.

Figure 2 summarizes the criteria for a relevant research in IS. Additionally, the research should be *accessible* to the IS professionals, written in a well understandable and fluent style.

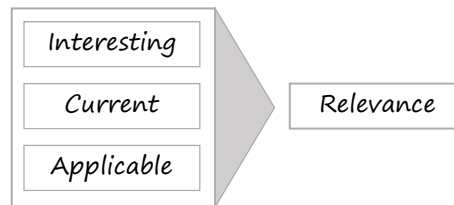


Figure 2: Relevance of Information Systems research [19].

Keeping that in mind, the research contributions added by this thesis are argued to be *relevant* while outlining the highlights in the following.

- *Requirements Analysis*. Based on the current status of the BPMN specification, the standard process engines, and with the help of domain experts, we present the requirements for an integrated architecture. The need for flexible event subscription points are detailed based on a usecase. The requirements elicitation is done in close collaboration with researchers as well as business professionals from the field. The integration challenges have been agreed to be of concern to all of them, making the contribution *interesting*.
- *Integrated Framework*. Addressing the requirements, an integrated architecture enabling efficient communication between external events and business processes are established. The framework fulfills the conceptual requirements such as separation of concerns between event processing logic and business process specification, exploiting event abstraction hierarchies to map raw-level events to higher-level business events. It also considers the technical requirements such as subscribing to an event source, receiving

events, and reacting to them. The generic architecture is realized with a process editor, a process engine, and an event processing platform as a proof-of-concept implementation.

- *Flexible Event Handling Model*. This is the main contribution of this dissertation. The work done in the course of this thesis introduces the concepts for flexible event handling, formalizes them, and provides implementation to show the feasibility of the concepts. Considering the shortcomings of current notation and process engines make this contribution *current*. While the overall contribution provides flexible subscription configurations for process execution, this can be further divided into the following sub-contributions:
 - The *event handling notions* for subscription, occurrence, consumption, and unsubscription of an event are discussed separately from the business process view as well as the event processing view. The temporal constraints between the notions are also explored.
 - Considering the usecase needs, the constraints between the event handling notions, and the process execution timeline; four *points of subscription* are specified. This addresses the research question “*RQ1: When to subscribe to an event source?*”. The points of subscription cover the BPMN semantics of listening to an event when the control-flow activates the event construct, and extends the possibility for an *early subscription* at process instantiation, at process deployment, and at engine initiation. This enables the process to receive an event even when the execution flow is not ready to consume it, to increase the chance of including an early occurrence of an event while it is still relevant for the process.
 - The concept of early subscription raises the need for storing the events temporarily till it is consumed by the process. This brings us to the next contribution, namely, an *event buffer* with buffer policies to control the lifespan for keeping an event in the buffer, to determine the most suitable event for a process instance when more than one occurrences have happened in between the subscription and consumption, and to facilitate reuse of an event information across instances as well as across processes. These buffer policies answer the questions discussed in the context of “*RQ3: How to store and retrieve the relevant events for a specific process execution?*”.
 - Similar to the points of subscription, the *points of unsubscription* are also specified to address “*RQ2: For how long to keep the subscription?*”. Though unsubscription is not mandatory, it is recommended to avoid engine overhead with events that are not relevant any more. The interdependencies among the

chosen event handling configurations are discussed in this context to guide a correct and efficient process execution.

- Finally, all the above concepts for flexible event handling are given formal semantics by *mapping the concepts to Petri net modules*, classified by the types of *event constructs*. While standard Petri nets are used for a clear semantics for the event handling notions and points of (un)-subscription, Coloured Petri Nets (CPN) are used to formalize the buffer policies. Thus, this thesis provides a formally grounded flexible event handling model for business processes and fulfills the research objectives. Based on the formal semantics, two applications demonstrating the impact of selecting event handling configurations are included as well, adding to the *applicability* of the contribution.

Figure 3 visualizes the summary of the contributions. The parts highlighted in green are the concepts built up during the course of the thesis, while the highlighted part in orange signifies the extension of existing mapping technique.

1.4 STRUCTURE OF THESIS

After the introduction to the motivation, problem statement, research objectives, and contributions; this section outlines the structure of the complete thesis. This thesis consists of three main parts, as described below.

PART I: INTRODUCTION & FOUNDATIONS. This is the first part of the thesis and we are already in this part. After the introduction chapter, the preliminaries are given based on which the research has been conducted. Since the thesis is vastly emerged into the concepts from both business process management and complex event processing fields, each of them are discussed as foundations. Chapter 2 starts with the BPM lifecycle. Our work is based on the specifications and shortcomings of Business Process Model and Notation. Therefore, introduction to the relevant core elements of BPMN 2.0 are given in this chapter. The role of events in BPMN is discussed in more detail. As a more formal process execution notation, Petri nets are introduced. The existing mapping from BPMN to Petri net by Dijkman et al. [37] is considered as a foundation as well, since this is later extended for mapping the event handling configurations. In Chapter 3, the basic concepts of event processing such as event abstraction, event queries, and event pattern matching are introduced. Chapter 4 concludes this part of the thesis with extensive discussion about the related work that focus on integrated applications of CEP and BPM as well as event subscription and buffering concepts.

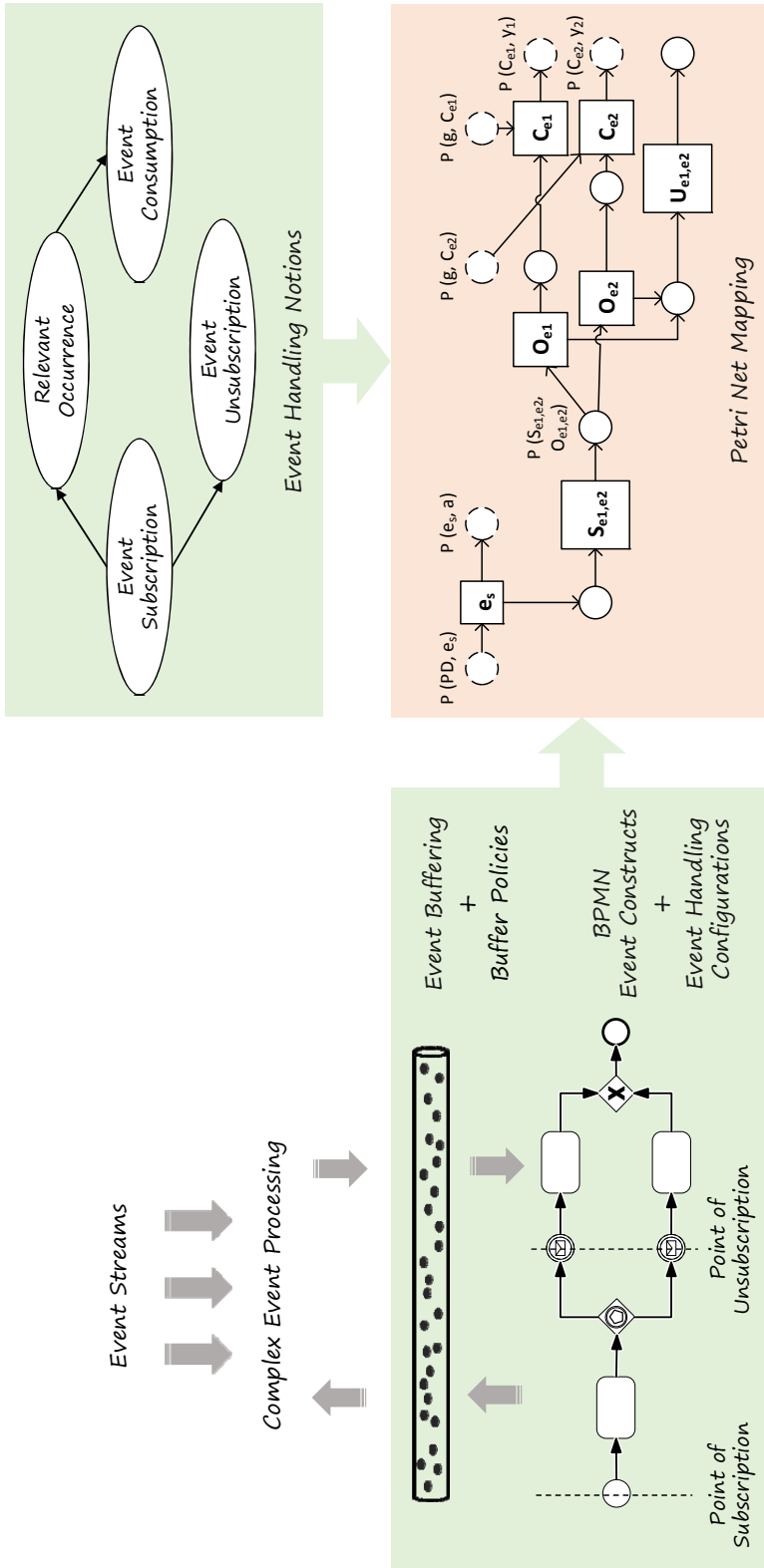


Figure 3: Summary of research contributions.

PART II: CONCEPTUAL FRAMEWORK. This part of the thesis introduces the core contributions. We start with the basic integrated architecture to use real-world events into business processes in [Chapter 5](#). The requirements elicitation for integration challenges are described first. Next, the solution architecture fulfilling the requirements is illustrated. [Chapter 6](#) goes deeper into event subscription mechanism and defines the advanced event handling concepts. The flexible subscription management with different subscription and unsubscription configurations are explored in this chapter. The event buffering concept with the buffer policies are also discussed in this context. Next, in [Chapter 7](#), formal semantics are assigned to the event handling concepts. This includes Petri net mapping for the event handling notions, the points of subscription and unsubscription for each event construct, and the event buffer policies.

PART III: EVALUATION & CONCLUSIONS. The last part of the thesis turns to evaluation of the concepts introduced so far. First, [Chapter 8](#) demonstrates the applicability and significance of event handling concepts based on the formal semantics. Trace analysis is used as a verification method for correct process execution. Based on the Petri net mapping, temporal constraints for each pair of event construct and point of subscription are defined for this application. Further, reachability analysis of a process path is investigated in presence of early subscription and reuse of events. As proof-of-concept, both basic and advanced event handling implementation are discussed in [Chapter 9](#). Using a process modeler, a process engine, and an event processing platform; the solution architecture is implemented to enable basic event-process interactions. To realize flexible event handling, BPMN notation is extended to specify event handling configurations. Next, the event processing platform is extended to incorporate buffer functionalities. As a complement to the event processing platform, an open-source process engine is adapted to control the time of issuing a subscription. Finally, this thesis is summarized in [Chapter 10](#) along with discussions about the limitations and future research directions.

Processes are everywhere, in our everyday lives. Whenever we aim to do something repetitively, we tend to follow certain steps. As soon as these steps and the end result bear business value, it becomes even more important to perform the tasks in a structured way and to consider the alternative approaches to execute the tasks to get a better result. Business processes capture these steps, the order between them, the decisions needed to be taken and the information influencing the decisions. The work presented in this thesis is grounded in the vast concepts of business process management (BPM). This chapter introduces the relevant ideas and techniques from the field of BPM which are necessary to follow the advanced work. Namely, the core concepts of modeling and enacting business processes are discussed with the help of standard notations such as Business Process Model and Notation and Petri Net (PN), along with the mapping from one to another. Furthermore, the use of external events in business processes are explored in detail to understand the aspects and challenges of the integration.

2.1 BPM LIFE CYCLE

Several definitions of a business process can be found in [22, 42, 64, 69, 116, 134]. We abide by the definition given by Weske in [134], that says a *business process* is a set of activities performed in a coordinated manner in a business or technical context to achieve pre-defined business goal(s). The complete process of supporting the design, administration, configuration, enactment and analysis of business processes with various concepts, methods, and techniques is known as *Business Process Management* (BPM). This section explains the phases of business process management to understand the underlying concepts and technologies in more detail. Figure 4 represents the business process lifecycle, as discussed in [134].

The entry point to the cycle is the *Design and Analysis* phase. BPM is a model-driven approach [42]. In this phase, the business processes are extracted and represented visually in a model. The processes can be elicited using different techniques such as interviewing the knowledge workers, observing the activities they perform, or conducting a workshop to model the process together with the domain experts [77]. Instead of extracting the model from the knowledge workers, they can also be discovered from the execution logs documented by the Enterprise Resource Planning (ERP) system or relational databases, using process mining techniques [124]. Typically, a process consists of the key activities to achieve certain goal(s), the decision points along the

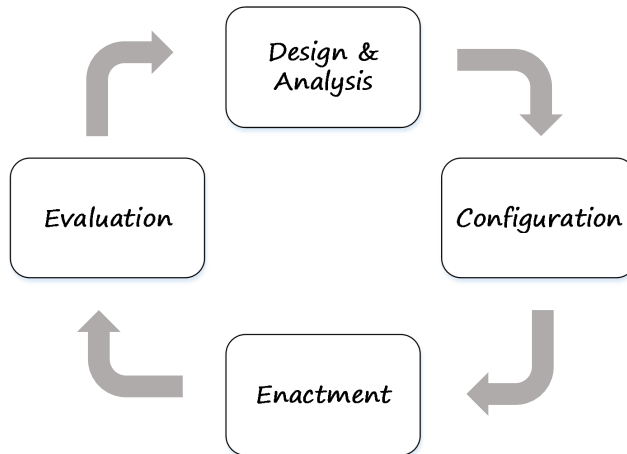


Figure 4: Business process management (BPM) lifecycle.

way to reach the goal(s), and the interactions with other processes or the environment that influence the tasks performed and the decisions taken. After extracting the process, it is modeled formally using a specific business process modeling notation. Once modeled, the process is simulated and verified to make sure it covers the complete set of execution possibilities and serves the modeling goal(s).

The next phase is *Configuration* of the process model to implement it. The implementation of the process can be done with or without a dedicated IT system [41]. If there is a dedicated IT system, such as a process engine, then the modeled process is enhanced with further technical details which are parsed and followed by the process engine to execute the process. If there is no IT support for process execution, then certain rules and procedures are distributed among the human resources responsible for conforming to the process while executing it manually.

In the *Enactment* phase, the configured process is actually executed following the process specification. There can be several executions of a single process. Each execution is one process instance. The process engine logs these execution data at runtime. This information can be used to monitor the status of the process execution at anytime during the execution [51, 120].

Next, *Evaluation* is done to check the conformance of the process execution with the process specification. The execution traces logged by the engine is analyzed in this phase to find the bottlenecks or deviations from actual model [25, 87, 132]. Evaluation can also detect performance metrics of a process. The process is further improved based on that, e.g., by increasing the flexibility, efficiency, effectiveness, and user satisfaction. Again, process mining techniques are often applied to the execution logs for conformance checking or performance evaluation [1, 66, 90]. Finding the patterns of state transition events leading to good or bad states by analyzing the execution log data is another

evaluation aspect. This is then used for predictive monitoring during future process executions [10, 43, 51, 52, 86, 107].

2.2 BUSINESS PROCESS MODEL AND NOTATION

As discussed above, the first step for process implementation is to model the process visually following the behavioral specifications. Later, these specifications are executed for each process instance. In addition, process models are discovered from the execution log for checking the conformance between an intended behavior and corresponding actual behavior [124]. Thus, process models lie at the heart of business process management.

Several languages and notations are available for process description. Depending on the purpose of process modeling, the most suitable language can be selected. For describing structured processes, procedural languages such as Business Process model and Notation (BPMN)¹, Event-driven Process Chain (EPC)², Web Service Business Process Execution Language (BPEL)³ might be used. Whereas for describing more flexible processes, there are approaches such as declarative [97], artifact-centric [59] or hybrid. The declarative modeling languages do not bind the activities in a sequence, rather specify some constraints that should be followed while executing them. However, the non-procedural approaches are still evolving, whereas procedural languages have already been in use in business context for years [98]. We followed the industry standard *BPMN 2.0* for our work. The basic concepts of BPMN relevant to our work are described next.

2.2.1 Core Elements

In essence, a process model is a graph with nodes and edges [134] where nodes can be activities — representing a task, gateways — representing branching in process execution, and events — representing relevant occurrences influencing the process. A process model is used as a blue print for a set of process instances which are the individual executions of this process.

Each process instance consists of several activity instances. These activity instances traverse through different life cycle states, as shown in Figure 5. Once the process is instantiated, each activity is initialized. This puts the activity instances in state *init*. As soon as the incoming flow of an activity is triggered, it enables the instance and changes the state to *ready*. With the start of activity execution, the state changes to *running*. Finally, the activity instance is *terminated* once the execution is completed. In some cases, while the activity instance is yet *not started*, the process execution chooses a different path. This puts the activity

Activity life cycle

¹ <https://www.omg.org/spec/BPMN/2.0/>

² <https://www.ariscommunity.com/event-driven-process-chain>

³ <https://www.oasis-open.org/committees/wsbpel/>

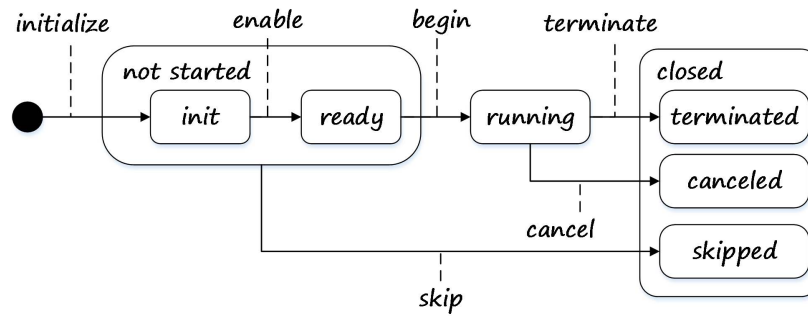


Figure 5: State transition diagram for activity lifecycle.

into the *skipped* state. Also, due to the occurrence of an exceptional situation, a running activity instance can switch to *canceled* state. During one process instance execution, an activity can be re-instantiated after it is terminated or canceled.

An activity can be a *task* that represents the atomic step in a process, a *sub-process* that abstracts the lower-level process within an activity, or a *call activity* that calls and gives the control to a global task. Depending on the nature of the task, it can be classified in following categories: *Service task* — performed by calling a web service or any other automated service, *Send task* — responsible for sending a message to an external participant, *Receive task* — responsible for receiving message from an external participant, *User task* — performed by a human user, *Manual task* — performed without any support of execution engine, *Business rule task* — abstracted decision logic involved in the activity [96], and *Script task* — executed by process engine. Different types of BPMN tasks are visualized in Figure 6 with their standard icons.

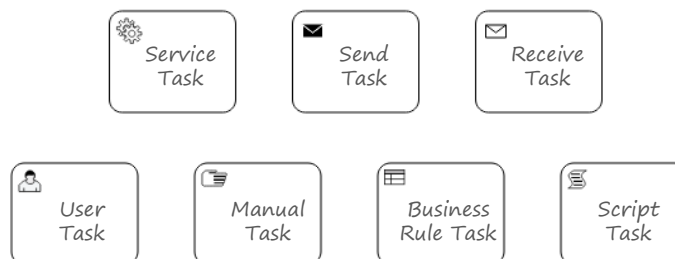


Figure 6: BPMN tasks.

Gateways

The control flow maintaining the sequence among these activities can be optimized using gateways. There are six types of gateways to specify the branching behaviors, as described below. *Exclusive gateways* are used for alternative paths based on data-based decisions. Only one of the outgoing branches following the exclusive gateway can be chosen for a process instance execution. The converging exclusive gateway merges the alternative branches. On the contrary, *inclusive gateways* can

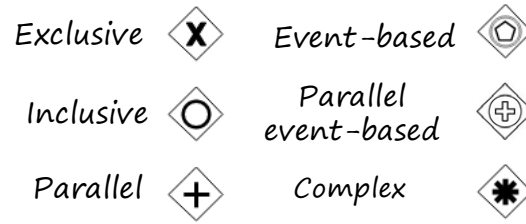


Figure 7: BPMN gateways.

be used to choose one or multiple paths among the alternatives. As the name suggests, the *parallel gateway* leads the process execution to branches that can be executed in parallel, independent of each other. Here, all the branches following the gateway have to be executed in order to move further. While the exclusive or inclusive gateways are based on data-based decisions, the *event-based gateway* directs the path to execute depending on the event occurrence order. The first event to occur among the events after the gateway leads the further execution in this case. *Parallel event-based gateway* needs all the associated events to be triggered for process completion. For more complex and specific branching conditions (such as when three out of five branches are to be executed), *complex gateways* might be used. [Figure 7](#) summarizes the different types of BPMN gateways along with their notations.

BPMN offers several event constructs to represent the interaction between two processes as well as a process and its environment. Since this thesis intensely deals with events in business processes, an elaborate description of BPMN events is given later in [Section 2.2.2](#). Based on the above discussion, we now formally define the relevant concepts of a process model for our work in the following.

Definition 2.1 (Process Model).

A *Process Model* is a tuple $\mathfrak{M} = (\mathbb{N}, cf, \mathcal{B})$ with

- a finite non-empty set of nodes $\mathbb{N} = \mathbb{N}_A \cup \mathbb{N}_E \cup \mathbb{N}_G$ where \mathbb{N}_A , \mathbb{N}_E , and \mathbb{N}_G are pairwise disjoint sets of the activities, the events, and the gateways, respectively,
- a control flow relation $cf \subseteq \mathbb{N} \times \mathbb{N}$, and
- a function \mathcal{B} that maps the activities to the associated boundary event(s).
- $\mathbb{N}_E = \mathbb{E}_S \cup \mathbb{E}_I \cup \mathbb{E}_E$, where \mathbb{E}_S , \mathbb{E}_I , and \mathbb{E}_E are pairwise disjoint sets of start events, intermediate events, and end events, resp.
- $\mathbb{E}_I = \mathbb{E}_{IC} \cup \mathbb{E}_{IT}$ where \mathbb{E}_{IC} and \mathbb{E}_{IT} are pairwise disjoint sets of intermediate catching events and intermediate throwing events, resp.
- $\mathbb{N}_G = \mathbb{G}_A \cup \mathbb{G}_X \cup \mathbb{G}_E$, where \mathbb{G}_A , \mathbb{G}_X , and \mathbb{G}_E are pairwise disjoint sets of AND gateways, XOR gateways, and event-based gateways.

- $\mathcal{B} : N_A \rightarrow \mathcal{P}(E_{IC})$ where $\mathcal{P}(E_{IC})$ is the power set of the intermediate catching events.

◆

For an activity $A \in N_A$, let A_b, A_t, A_c be the beginning, termination and cancellation of A , respectively. A start event is always a catching event, i.e., the process receives the event whereas an end event is always a throwing event, i.e., the process produces the event. The preset of a node $n \in N$ is defined as $\bullet n = \{x \in N \mid (x, n) \in cf\}$. The postset of a node $n \in N$ is defined as $n\bullet = \{x \in N \mid (n, x) \in cf\}$.

2.2.2 Events in Business Processes

Events are the happenings that are relevant to process execution. It can signify the state changes for an activity, the state changes of a data object, and the communication with the environment [75]. The processes can consume events using the *catching events*, and can also generate events, modeled as *throwing events*. A process always gets instantiated by a *start event* which can be a catching event received from the environment or an engine generated event. Each process path should terminate with an *end event*. Start events are always catching events and end events are always throwing events. The *intermediate events* happen in between start and end of a process and they can either be throwing or catching by behavior.

BPMN specifies a range of event constructs to capture different semantics of happenings that influence a process execution. The complete description of BPMN events are found in the standard [94]. Figure 8 gives brief introduction to the events and classifies them according to start, intermediate, and end events as well as their catching and throwing nature. The event types used in this thesis are described in detail below.

BLANK EVENT. This do not add any special semantics to the event. Rather, it just signifies the state depending on the position. Blank events can be used as a start event, throwing intermediate event, and end event.

MESSAGE EVENT. This is used for message flow between process participants. It can be both throwing (sending message) and catching (receiving message). Message events are also used for receiving external information into processes. Message events might have *data association* to implement the input of information to the event payload by the sending process and to write the information carried by the event to a data object to use it further in the receiving process. If the message event is attached to an activity (boundary event), then it works as an exception trigger and initiates an exception handling path in the process. If the boundary event is interrupting, then the associated activity is canceled upon firing of the event.

| Events | Start | | | Intermediate | | | End |
|---|----------|--------------------------------|------------------------------------|--------------|-----------------------|---------------------------|----------|
| | Standard | Event Sub-Process Interrupting | Event Sub-Process Non-Interrupting | Catching | Boundary Interrupting | Boundary Non-Interrupting | Throwing |
| None: Untyped events, indicate start point, state changes or final states. | | | | | | | |
| Message: Receiving and sending messages. | | | | | | | |
| Timer: Cyclic timer events, points in time, time spans or timeouts. | | | | | | | |
| Escalation: Escalating to an higher level of responsibility. | | | | | | | |
| Conditional: Reacting to changed business conditions or integrating business rules. | | | | | | | |
| Link: Off-page connectors. Two corresponding link events equal a sequence flow. | | | | | | | |
| Error: Catching or throwing named errors. | | | | | | | |
| Cancel: Reacting to cancelled transactions or triggering cancellation. | | | | | | | |
| Compensation: Handling or triggering compensation. | | | | | | | |
| Signal: Signalling across different processes. A signal thrown can be caught multiple times. | | | | | | | |
| Multiple: Catching one out of a set of events. Throwing all events defined | | | | | | | |
| Parallel Multiple: Catching all out of a set of parallel events. | | | | | | | |
| Terminate: Triggering the immediate termination of a process. | | | | | | | |

Figure 8: BPMN events, as presented in BPMN 2.0 Poster by BPM Offensive Berlin (cf. <http://www.bpmb.de/index.php/BPMNPoster>).

TIMER EVENT. This is a process (engine) generated event that gets enabled with the control flow reaching the event construct and fires once the specified time has passed. Since the firing is received as a trigger to the process, it is always a catching event for the process. Timer event can be conditioned as a specific fixed date and time (2019-02-25 16:25:14), as a time duration (50 min), and as a time cycle (every Monday at 9:00 am). The timer event can be set as a start event, except from the time duration condition. It can also be used as a catching intermediate event, including usage as a boundary event.

2.3 PETRI NETS

Petri nets are one of the most popular and standard techniques to represent the behavior of concurrent systems [69]. It is named after *Carl Adam Petri*, who visioned the initial foundation for Petri nets in 1962. The net is composed of places and transitions, connected with directed arcs between them in a bipartite manner. The transitions represent active components of a system, such as activities, events or gateways in a process. On the other hand, the places are used to model the passive components, e.g., the input place models the precondition and the output place models the postcondition of a transition. We chose Petri nets for our mapping since it gives clearer implementation semantics than BPMN. The Petri net semantics used here follow the definitions proposed in [69] and [131].

A marking of a Petri net signifies the system state. A marking is a snapshot of the distribution of tokens over the places of the net. The firing of a transition can change the marking, i.e., the state of the system. Firing of transitions are considered as atomic step. The behavior of the system is described by all firing sequences of a net that start with an initial marking. A single firing sequence is named as a *trace*. The relevant definitions are quoted in the following:

Definition 2.2 (Petri Net).

A *Petri net* is a tuple $\mathfrak{N} = (P, T, F)$ with

- a finite set P of places,
- a non-empty, finite set T of transitions, such that $T \cap P = \emptyset$, and
- a flow relation $F \subseteq (P \times T) \cup (T \times P)$.

◆

A marking of \mathfrak{N} is a function $M : P \rightarrow \mathbb{N}_0$, that maps the set of places to the natural numbers including 0. $M(p)$ returns the number of tokens on the place $p \in P$. Let \mathbb{M} be the set of all markings of \mathfrak{N} . A Petri net system is a pair $S = (\mathfrak{N}, M_0)$, where \mathfrak{N} is a Petri net and $M_0 \in \mathbb{M}$ is the initial marking. A sequence of transitions $\sigma = t_1, t_2, \dots, t_n, n \in \mathbb{N}$, is a firing sequence, iff there exist markings $M_0, \dots, M_n \in \mathbb{M}$, such that for $1 \leq i \leq n$, transition t_i changes the system from (\mathfrak{N}, M_{i-1}) to (\mathfrak{N}, M_i) .

The set of *traces* \mathcal{T} contains all firing sequences σ , such that σ is enabled in M_0 .

The simplest form of Petri net is a *Condition Event Net* where the places represent conditions and the transitions represent events. If a condition is met then there is a single token on the place representing that condition and it enables the firing of the succeeding event. Note that a transition is enabled only if there is no token on any output place of the event which is not an input place as well. A *Place Transition Net*, on the other hand, allows multiple tokens in a place. The number of tokens in a place is generally unbounded. Depending on the need of workflow, the number of tokens in an input place (to be consumed by events) and number of tokens in an output place (to be produced by events) can be defined using a weighing function. The classic Petri nets are representations of discrete states, as defined by their markings. In real-life workflows, however, continuous parameters such as time is often an important concern. There exist hybrid Petri nets where time-dependency of a place, transition, and arc can be described. *Time Petri Net*, *Timed Petri Net*, and *Petri Net with Time Windows* are such extensions [100].

*Extensions of
Petri nets*

Another aspect that is ignored in the traditional Petri nets is the differentiation of the tokens based on the data they carry. *Coloured Petri Nets* (CPN) address this shortcoming by extending Petri nets with data handling concepts. The places are typed with a colour set that determine the data type of the tokens that can be kept in that place. In complement, the tokens are assigned definite colours, i.e., values of the specified data type. The flow arcs are annotated with arc expressions that contain functions and variables. The arc expressions specify how tokens might be consumed and produced upon firing of a transition. Additionally, the transitions are restricted with guard conditions which are Boolean predicates that need to be true to fire the transition. The definition of Coloured Petri Net is given in the following, based on the definitions provided in [63, 135]. An exhaustive discussion on different kinds of Petri nets is found in [104].

Definition 2.3 (Coloured Petri Net).

A *Coloured Petri net* is a tuple $\mathcal{CPN} = (\Sigma, P, T, A, N, C, E, G)$ with

- a non-empty, finite set Σ of typed colour sets,
- a finite set P of places,
- a non-empty, finite set T of transitions, such that $T \cap P = \emptyset$,
- a finite set A of arc identifiers, such that $P \cap T = P \cap A = T \cap A = \emptyset$,
- a node function $N : A \rightarrow (P \times T) \cup (T \times P)$,
- a colour function C that associates the places with a typed colour set, such that $C : P \rightarrow \Sigma$,
- an arc expression function $E : A \rightarrow \text{Expr}$, and

- a guard function $G : T \rightarrow \text{BooleanExpr}$.

◆

The initial marking of \mathcal{EN} follows the same semantics as that of \mathcal{N} , as presented in [Definition 2.2](#). In addition, we define a list of m elements as $l = \langle x_1 \dots x_m \rangle$ to specify arc expressions and guard conditions. $|l| = m$ denotes the length of the list. The i -th element of the list is referred as $l(i) = x_i$.

2.4 BPMN TO PETRI NET MAPPING

Motivation

In the *design and analysis* phase of business process management lifecycle ([Section 2.1](#)), the processes are verified and simulated after modeling to make sure that it is sound and free from any deadlock or livelock. BPMN satisfies the requirements to be an expressive and user-friendly graphical notation to design processes efficiently. However, when it comes to static analysis, BPMN lacks the needed unambiguity and clear formal semantics for model checking [63]. On the other hand, there are popular tools for semantic analysis that uses Petri nets as input. Mapping BPMN process structures to Petri nets thus assigns clear semantics to the process behavior and enables the model to have a proper analysis. This section describes the mapping of core BPMN elements to Petri nets as formulated by Dijkman et al. [37]. The mapping focuses on the control-flow aspects of BPMN and abstracts from the organizational aspects such as pools and lanes. Only the relevant concepts are discussed here, the readers are referred to the main paper for more details. The work in this thesis further extends this formalism with the concepts for event handling (cf. [Chapter 7](#)).

TASK. A task in a BPMN process is mapped to a transition with the same label in Petri net. The transition has one input and one output place that connects it with the predecessor and successor node, respectively. For a task T , x denotes the predecessor node of T , y denotes the successor node of T . Places with dashed borders mean they are not unique to one module, i.e., they can be used to connect with other modules that map other BPMN artifacts. [Figure 9](#) visualizes the mapping.

GATEWAYS. The fork and join gateways for both XOR and AND are mapped to silent transitions that represent their routing behavior. The XOR split has one common place for all outgoing branches that is followed by separate silent transition for each branch. This captures the data-driven decision alternatives and the exclusivity of the branches. The AND split shares the token from one place to the parallel branches through a single transition. The XOR join merges the transitions for each branch to a common output place for passing the token to next node, whereas in case of an AND join, the corresponding silent transition needs a token in each branch for firing. For event-based gateways

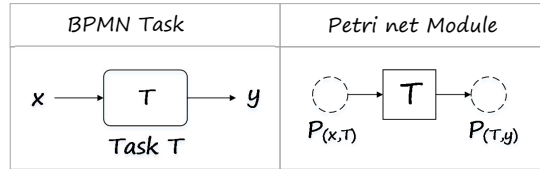


Figure 9: Mapping of BPMN task to Petri net modules.

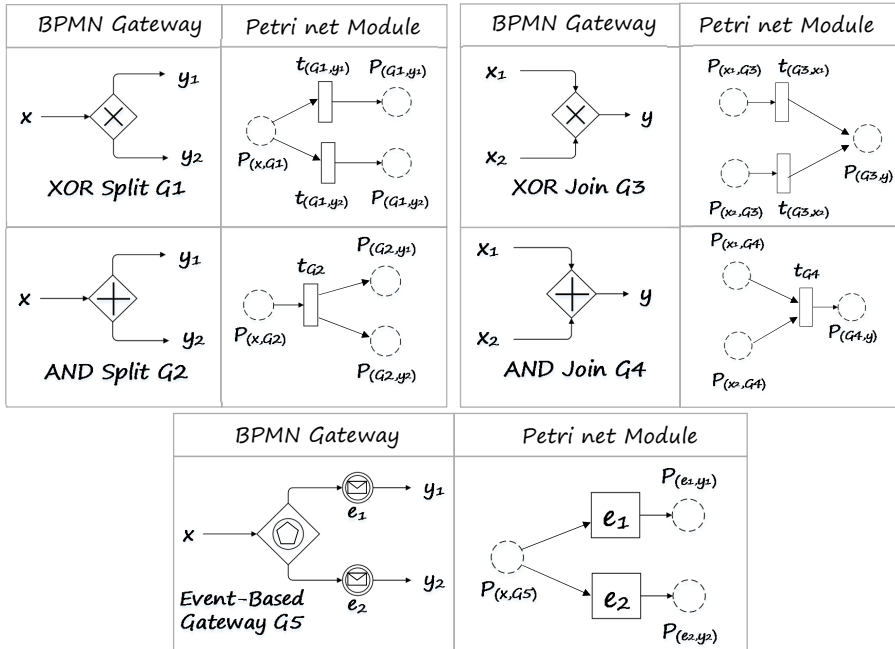


Figure 10: Mapping of BPMN gateways to Petri net modules.

the silent transitions are replaced by transitions reflecting each event after the gateway. These transitions share a common place for having the input token to capture the racing situation. Figure 10 shows BPMN gateways and corresponding Petri net modules. x, x_1, x_2 denote the predecessor nodes and y, y_1, y_2 denote the successor nodes of gateway.

EVENTS. According to the mapping techniques by Dijkman et al., the intermediate events are mapped exactly in the same way as the tasks — a transition with the same label as the event, with one input and one output place to connect to the previous and next node, as shown in Figure 11. However, a start event does not have any incoming edge and is mapped to a place followed by a silent transition that signals the instantiation of the process. Similarly, an end event does not have an outgoing edge which is reflected by the silent transition in the that leads to the end place.

The boundary events are mapped to transitions with same label that share a common input place with the transition mapping the associated task, as presented in the Petri Net Module in Figure 12. As long as the task is not executed, the event can steal the token from the common

Boundary event


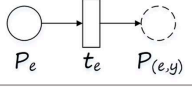
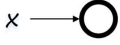
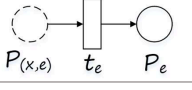

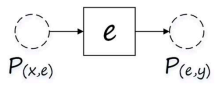
| BPMN Event | Petri net Module |
|---|---|
|  Start e |  P_e t_e $P_{(e,y)}$ |
|  End e |  $P_{(x,e)}$ t_e P_e |
|  Intermediate e |  $P_{(x,e)}$ e $P_{(e,y)}$ |

Figure 11: Mapping of BPMN event constructs to Petri net modules.

place and fire to initiate the exception handling path. Note that the mapping of boundary event according to Dijkman et al. [37] assumes atomic execution of the associated task. Precisely, the firing of the transition mapped to the associated task means the task has been executed, therefore, “terminated”. Once the task is terminated, the boundary event can not occur any more, since the token from the shared input place is consumed by the task. In case the boundary event occurs before the task is terminated, the event consumes the token instead. As a result, the task can not terminate any more.

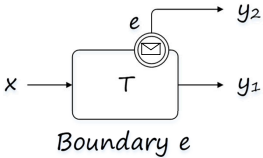
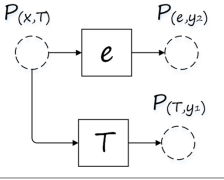
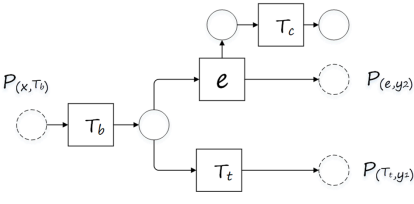
| BPMN Event | Petri net Module |
|--|--|
|  Boundary e |  $P_{(x,T)}$ T $P_{(T,y_1)}$ $P_{(x,e)}$ e $P_{(e,y_2)}$ |
| <i>Revised Petri net Module</i> | |
|  $P_{(x,T_b)}$ T_b T_c T_e e $P_{(e,y_2)}$ $P_{(T_e,y_1)}$ | |

Figure 12: Mapping of BPMN boundary event construct to Petri net module.

However, the semantics of boundary event demands to consider the duration of the associated task explicitly, since the token from the common place is allowed to be stolen by the event only when the task is in the “running” state. Moreover, the associated task has to be cancelled if the boundary event occurs. The *Revised Petri Net Module* with separate transitions for the beginning of task T (T_b), termination of task T (T_t), and cancellation of task T (T_c) is shown in Figure 12. Here, the shared place for the boundary event and the termination of the task receives the token only after the activity has begun. Now the previous semantic

of a racing condition between the transitions depicting the boundary event e and the termination of the task T_t apply. If the boundary event does not occur when the activity is running, the task terminates at a due time and the next node in the normal branch is activated, shown by the place $P(T_t, y_1)$. On the contrary, once the boundary event occurs before the activity terminates, a token is placed at the input place of the transition T_c as well as at the input place of the next node in the exceptional branch, shown by the place $P(e, y_2)$.

COMPLEX EVENT PROCESSING

An event is a specific occurrence at a specific point in time. These events form streams of information that can be analyzed to gain insight about the series of happenings. Complex event processing (CEP) is the field of research to investigate the concepts and techniques for efficient processing of the large number of events which could be related to each other by causal, temporal, or structural means. Event processing enables the analysis and automation of information exchange between distributed IT systems that can be beneficial to several application domains such as health care, logistics, surveillance, agriculture, production line, and the Internet of Things (IoT). The current thesis exploits complex event processing techniques to make business processes aware of and reactive to a relevant contextual occurrence, leading to increased efficiency and flexibility. This chapter introduces the basic concepts from the CEP area, followed by detailed event processing techniques used in the course of this work.

3.1 BASIC CONCEPTS

In the business process management world, the term *event* can refer to several overlapping yet distinguishable concepts. The state changes during process execution are recorded by the process engine in an event log. These are generally known as *transitional events*. The state changes in activity lifecycle discussed in Section 2.2.1 will generate such events, e.g., beginning and termination of a certain activity. Process mining applications rely on these event logs for process discovery [1]. The *BPMN events* described in Section 2.2.2 refer to the event nodes defined in BPMN standard to model the communication of a process with its environment. These nodes are mapped to the actual outer world occurrences, known as *external events*.

In essence, events are anything that has happened (e.g., an accident) or is supposed to have happened (e.g., a fraud login attempt) at a specific point in time in a specific context [46]. In an everyday life, we are surrounded by real events such as waking up with an alarm, getting a coffee, a train getting canceled on our way to work, an accident on the highway that makes us take a detour, and finally reaching our destination. The real events are digitized to create an event object and use the information further. Event objects or events can be generated by sensors, mobile devices, business processes, or even human beings. These event sources or *event producers* can produce events in different formats such as XML, CSV, RSS feed, plain text, email and so on. The entities listening to such events and using the information carried by

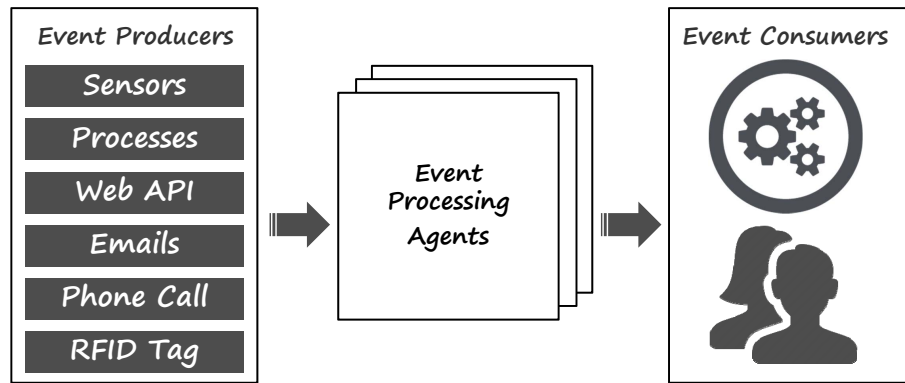


Figure 13: Event processing network.

the events are called *event consumers*. Again, event consumers can be a person as well as a system, e.g., a BPMS.

However, often the raw events produced by the event producers are too low-level for the event consumers. For example, a single login attempt might not be interesting, but five consequent wrong login attempts might raise a fraud alert. Hence, *Event Processing Agents (EPA)* are employed to perform a large range of operations on events from single or multiple event sources. Event producers, event consumers, and EPAs together form an *Event Processing Network (EPN)*. Figure 13 shows the entities and their connections in an event processing network.

Events that occur in a stateless way are called *atomic events*. They do not take any time to occur, e.g., the single login attempt in previous example. But to come up with the fraud alert, it is necessary to observe five consequent login events for the same account. Here, fraud alert can be an example of a *complex event*. Each atomic and complex event has a timestamp and an unique identification. Additionally, the events might carry data, often called *payload*, which can be used in further applications.

An event producer produces events of specific types. The events of the same *event type* are the instances of that type. Each event of the same type has the same structure. The attribute values for events are specific to that instance. Based on the description above, an event can be defined as follows:

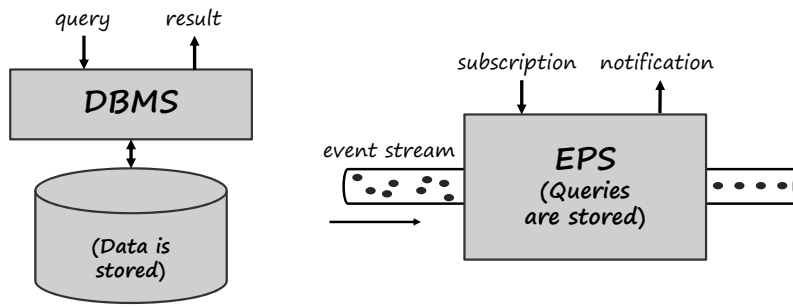


Figure 14: Data in rest vs. data in motion.

Definition 3.1 (Event).

An *Event* is a tuple $E = (et, id, ts, pl)$ where

- *et* is the event type, i.e., the list of attributes with assigned datatype,
- *id* is the identification, i.e., the unique identifier assigned to each event,
- *ts* is the timestamp, i.e., the point in time when the event occurs, and
- *pl* is the payload, i.e., the key-value pairs of the attributes of the corresponding event type *et*.

◆

Both Event Processing Systems (EPS) and Database Management Systems (DBMS) use query languages to calculate insightful results from the data. However, event processing in its essence, has a complementary orientation compared to DBMS. In databases, the data is stored and updated periodically. The queries are fired on a set of static data to return results matching the queries. For EPS, the queries are registered instead. The queries are checked with each new event in the event stream(s). Whenever a matching event occurs, the corresponding query is satisfied. [Figure 14](#) visualizes the different approaches towards static data and dynamic data.

Difference in approach between EPS & DBMS

3.2 EVENT DISTRIBUTION

In an event processing network, the event producers, event processing agents, and event consumers communicate through their interfaces and may follow different communication patterns. The event producers and event consumers can talk directly to each other. They might also communicate indirectly through a channel – such as a network of EPA(s), an event bus, or an event processing platform. The communication can be either pull-style or push-style in nature. In a pull-style communication the initiators of the conversations send requests to fetch the information from the producers and waits till the reply is received before they can start working on the next task [46]. Thus, pull-style communication

| | <i>Pull</i> | <i>Push</i> |
|-----------------|--|-------------------------------|
| <i>Direct</i> | <i>Request/ Response</i> | <i>Callback</i> |
| <i>Indirect</i> | <i>Anonymous Request/ Response</i> | <i>Publish/ Subscribe</i> |

Figure 15: Communication models for event distribution.

is synchronous in nature. On the contrary, in push-style communication the initiator invokes the communication, keeps working on other task(s), and receives the response at a point of time in future without waiting idly for it, making the communication asynchronous in nature. Based on the above two aspects the communication models for event distribution can be classified as shown in [Figure 15](#).

REQUEST/RESPONSE MODEL. Request/response model is followed extensively in service-oriented architectures and distributed computing. Here, the initiator directly sends the request for information or update, such as a client-server framework or a remote procedure call. The server then sends the information or a confirmation that the update request is received, respectively. The nature of request/response communication is usually synchronous.

ANONYMOUS REQUEST/RESPONSE MODEL. This is similar to request/response, but the provider is not specified directly [89]. Instead, a random provider or a set of providers receive the request. For example, in the shared car system Uber¹ a set of nearby drivers are informed when the passenger sends a request.

CALLBACK MODEL. Following this communication model the consumers subscribe for specific events directly to the producers along with a callback address. After the subscription is done the consumers can focus on other tasks, since callback is asynchronous in nature. Once the subscribed event occurs, the event producer pushes the notification to the consumers who have already subscribed for the specific event, since the producer has the callback addresses of the consumers.

While callback method is popular as part of the programming language Java, a non-technical example of callback model can be ordering a meal in a restaurant – the consumer orders a specific meal and talks to other people or works on her laptop. When the meal is ready, she is served the meal at her table. Thus, callback is a direct but asynchronous communication between the producer and the consumer.

¹ <https://medium.com/@narengowda/uber-system-design-8b2bc95e2cfe>

PUBLISH/SUBSCRIBE MODEL. Event-driven systems use publish/-subscribe principle [58] intensively to access event information. This is similar to callback, but the producers and consumers do not communicate directly, rather they are connected through a channel such as a message queue or an event service. In a subscription-based publish/-subscribe model consumers register their interest in specific events by issuing a subscription. Whereas, in an advertisement-based publish/-subscribe model, the producers issue advertisements to show their intend to publish certain events in future. Following the advertisement the consumers subscribe for the events they want.

Once a consumer subscribes to an event, it can consume the event whenever it occurs and as many times as it occurs. If the consumer issues an unsubscription for the event at a certain point, no further consumption of that event is possible by that specific consumer. Also, the producer might choose to unadvertise a specific event at some point.

Publish/subscribe communication model has the following advantages over the other communication models, making it the popular choice for event-driven architectures for the last two decades [46, 58, 89].

- This is a push-style distribution, i.e., the consumer subscribes once and until unsubscription the event notifications are pushed by the channel as soon as and as many times as it has a matching event to distribute. The other communication model such as request/response on the other hand is pull-style, i.e., the consumer needs to fetch the event in a regular basis to have updated notification about the event occurrence. The push-style makes publish/subscribe more efficient in terms of processing latency and overhead for the consumers [89].
- Being an indirect interaction pattern, the producers and consumers are connected through an intermediate channel, not being aware of each other. This supports loose coupling such as easy addition and deletion of the consumers and reuse of events. The event producer do not need to keep track of the consumers and the event consumers can have multiple subscriptions to multiple event sources, since the task of event distribution is delegated to the dedicated event service.
- Finally, the asynchronous communication allows the consumers to work on other tasks rather than waiting idly for the event occurrence.

3.3 EVENT ABSTRACTION

The main goal of event processing is to interpret the data representing environmental occurrences in a meaningful way. Thereby, it is important to find the connections between the events. An event hierarchy represents the intra- and inter-relations among events from separate event

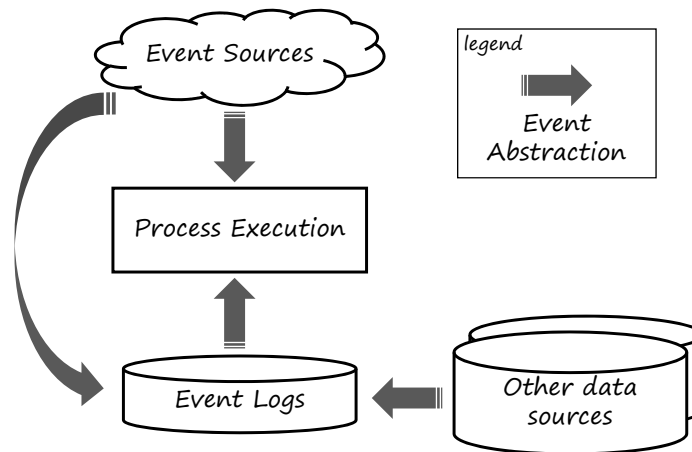


Figure 16: Applications of event abstraction in context of process execution.

streams. The raw events produced by the event sources are clustered together, using temporal or semantic information. Afterwards, the clustered events are used to derive business events for consumers. The notion of deriving information ready to be analyzed from events that are recorded at a lower granularity is known as *event abstraction* [74].

Event abstraction is used in several application domains to hide the complexity of event aggregation and other operations (explained later in [Section 3.4](#)) done on raw event streams. In process mining, this is vastly used to prepare the event log for process discovery. The data elements are grouped by means of clustering [54] and supervised learning techniques [119], to map to a single event representing an activity state change. More advanced techniques include exploring behavioural structures that reveal activity executions via identifying control flow patterns such as concurrency, loops, and alternative choices [83]. Natural language processing enriched with contextual information [11, 106] is also a popular technique for event abstraction when a process model is available. Further, insight about the domain and context of process execution leads to successful event abstraction that enables mapping sensor data to engine logged events [114]. Since the context data is not always available in a single database, event abstraction on several data stores used by process execution might be efficient [16].

The areas with process execution in the center of application for event abstraction is shown in [Figure 16](#) where the arrows depict use of event abstraction techniques. The above mentioned works in the process mining area concentrate on the event hierarchy from low-level data elements from several sources to higher-level events logged by the process engine and from the event log to activity execution in a process. On the contrary, the focus of this work is on event abstraction applications from the event sources to process execution [67]. Using the term abstraction in this context signifies coming up with the higher-level events required

| Online Shop | | |
|-----------------------|-----------|--------------------------------------|
| Timestamp | Location | Status of Packet |
| ● 2019-02-27 17:19:32 | | Shipped |
| ● 2019-02-27 14:56:00 | Warehouse | Package left warehouse |
| ● 2019-02-27 08:54:34 | Warehouse | Awaiting shipment |
| ● 2019-02-26 17:55:55 | Warehouse | Awaiting packaging |
| ● 2019-02-26 17:15:05 | Warehouse | Order confirmed and Awaiting picking |
| ● 2019-02-26 17:14:09 | Warehouse | Order paid successfully |
| ● 2019-02-26 17:13:10 | Warehouse | Order submitted |

Customer

Packet dispatched

Figure 17: Event abstraction for order tracking in an online shop.

by the event consumers such as process engines from the layers of low-level events generated by the event sources such as sensors.

A real-life example of such an application of event abstraction is visualized in Figure 17. Upon receiving an order, an online shop tracks the status of the packet. However, only when the order is confirmed, picked, packaged, and shipped; the customer gets notified about the packet being dispatched. Here, the internal events are important for the online shop to monitor that the shipment of the packet followed correct steps, but they are too fine-granular for the customer.

To realize event abstraction, the EPAs operate on event streams generated by the event sources. An *event stream* consists of a set of events of the same event type. Performing operations on events can be done in different multitudes depending on the *event hierarchy* needed to come up with the higher-level business event from the low-level raw events. It can be classified in following three categories: *Simple Event Processing (SEP)* — operations are done on a single event, *Event Stream Processing (ESP)* — more than one events from the same event stream is aggregated, and *Complex Event processing (CEP)* — multiple events from multiple event streams are acted on. The event streams with different possibilities of processing are shown in Figure 18.

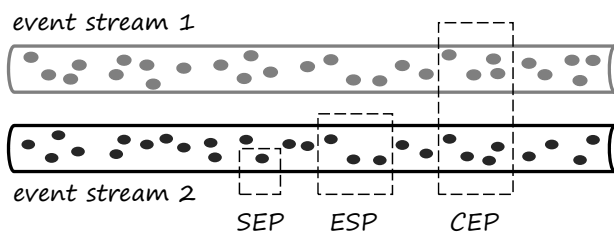


Figure 18: Stream processing operations on events.

Often, several EPA functionalities are consolidated in an Event Processing Platform (EPP). EPPs are able to connect to different event sources, perform complex event processing operations on the event streams, and notify the BPMS about a specific happening. We use the

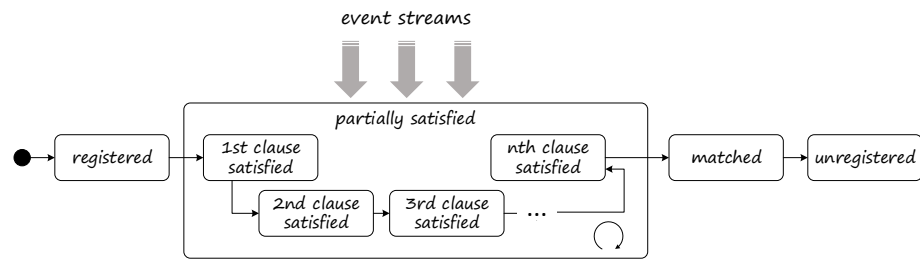


Figure 19: State changes for an event query in an event processing platform.

terms EPP and CEP platform interchangeably in the thesis. The advantages and underlying technicalities of integrating a CEP platform to a process engine are discussed further in [Chapter 5](#).

A CEP platform evaluates incoming event streams based on predefined rules for event abstraction, known as *event query*. The processing of an event query is represented as a state transition diagram in [Figure 19](#). Once the query is registered at the platform, it is checked for a match on every event occurrence across the connected event streams. An event query can have several clauses similar to Structured Query Language (SQL) queries. Once the first clause is satisfied, the query enters the state partially satisfied. Depending on the complexity, the query stays in this state for a while, although the internal state changes with each clause being satisfied. As soon as the last (n-th) clause is satisfied, the query is considered to be matched. A query can be matched several times until it is unregistered from the platform, visualized with the loop sign inside the partially satisfied state. The next section introduces the complex event processing techniques that can be applied on event streams to build an event hierarchy by satisfying clauses of an event query.

3.4 EVENT PROCESSING TECHNIQUES

Event processing operations can be classified in three major categories, namely *filtering*, *grouping*, and *derivation*. A brief introduction about the EPAs are given below. All the CEP techniques described below can be implemented in flexible order and combination. [Figure 20](#) shows an example event processing network. We have two event producers and one consumer. The filter agent takes the event streams from both the producers as input. Based on the filter expression (shown as the round corner rectangle inside the EPA), it filters in a subset of input events and feed to the group EPA. The group EPA then clusters the filtered events according to the grouping criteria. Next, the derive EPA composes a higher level event from each group and sends them to the consumer.

Query languages (similar to SQL) are built for the specific purpose of writing queries that can process the event streams. We follow the syn-

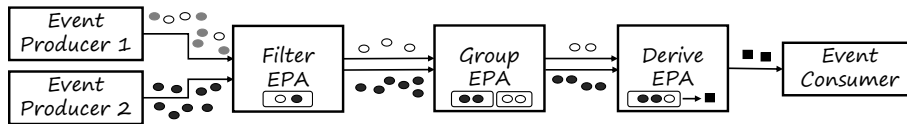


Figure 20: An example EPN showing the different components.

tax and semantics of ESPER Event Processing Language (EPL)², which is a declarative domain specific language based on SQL, designed for processing events with time-based information. As a running example throughout this section, let us think of a bank issuing credit cards to customers who also have a standard account at the bank. Each credit card transaction is recorded to the bank with transaction id, customer name, the status of the card (silver/gold/platinum), the amount credited along with the purpose, and the location (country) where the transaction was initiated. The event type Transaction is represented as XML schema definition in Listing 1.

Listing 1: Event type definition of Transaction

```

<?xml version="1.0" encoding="utf-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns="Transaction.xsd" targetNamespace="Transaction.xsd"
  elementFormDefault="qualified">
  <xs:element name="Transaction">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="transactionId" type="xs:string"
          minOccurs="1" maxOccurs="1" />
        <xs:element name="customerName" type="xs:string"
          minOccurs="1" maxOccurs="1" />
        <xs:element name="cardStatus" type="xs:string"
          minOccurs="1" maxOccurs="1" />
        <xs:element name="creditAmount" type="xs:double"
          minOccurs="1" maxOccurs="1" />
        <xs:element name="purpose" type="xs:string"
          minOccurs="1" maxOccurs="1" />
        <xs:element name="location" type="xs:string"
          minOccurs="1" maxOccurs="1" />
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>

```

FILTERING. Filtering is a stateless operation performed on a single event stream. It is done based on the evaluation of certain filter expressions on event attributes. If the filter expression evaluates to true,

² <http://esper.espertech.com/release-5.5.0/esper-reference/html/index.html>

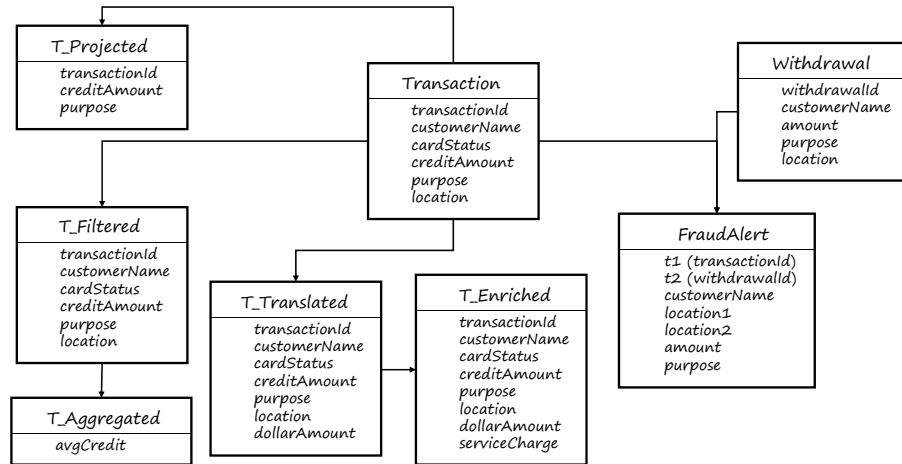


Figure 21: Event hierarchy showing Transaction and events derived from it.

then the event is sent to the output stream, otherwise it is discarded. The bank might want to know all credit card transactions done by their platinum customers. There, an event filter can be implemented with filter expression `Transaction.cardStatus = 'Platinum'`.

GROUPING. Grouping EPAs take single or multiple event streams as input and produces clusters of events as output. Grouping can be performed based on time, number of events, location, and attribute values. In context of the credit card example, fixed time window (all transactions during weekend) or sliding time window (transactions in each hour) can be used as temporal grouping.

Similar to the time interval, events can also be used to determine the interval boundaries. In this case, continuous event window or sliding event window can be defined. A customer can be charged for the transactions initiated each week. This is an example for continuous event window. On the other hand, to be alerted of misuse of the credit card, every three ATM withdrawal events following an overseas withdrawal can be monitored. Sliding event window will be appropriate for tracking the withdrawals in this case.

Related to the credit card scenario, location based grouping can also be useful, e.g., all withdrawals outside Euro zone should be grouped to be charged for currency conversion. Last but not the least, the events can be grouped based on a specific attribute value. If the bank providing credit cards wants to know how many transactions over 500 Euro take place on average for the platinum card holders, they might group all filtered events with `Transaction.creditAmount > 500`.

DERIVATION. The derivation EPAs can take single events or groups as input and modify them in several ways, as shown in Figure 21. A *project* EPA takes a single event, selects a subset of its attributes, and produces a corresponding event. Here, the number of attributes are

reduced, but no attribute value is changed. For example, the bank might want to know how much amount is credited for which purpose such as online shopping or travel booking. Therefore, they generate a projected event from each Transaction event as presented in [Listing 2](#).

Listing 2: Example of Project EPA

```
INSERT INTO T_Projected
SELECT t.transactionId as transactionId,
       t.creditAmount as creditAmount,
       t.purpose as purpose
FROM PATTERN[every t=Transaction];
```

On the contrary, an *enrich* EPA adds more attribute to an event, adding data from external sources. A *translate* EPA can also add attributes by modifying an existing attribute value, rather translating an attribute value to a corresponding value. For example, the amount in a credit card transaction can be changed from Euro to Dollar using translation. Afterwards, service charge for conversion can be added according to the bank rate stored separately, as shown in [Listing 3](#) followed by [Listing 4](#).

Listing 3: Example of Translate EPA

```
INSERT INTO T_Translated
SELECT t.transactionId as transactionId,
       t.customerName as customerName,
       t.cardStatus as cardStatus,
       t.creditAmount as creditAmount,
       t.purpose as purpose,
       t.location as location,
       (t.creditAmount)*1.14 as dollarAmount
FROM PATTERN[every t=Transaction];
```

Listing 4: Example of Enrich EPA

```
INSERT INTO T_Enriched
SELECT tr.transactionId as transactionId,
       tr.customerName as customerName,
       tr.cardStatus as cardStatus,
       tr.creditAmount as creditAmount,
       tr.purpose as purpose,
       tr.location as location,
       tr.dollarAmount as dollarAmount,
       (tr.dollarAmount)*rate as serviceCharge
FROM PATTERN[every tr=T_Translated];
```

An *aggregation* EPA takes groups of events as input. A single event is derived from one group of events, aggregating them based on certain attribute values. Calculating average credit amount for the platinum card

holders on a day can be an example of aggregation, as shown in the listing below. Other operations can be to calculate the maximum/minimum/total amount, and the number of transactions.

Listing 5: Example of Aggregation EPA

```
INSERT INTO T_Aggregated
SELECT Avg(creditAmount) as AvgCredit
FROM T_Filtered.win:time(24 hour);
```

Compose EPA, on the other hand, takes input events from multiple event streams and comes up with a composed event as output. These events from different streams can be joined based on common attribute values, can be used to identify a pattern based on their order of occurrences, or can be calculated to identify a certain trend, among many other possibilities [34, 127]. A fraud detection rule can be brought up in this context. As obvious, along with the credit card transactions, the bank also records all the ATM withdrawals for its customers. Now, if a withdrawal is observed within an hour of a credit card transaction where the location of withdrawal is a different country than the location of initiating the transaction, a misuse of the card is suspected.

The event type `Withdrawal` and `FraudAlert` are defined in Listing 6, whereas the rule for composing the fraud alert can be written using Esper EPL is shown in Listing 7.

Listing 6: Event types `Withdrawal` and `FraudAlert`

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
xmlns="Withdrawal.xsd" targetNamespace="Withdrawal.xsd"
elementFormDefault="qualified">
<xs:element name="Withdrawal">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="withdrawalId" type="xs:string"
        minOccurs="1" maxOccurs="1" />
      <xs:element name="customerName" type="xs:string"
        minOccurs="1" maxOccurs="1" />
      <xs:element name="amount" type="xs:double"
        minOccurs="1" maxOccurs="1" />
      <xs:element name="purpose" type="xs:string"
        minOccurs="1" maxOccurs="1" />
      <xs:element name="location" type="xs:string"
        minOccurs="1" maxOccurs="1" />
    </xs:sequence>
  </xs:complexType>
</xs:element>
</xs:schema>
```

```

<?xml version="1.0" encoding="utf-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
xmlns="FraudAlert.xsd" targetNamespace="FraudAlert.xsd"
elementFormDefault="qualified">
<xs:element name="FraudAlert">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="t1" type="xs:string"
        minOccurs="1" maxOccurs="1" />
      <xs:element name="t2" type="xs:string"
        minOccurs="1" maxOccurs="1" />
      <xs:element name="customerName" type="xs:string"
        minOccurs="1" maxOccurs="1" />
      <xs:element name="location1" type="xs:string"
        minOccurs="1" maxOccurs="1" />
      <xs:element name="location2" type="xs:string"
        minOccurs="1" maxOccurs="1" />
      <xs:element name="amount" type="xs:double"
        minOccurs="1" maxOccurs="1" />
      <xs:element name="purpose" type="xs:string"
        minOccurs="1" maxOccurs="1" />
    </xs:sequence>
  </xs:complexType>
</xs:element>
</xs:schema>

```

Listing 7: Example of Compose EPA

```

INSERT INTO FraudAlert
SELECT t.transactionId as t1,
       w.withdrawalId as t2,
       t.customerName as customerName,
       t.location as location1,
       w.location as location2,
       w.amount as amount,
       w.purpose as purpose
FROM PATTERN[every(t=Transaction->w=Withdrawal)].win:time(60 min)
WHERE t.customerName=w.customerName
AND t.location!=w.location;

```

To test the event processing operations, all the event types and event processing rules discussed above have been defined in Unicorn³, an event processing platform that uses the Esper engine and EPL for executing queries. Upon receiving a Transaction event followed by a Withdrawal event, Unicorn produces the list of events shown in [Figure 22](#).

³ <https://bpt-lab.org/unicorn-dev/>

(We currently limit the display to the last 10000 events, i.e. in the streaming database may be more events)

Filter ID Condition =

Filter Reset Delete Select All

| ID | Timestamp | EventType | Values |
|-----|----------------------------|-------------------|--|
| 155 | 2019-03-21 09:31:48.047 | FraudAlert (27) | amount=391.33, purpose=shopping, location1=DE, location2=IT, 11=DE290, customerName=Julia Schindt, 12=IT387 |
| 154 | 2019-03-21 09:31:48.036 | Withdrawal (22) | amount=391.33, withdrawalId=IT387, purpose=shopping, location=IT, customerName=Julia Schindt |
| 153 | 2019-03-21 09:31:43.511 | T_Enriched (26) | serviceCharge=-2.388528, dollarAmount=159.2352, purpose=insurance, location=DE, creditAmount=139.68, customerName=Julia Schindt, transactionId=DE290, cardStatus=P |
| 152 | 2019-03-21 09:31:43.505 | T_Projected (24) | purpose=insurance, creditAmount=139.68, transactionId=DE290 |
| 151 | 2019-03-21 09:31:43.495 | T_Translated (25) | dollarAmount=159.2352, purpose=insurance, location=DE, creditAmount=139.68, cardStatus=P, transactionId=DE290, customerName=Julia Schindt |
| 150 | 2019-03-21 09:31:43.487 | T_Filtered (23) | purpose=insurance, location=DE, creditAmount=139.68, customerName=Julia Schindt, cardStatus=Platinum, transactionId=DE290 |
| 149 | 2019-03-21 09:31:43.475 | Transaction (21) | purpose=insurance, location=DE, creditAmount=139.68, customerName=Julia Schindt, cardStatus=P, transactionId=DE290 |

© 2012-2015 Business Process Technology group

INSTITUTO TECNOLÓGICO DE AERONÁUTICA Y ESPACIO
CONSEJO SUPERIOR DE INVESTIGACIONES CIENTÍFICAS
EUROPEAN UNION
UNICORN

Figure 22: Screenshot from Unicorn showing generation of complex events following pre-defined rules.

RELATED WORK

“What do researchers know? What do they not know? What has been researched and what has not been researched? Is the research reliable and trustworthy? Where are the gaps in the knowledge? When you compile all that together, you have yourself a literature review.”
 – Jim Ollhoff, *How to Write a Literature Review*¹

4.1 OVERVIEW

In this chapter, we outline the existing research that are closely relevant to the work presented in this thesis. To begin with the literature review, the status of using external events in BPM is discussed in [Section 4.2](#), having a focus on the IoT context. This is helpful for positioning the thesis with respect to the significance of the contribution and the challenges associated with it. As narrated in [Chapter 1](#), this thesis builds a detailed subscription management system based on a basic framework integrating external events into business processes. Therefore, the basic integration work involving CEP and BPM are explored in [Section 4.3](#). This section highlights the use of concepts from complex event processing area in different phases of business process management lifecycle, and the integrated applications that exploit concepts from both CEP and BPM fields. Next, the event subscription mechanism and event buffer middlewares from BPM world as well as related research fields are discussed in [Section 4.4](#). The chapter ends with a summary of related work in [Section 4.5](#). The overview of the research areas covered in this chapter is given in [Figure 23](#).

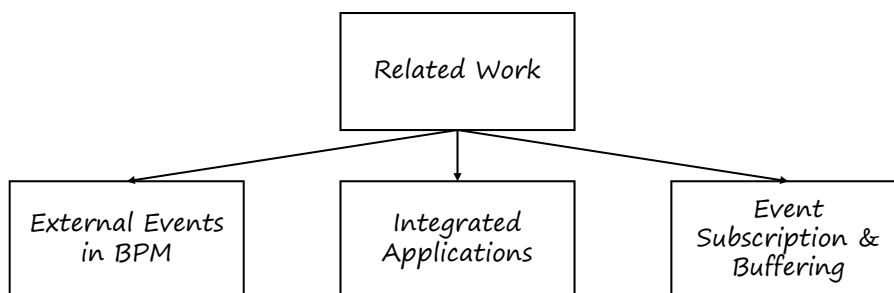


Figure 23: Overview of discussed research areas as related work.

¹ <https://www.goodreads.com/book/show/19072895>

4.2 EXTERNAL EVENTS IN BPM

Process executions are often supported by dedicated IT systems and the transitional events are logged by enterprise software systems to record the state changes in the cases correlated to the business processes. This results in event streams that can be used as input for several applications related to business process analytics [40] such as process discovery, compliance checking with respect to process specification violations, and finding the performance bottlenecks. Ever since the IoT enabled big data got accessible by the IT systems, new business scenarios are raising that can exploit the era of digitization [53]. This leads to use not only the transitional events, but also the external events as source of information to have better process analytics. However, using IoT concepts in BPM and (vice versa) calls for a detailed understanding of the IoT field. [38] shows the important building blocks for IoT business model based on an extensive survey, represented with the Business Model Canvas². To understand the actors in IoT applications and the interactions between them, detailed surveys are available that list the commonly used technologies, concepts, and architectures for an IoT project [6]. More focused classification applicable in a BPM context can be found in [82] where the authors traverse through a large number of IoT scenarios and suggest certain usecases where BPM and IoT can benefit from each other. For instance, to employ a BPMS as the controller of the interacting things, i.e., the sensors, actuators, and complex devices can help to monitor the big picture of a distributed IoT application.

The use of external events in business processes in an IoT context is also being explored in multiple recent work. In [112], the authors suggest that IoT applications are data-intensive where the data needs pre-processing for getting ready for analysis. These data might trigger business activities in real time, communicated through ubiquitous communication means to the users. To handle these features of IoT environment, process models need to be re-engineered — either with extended modelling elements or with rearrangement of the activities. To bridge the abstraction gap of raw data and business events in execution level, use of context variables is suggested. In a related publication realizing the requirements [111], the authors argue that for an IoT-aware process execution setup, a BPMS must be aware of current values of IoT objects while the defined variable must be referenced in the executed process model. Moreover, the responsible users executing the tasks must be notified on mobile devices in real time about the context-specific knowledge according to the access rights of the users. As an evaluation of the concepts, the authors conducted a case study in corrugation industry. The real-world objects such as the IoT devices were connected to human operators using wearable devices. A BPMS controlled the data exchange via a communication middleware that includes an IoT data

² <https://www.strategyzer.com/canvas/business-model-canvas>

server. A group of operators with wearable IoT devices received status of the ongoing activities, remaining time to start the next activity, and error messages; where the other group did not use any wearable device. The study shows application of the IoT enhanced BPMS leads to less machine stops because users had context information that helped them to recognize the reactions needed to a situation in an efficient way.

IoT-aware process execution

Events are usual representations for exceptions in business process context. Recent studies show that exceptions are an inevitable yet highly influencing factor when it comes to performance of a process [39]. The survey classifies the exception patterns that might unsettle the normal control flow and evaluates the throughput time for each of the patterns. The case study presented in [129] shows similar results about handling unprecedented events and process performance for declarative process execution as well. Here, two groups of students (non-experts) had to execute a journey in order to meet a predefined business value, keeping in mind certain constraints. One group had no unexpected event, whereas the other group encountered with increasing time of activities and traffic disruption that caused the participants to rearrange the journey partially. Business value and number of failed journeys were calculated as evaluation parameters. The results show that the participants having to adapt to unforeseen environmental occurrences had significantly higher difficulty to plan the journey.

While the advantages of integrating IoT and BPM are already established, the challenges are being investigated as well. In [108], it is claimed that BPM is still not ready to utilize the huge benefits big data analytics can offer. The authors classify data-intensive operations in BPM into following categories: event correlation, process model discovery, conformance checking, runtime compliance monitoring, process enhancement, and predictive monitoring. They propose the term *process footprints* that include process model, transitional events logged by the IT systems, external events i.e., sensor data received during process execution, as well as any social media interaction or user interaction that happens in the enactment phase. All these source of information are highly valuable since delay or non-detection of any of those can have a harmful impact on the organizations.

An exhaustive discussion on the mutual benefits and challenges of IoT and BPM is found in [61]. The list of challenges cover the complete integration aspects such as the physical deployment of sensors in a cost-effective way, visualization of both automated activities and manual tasks, considering new scenarios that are sometimes unstructured and distributed, having loosely coupled small fragments of processes rather than a big predefined process, coming up with specifications for the abstraction level, and autonomous level. Also, the social role of the process participants is discussed in this context. The technical challenges mention that large amount of available data demands advanced guide-

Challenges of integrating IoT & BPM

lines for event handling, online conformance checking, and resource utilization.

Another significant research work by Soffer et al. [117] explores the challenges and opportunities with respect to integrating event streams and business process models. The authors detect four key quadrants for combining CEP and BPM according to BPM lifecycle phases. Enriching expressiveness of process models in design phase and using CEP constructs for process mining in evaluation phase look at the integration aspects from CEP to BPM. In configuration phase, deriving CEP rules from process models is proposed to determine which activities to monitor and which event occurrences are important to notice. Further, executing processes via CEP rules in execution phase is proposed. This essentially means mapping external (sensor) events to transitional events to automate beginning and termination of an activity without involving the user. The configuration and execution phase here focus on the flow of control from BPM to CEP. For all of the key quadrants, challenges and benefits are discussed with the support of existing work.

Having discussed the aspects and challenges of integrating events in business processes to enable the collaboration with big data, IoT, and CEP; the rest of the section showcases specific applications where the concepts from CEP has been used in BPM scenarios.

4.3 INTEGRATED APPLICATIONS

The concept of event driven business process management (EDBPM) has been already in discussion for more than ten years [128]. So far, several approaches have been presented aiming to extend BPMN with modeling constructs for concepts of CEP [7, 13, 45]. Krumeich et al. give a comprehensive overview of the state-of-the-art in using events for event-driven architectures (EDA) or BPM in [67]. While some of those propose conceptual enhancement, some provide execution support in the form of an engine. For instance, the work by Kunz et al. [68] suggest the mapping of event processing queries to BPMN artifacts. The authors propose to map the parts of an EPL query to an *event-sensitive BPMN element* (BPMN service task). The task reads the data objects as incoming data flow to specify FROM clause, takes data object collections to realize the SELECT clause. Once the data values match the WHERE clause of the EPL query, a boundary conditional event is fired to trigger the exceptional flow. Another approach in the same line is by Appel et al. [7] where the authors integrate complex event processing into process models by means of event stream processing tasks that can consume and produce event streams. An approach from the other direction is presented in [132], where event queries are derived from the control flow of a process model, deploy them to an event engine and use them to find violations of the control flow. A similar derivation of event queries from the process model is done by [10].

The authors in [13] suggest to integrate descriptions of event patterns into process modeling languages and consequently extend engines to handle such patterns, as they argue that both process and events are integral aspects to be captured together. They present a catalog of event patterns used in real-world business processes and find that most of those patterns are neither supported by BPEL nor BPMN. For example, they identify support for event hierarchies, i.e. abstraction of low-level events into high-level business events, as an important feature which is not yet supported. Similar to modelling complex events, configuring process models to model the interactions in an internet of things context is addressed in [118]. The author proposes modelling IoT resources w.r.t. shareability and replicability and builds a cross-domain semantic model with concepts from both BPM and IoT domain, validated by prototype tools.

Besides design level integration, several execution level applications have been developed to take advantage of event processing concepts for business processes. For instance, [45] propose an IT solution architecture for the manufacturing domain that integrates concepts of SOA, EDA, business activity monitoring (BAM), and CEP. They suggest to embed event processing and KPI calculation logic directly into process models and execute them in an extended BPMN engine. The authors sketch such an engine for executing their extended process models, but refrain from giving technical details. However, they suggest that some processes collect simple events, evaluate and transform them, and provide high-level events for use in other process instances, realizing an event hierarchy.

For processes, in which some tasks are not handled by the process oriented information system (POIS), monitoring of events can be used to determine the state of these tasks, e.g. to detect that an user task terminated. When a process is not supported by a POIS at all, monitoring can still capture and display the state of the process by means of events. For example, Herzberg et al. [56] introduce *Process Event Monitoring Points (PEMPs)*, which map external events, e.g. a change in the database, to expected state changes in the process model, e.g. termination of a task. Whenever the specified event occurs, it is assumed that the task terminated, thus allowing to monitor the current state of the process. The authors separate the process model from the event processing and allow the monitored events to be complex, high-level events. The approach has been implemented, however the event data is not used by and does not influence the process activities. Rather, the engine uses them to determine the current state of the process instance. Similar frameworks for predictive monitoring of such continuous tasks in processes are presented in [23, 127]. The framework by Cabanillas et al. [23] defines monitoring points and expected behavior for a task before enactment. Event information from multiple event streams are captured and aggregated to have a meaningful insight. These aggre-

Application areas

gated events are then used to train the classifier and later the classifier can analyze the event stream during execution of the task to specify whether the task is following a safe path or not.

Processes from the logistics domain contain long-running activities, and therefore, needs continuous monitoring (e.g. shipment by truck). In these scenarios external events, e.g. GPS locations sent by a tracking device inside the truck, can provide insight into when the shipment task will be completed. Appel et al. [7] integrate complex event processing into process models by means of event stream processing tasks that can consume and produce event streams. These are used to monitor the progress of shipments and terminate either explicitly via a signal event or when a condition is fulfilled, e.g. the shipment reached the target address. While these tasks are active, they can trigger additional flows if the event stream contains some specified patterns. The authors provide an implementation by mapping the process model to BPEL and connecting the execution to a component called *eventlet manager* that takes care of event processing. [18] takes one step further and shows how the event driven monitoring influences the process execution in a multi-modal transportation scenario. The authors use a *controller* that is used as a dashboard for controlling and monitoring the transportation processes that is connected to an event processing platform. The events impacting the transportation are visualized in the dashboard. In the background, the processes are executed in Activiti³, a BPMN-based process engine. In this context, another publication [17] explains a methodology for using events in business process monitoring and execution. Based on a usecase in logistics domain, requirements and lessons learned are described for process design, execution, and required event processing. Here, the authors argue that event subscription information should be annotated to the process model such that the subscription queries are automatically registered to the event platform by a process engine.

Process monitoring

Another advanced monitoring approach is proposed in the recent research [85]. The author suggest artifact-driven business process monitoring that considers the changes in the state of the artifacts participating in a process to detect when activities are being executed. Using this technology, smart objects can autonomously infer their own state from sensor data. This approach is helpful for exploiting IoT paradigm while monitoring inter-organizational processes where a monitoring platform can not cross the border of an organization. *SMARTifact*, an artifact-driven monitoring platform has been built as a prototype.

Pufahl et al. in [101] present another application of complex event processing, used for efficient decision making in business processes. The authors here consider updated event information to *re-evaluate decision* till the point a reset is possible. To receive the updated events, the subscription query is enriched with the decision logic behind the current execution path and only interrupts the execution if the event

³ <https://www.activiti.org/>

data result in a different decision. More applications regarding event processing in BPM are found in [5, 33, 91, 113, 138].

While the above research shows a large sample of applications integrating events and processes, research are being done to enhance the performance of such applications as well. Fardbastani et al. looks at a different aspect of process monitoring in presence of events [50]. The authors argue that distribution of CEP responsibilities are needed to enable large scale event-based business process monitoring. They suggest to either distribute parts of event abstraction rules to multiple EPAs, or to cluster CEP engines on independent computational nodes. A distributed architecture named *dBPM* is developed based on the latter idea. This includes several BPMSs, coordinators that control resource utilization by routing the events from a BPMS to assigned CEP node(s), the CEP nodes, and the event consumers. A case study involving processes of customs administration shows increasing the number of CEP nodes leads to almost linear increase in the throughput of process monitoring.

*Performance
optimization*

The approach proposed by Weidlich et al. [133] looks at efficient ways of event pattern matching from another perspective. The authors here map different pattern matching rules to behavioral constraints that are derived from the process models. Based on the knowledge extracted from the process model specification, the probable event occurrence sequences are written in this approach. The constraints elicited from behavioural profiles [131] are considered while writing the valid event sequences. This, in turn, leads to an optimized performance with respect to event pattern matching following *execution plans* that include temporal and context information for transitional events as well as the subscription mechanism (push/pull) used to receive an event from the event platform.

4.4 FLEXIBLE EVENT SUBSCRIPTION & BUFFERING

This section focuses on the specific work related to event subscription and buffering. From a business process perspective, there is very limited research that includes flexible event subscription semantics. The existing integrated applications for event-process integration [17, 113, 140] and distributed systems [71, 125] follow the basic publish-subscribe paradigm [58] to implement subscription and notification of events. The state-of-the-art BPMN process engines [2, 24], on the other hand, realize the BPMN semantics of listening to an event when the control flow reaches the event node and ignore the scenarios demanding increased flexibility in event handling.

However, there are few significant ones that have been great inspiration to our work. For instance, the causal relationship stated by Barros et al. in [13] is the one we have extended for more explicit event handling. A comparison between the dependencies suggested by the authors in [13] and the one followed by the event handling model

proposed in this thesis is found in [Section 6.2](#). The *CASU framework* proposed by Decker and Mendling [36] is another work that has highly influenced our work. The framework is built to conceptually analyze how processes are instantiated by events. The name of the framework is an abbreviation that considers when to create new process instances (C), which control threads are activated due to this instantiation (A), which are the remaining start events that the process instance should still subscribe to (S), and when should the process instance unsubscribe from these events (U). While this essentially talks about point of (un)-subscription as well as the duration of a subscription, our work gives more detailed and formal semantics. Moreover, the authors in [36] restrict themselves to the start events, whereas we focus on the intermediate catching events that bring in much more variety and complexity in semantics.

While on the one hand event subscription is neglected in BPM community, on the other hand, this has gained a lot of attention in the complex event processing research. Event-based systems such as *JEDI* [31], *Hermes* [99], *STEAM* [84] and event processing engines such as *Cayunga* [21] and *ESPER* offer detailed description about the subscription mechanism and the event processing middlewares. A formal semantics of the middleware designs for Enterprise Integration Patterns (EIP) is given using coloured Petri nets in [49].

In publish-subscribe system, subscriptions can follow different models [12], as listed below:

- *Topic-based*: the subscriber shows interest in a specific topic and starts getting notified about all the events related to that topic [14],
- *Content-based*: the subscriber specifies filtering conditions over the available notifications [4, 31, 99],
- *Type-based*: a specific event type is subscribed to [47],
- *Concept-based*: subscription is done on a higher abstraction level, without knowing the structure or attribute of the events [29], and
- *Location-aware*: location-aware notifications supporting the mobile environment [84].

Advanced event based system such as *PADRES* [60] takes into account the event information that has happened in the past in addition to the traditional publish-subscribe mechanism that considers only events that might happen in the future. This event processing network consists of several brokers which are essentially event processing engines. These engines can talk to the next neighbors in the network using content-based publish-subscribe model. For accessing historic events, the brokers are attached to databases. The subscribers can fetch the stored data from the databases attached to the publishers. The unsubscription is done once the requested result set has been published.

Apache Kafka⁴ is a large-scale distributed streaming platform that is renowned in industries as well as for academic applications. Kafka

⁴ <https://kafka.apache.org/>

implements topic-based subscription where consumers can subscribe to multiple topics. These topics are stored as records with unique key-value pair with a timestamp. The combination of messaging, storing, and stream processing functionalities added with the fault-tolerant, durable exchange of records offered by Kafka makes it a popular event stream buffer.

While most of the event stream processing platforms offer buffering, Sax et al. [109] handles data arrival with out-of-order timestamps without explicit buffering. The approach prioritizes low processing of online data handling over traditional data buffering and reordering of records. The dual streaming model proposed by the authors consists tables that capture the updates of data arrival for a streaming operator such as an aggregation operator with specific grouping/filtering conditions. In turn, a changelog stream updates the records for an operator whenever a new record with the same key is available. This retains the consistency in the logical and physical order of data. The model is implemented in Apache Kafka to show the feasibility of the concepts.

4.5 SUMMARY

Transitional events, i.e., events logged by process engines and external events, i.e., events received from environmental sources such as sensors, as well as hybrid approaches including both are used extensively for process analytics [40]. This involves plenty of applications for process discovery, process conformance checking, and process monitoring [1, 7, 17, 23, 56, 85, 127]. Concepts from CEP research field has already gained popularity for modeling [7, 13, 45, 68] and executing [10, 132] these integrated applications. Data stream processing languages such as Continuous Query Language (CQL) [8], and complex event processing languages such as Esper EPL [44] offer a big range of operations to aggregate the events based on timestamps and attribute values. More advanced event specification language like TESLA [32] provide rules that are formally defined and considers the requirements of larger set of application scenarios. Built on publish-subscribe communication style, the distributed agent based systems such as PADRES [60] maintains the strong influence of event processing at its core.

Recently, integrated applications are being developed that enable the event sources such as sensors to talk to each other as well as to talk to a controller, e.g., a BPMS [82, 112]. This has two reasons: On one hand, the advent of IoT era makes the event information highly accessible for the IT systems [6, 38], and on the other hand, new business scenarios are emerging that involve highly intense communication between the process participants and execution environment [53]. At the same time, handling big data efficiently in BPMS [108], load balancing of CEP operations for scalability [50], and optimization of event processing [133] are

emerging to address the challenges of integrating external events and event processing techniques in business process management [61, 117].

Since the interaction plays a pivotal role in event driven process execution, it also demands a more flexible event handling mechanism [81]. However, even if there are several applications consisting of BPMN processes that exploit the information carried by the external events, less attention is given for a standardized generic event handling model.

There exist significant amount of research work for flexible process execution such as case management [35, 57, 70, 103, 110]. Also, there are approaches that propose redesigning processes to avoid unwanted delays and deadlocks [30, 78]. Nevertheless, only a small set of work considers flexibility in terms of event subscription management. On the contrary, the complex event processing field is well enriched with concepts and technicalities for subscription-notification, event streaming, and event buffering. Our work addresses these shortcoming and proposes an advanced event handling model for more flexible communication between processes and events.

Part II

CONCEPTUAL FRAMEWORK

INTEGRATING REAL-WORLD EVENTS INTO BUSINESS PROCESS EXECUTION

This chapter presents the conceptual and technical challenges to enable a basic end-to-end integration of external event information into business processes.

First, the requirements have been elicited. This includes design level requirements such as separation of business process specification and complex event processing techniques; as well as implementation level requirements such as event subscription and correlation. Later, the conceptual framework has been proposed to address those requirements. A prototypical implementation of the proposed framework is discussed in [Chapter 9](#). The integrated architecture has been published in “A Framework for Integrating Real-World Events and Processes in an IoT Environment” [80].

5.1 MOTIVATION & OVERVIEW

Business process management (BPM) and complex event processing (CEP) are well explored fields in their own right and it has already been established that they can complement each other significantly [46]. Lately, the development of Internet of Things (IoT) caused the availability of an abundance of data, also referred to as *big data explosion*. Business processes can take advantage of this era of digitization of physical properties and react to the environment as soon as there is a certain change that might impact the process flow. In other words, event information enhances business processes to be more flexible, robust, and efficient to take reactive measures.

On the other hand, complex event processing techniques provide means to filter, correlate, and aggregate raw events produced by the sensors to come up with a more meaningful business level event. The IoT raises business scenarios that were not possible before, such as remotely controlling devices in a *smart home*^{1,2}, facilitating thousands of gadgets in a *smart factory*³, or simply tracking valuable objects⁴. These scenarios include frequent communication between the devices, sensing data, interpreting it, and actuating the reactions. BPM concepts can be beneficial in controlling and monitoring these interactions [82]. Emerging applications of predictive monitoring in an IoT environment [5, 91]

IoT enabled business processes

¹ Samsung Family Hub.

<http://www.samsung.com/us/explore/family-hub-refrigerator/>

² Nest Learning Thermostat. <https://store.nest.com/product/thermostat/>

³ Daimler Smart Production. http://www-05.ibm.com/de/pmq/_assets/pdf/IBM_SPSS_Case_Study_Daimler_de.pdf

⁴ DHL Luxury Freight Tracking. http://www.dhl.fr/en/logistics/industry_sector_solutions/luxury_expertise.html

often employ CEP techniques to understand and detect the patterns of environmental occurrences that might lead the system to a bad state and complement it by triggering business processes for preventing the bad state in a proactive way.

The work presented in this thesis revolves around the idea of integrating contextual information, represented in form of events, during run-time of a business process execution. Even though there is lot of research going on individually in both the areas of BPM and CEP, there has been no solid guideline or framework to integrate these two worlds, conceptually as well as technically (cf. [Chapter 4](#) for more details). Hence, it was required to build an integrated architecture that encompasses the whole cycle of event-process communication; starting from aggregating raw events into business events, via detecting and extracting the event information to map it to process variables, up until reacting to the event following a business process specification.

The rest of the chapter is structured as follows: first, we identify the requirements to build an integrated architecture. The requirements are described with the help of a motivating example from the logistics domain in [Section 5.2](#). Based on the requirements, the integrated system architecture is presented in [Section 5.3](#). Finally, [Section 5.4](#) summarizes the chapter.

The integrated architecture presented in this Chapter includes event processing platform Unicorn (see [Section 9.1.1](#)), process engine Chimera (see [Section 9.1.3](#)), and process modeler Gryphon (see [Section 9.1.2](#)). Most part of Unicorn [18] and an initial version of Chimera, known as JEngine [55], was developed before the author started working at the chair of Business Process Technology⁵ at Hasso Plattner Institute, University of Potsdam. However, Unicorn and Chimera did not have any interaction yet and Gryphon did not exist either. The integration framework was developed in the context of an industry project with Bosch Software Innovations GmbH [20]. The work presented in this Chapter is jointly developed with *Marcin Hewelt*, partly as co-supervisors of Bachelor project for consecutive two years (2015-16 and 2016-17), and has been published in CoopIS 2017 [80].

Acknowledgements

5.2 REQUIREMENTS ANALYSIS

An extensive analysis of the aspects to consider and the challenges to overcome while integrating events and processes is the basis of the proposed architecture. First, the standard process engines are explored to learn about the state-of-the-art. To scope the work, we consider the process engines that implement the semantics for usage of activities and events in a process model according to the BPMN specification [94]. The literature survey discussed in [Chapter 4](#) gave us insight about the conceptual dimensions required for the integration framework. The

⁵ <https://bpt.hpi.uni-potsdam.de/Public/>

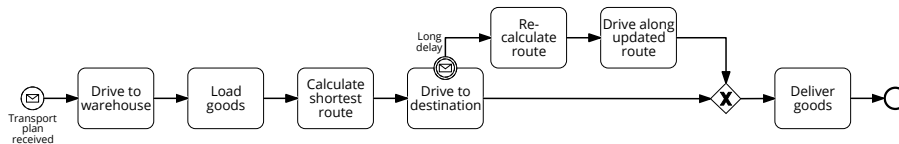


Figure 24: Usecase from the logistics domain

project partners and domain experts from both academia and industry contributed vastly to extract use cases that include communication between processes and their environment.

One such use case from the logistics domain is shown in Figure 24. The process starts when the truck driver receives the transport plan from the logistics company. Then she drives to the warehouse to load the goods. Once the goods have been loaded, the driver follows the shortest route to reach the destination. While driving, the driver gets informed in case there is a long delay caused by an accident or traffic congestion. If the notification for a long delay is received, the driver stops following the initial route and calculates an alternative way which might be faster. Note that in reality a driver can recalculate the route as often as she wants to while driving to the destination, but this is implicit in the driving activity; the explicit activity for recalculating the route is executed only when it is triggered by the long delay event. Once the destination is reached the goods are delivered and the process ends. This can be easily related to all of us driving on a highway, listening to the radio to be updated about the traffic situation, taking a detour if there is a traffic congestion. The more interesting business value here is that the logistics companies generally control the transportation of 50-200 trucks and they need to know the whereabouts of the trucks continuously to update the estimated time of delivery. Therefore, automating the process by using a process engine to execute and monitor the transportations is efficient. Having the above scenario as a basis, the requirements for using events in processes are identified and described in the remainder of this section.

Motivating example capturing importance of event integration

5.2.1 R1: Separation of Concerns

Using external information in business processes is essentially equivalent to connecting the two fields of business process management and complex event processing. Process engines could directly connect to event sources to get notifications. This may be done by querying their interfaces, listening to event queues, or issuing subscriptions. However, from a software engineering perspective, this design decision would dramatically increase the complexity of the engine. Also, it will violate established architectural principles like single responsibility and modularity. The separation of concerns, i.e., separation of process execution behavior and complex event processing techniques is therefore considered a major requirement.

Different event sources produce events in different formats (e.g., XML, CSV, JSON, plain text) and through different channels (e.g., REST, web service, or a messaging system). In the example scenario, the probable event sources are the logistics company, the GPS sensor in the vehicle, and the traffic API. Each of them might have their own format of published events. If the process engine needs to directly connect with event sources, it has to be extended with adapters for each of the sources to parse the events. After the event sources are connected, there is also the need for aggregating those events to generate the business event needed for the process. This yields yet another component in the process engine that retains the event processing techniques and operates on the event streams.

From an architectural point of view, to include all the event processing functionalities in a process engine will increase the complexity and redundancy of the engine to a great extent. From a logical perspective, process engines are meant for controlling the execution of process instances following the process specification, not for dealing with event processing. Besides, certain events can be interesting for more than one consumer. For example, the long delay event might be relevant not only for the truck drivers, but also for other cars following the same route. Having a single entity responsible for both process execution logic and event process techniques is, therefore, not a preferred option.

5.2.2 *R2: Representation of Event Hierarchies*

Simple event streams generated from multiple event sources can be aggregated to create complex higher-level events. One could propose to use BPMN parallel multiple events to represent the event hierarchy, at least to show the connection among simple and complex events. However, using that approach one cannot express the various patterns of event abstraction such as sequence, time period, count of events or the attribute values. Different patterns of event sequences are thoroughly discussed in [75]. A structured classification for composite events can be found in [13]. Expressing event processing patterns using process artifacts would complicate the process model and defeat its purpose of giving an overview of business activities for business users and abstract from underlying technicalities. As a user of BPM, one would be interested to see only the higher-level event that influences the process, rather than the source or the structure of the event. For example, the driver is only interested to know if there is a long delay that might impact her journey, but she does not care what caused the delay.

An event hierarchy essentially takes care of the flow of information from the low-level events to the higher-level events. As explained in [Chapter 3](#), single event streams can be filtered based on certain time windows, specific numbers of event occurrences, or attribute values of the events. Also, multiple events from multiple event streams can be aggregated based on predefined transformation rules to create complex

events relevant to a process. Exploiting event hierarchies, the process model includes only the high-level business events relevant for the process and easily understandable by business users. The model is not supposed to be burdened with details of event sources and abstraction techniques. However, the business event on top of the hierarchy needs to be mapped to the BPMN event modeled in the process.

5.2.3 *R3: Implementation of Integration*

The two requirements above address the logical distribution made from the architectural point of view and the conceptual mapping of event processing to process execution. Now we define the following technical requirements to realize the integration of events and processes from an implementation aspect.

R3.1: BINDING EVENTS. The business events modeled in the process model need to be correlated with the higher-level event defined by the event hierarchy in the CEP platform to make sure that the correct event information is fed to the process. For example, the driver should be informed only about delays on the route she is following.

R3.2: RECEIVING EVENTS. The process engine should listen to specific event sources to get notified once the relevant event occurs. Essentially, an event can occur at any time, often independent of the process execution status. However, unless the process explicitly subscribes to a specific event, the event occurrence will not be detected in the process engine. In other words, the driver must subscribe for the Long delay event modeled in [Figure 24](#) to get a notification about it.

R3.3: REACTING TO EVENTS. The consumption and further use of event information in later process execution should be provisioned. Sometimes, only the event occurrence is required to trigger certain reactions, such as instantiating a sequence of activities for exception handling. In other cases, the payload or the data carried by the events can also be valuable for process execution. For example, the driver might need to decide which alternative route is faster than the current one. The duration of the delay will be helpful for this decision making. Therefore, information carried by the events should be stored for later use in the process.

5.3 SYSTEM ARCHITECTURE

This section presents the architectural framework to enable an efficient integration of external events into business processes. The concepts are explained using the same example presented in [Figure 24](#). Our

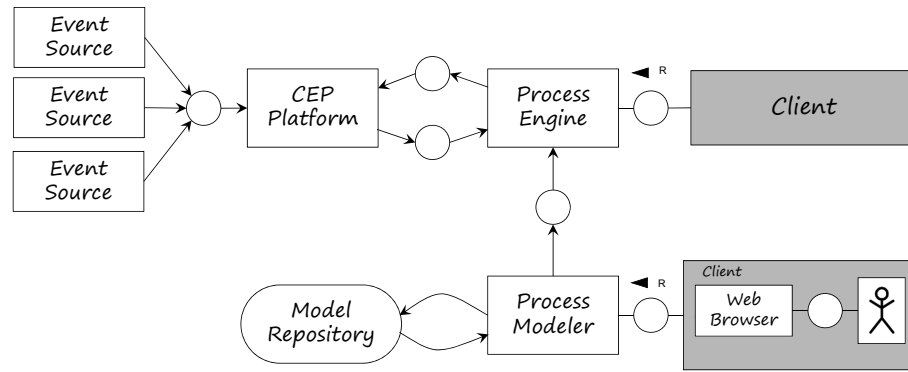


Figure 25: Proposed system architecture for event-process integration.

system architecture includes three components: *Gryphon*⁶, the process modeler; *Chimera*⁷, the process engine; and *Unicorn*⁸, the event processing platform [121]. Exhaustive information about the components, their features, and interplay are found in [Chapter 9](#).

Note that even though some technical details are discussed with examples based on these components, the integration architecture is independent of specific engines, platforms, technologies, and interfaces. The conceptual and technical solutions to address the requirements discussed above are described below. They can be implemented using any process modeler, process engine, and event processing platform communicating with each other following the proposed framework. The generic components along with the connections between them are visualized in [Figure 25](#).

5.3.1 Distribution of Logic

To address the issues raised by *R1: Separation of Concerns*, we propose to employ a complex event processing platform in addition to the process engine. CEP platforms are able to connect to different event sources and can perform further operations on event streams [56]. They can receive events, parse them, filter them and transform them to create new events. The event consumers can then subscribe to the event platform to be notified of the relevant events. This solution keeps the process execution and event processing logic separate, resulting in no extra overhead to the process engine. Since the CEP platform is a stand-alone entity, multiple process engines or other consumers can subscribe to a certain event, facilitating the reuse of event information. This separation of logic is also efficient from the maintenance perspective. If there is a need to change the event source or the abstraction logic, then the process model does not need to be altered.

6 <https://github.com/bptlab/gryphon>

7 <https://github.com/bptlab/chimera>

8 <https://github.com/bptlab/Unicorn>

EVENT PROCESSING LOGIC. According to the usecase model, we need two events for the transport process: a catching start event and a catching interrupting boundary event. The start event, sent by the logistics company, contains the location of the warehouse to load goods, the destination for delivery and the deadline for delivery. This is an example of a simple event which might be sent to the truck driver via email or as a text message directly from the logistics company. The boundary event, on the other hand, is a higher-level business event that needs to be aggregated from the raw events. This complex event is created in the event processing platform, in our case Unicorn. Since we did not have access to real “truck positions”, we used the sensor unit Bosch XDK developer kit⁹, a package with multiple integrated sensors for prototyping IoT applications. The sensors were used to mock the location of the truck. The unit sends measurement values over wireless network to a gateway. The gateway then parses the proprietary format of the received data and forwards it to Unicorn using the REST API. The traffic updates were received from Tomtom Traffic Information¹⁰. The next section discusses how the raw events from the connected event sources are aggregated to the business event.

PROCESS EXECUTION LOGIC. In our system architecture, the process engine is responsible for carrying out the deployment, enactment, and execution of processes. The Chimera process engine follows BPMN semantics for executing process specification. The processes are modeled in an additional editor before the models are deployed to the engine. Each occurrence of a start event triggers a process instance. Following the process model, the activities are carried out. The intermediate catching events are awaited once the corresponding event node is enabled. Upon occurrence of a matching event, the process engine gets the information from the CEP platform. The processes are able to send and receive messages among themselves, controlled by the process engine itself. The data-based decisions are taken based on the information generated during process execution, optionally stored in a data object. The event-based decision points such as an activity with a boundary event or an event-based gateway receive external events through the CEP platform.

5.3.2 Use of Event Abstraction

To hide the complexity behind the event hierarchy, the notion of event abstraction is implemented. Based on the subscription query, an abstraction rule is defined in the CEP platform. The platform then keeps listening to the relevant event sources. Having received each event, the abstraction rule is evaluated. If the event occurrence matches any part of the rule, the rule is partially satisfied. Once there is enough event in-

⁹ <http://xdk.bosch-connectivity.com>

¹⁰ https://www.tomtom.com/en_gb/sat-nav/tomtom-traffic/

formation to satisfy the complete rule, the output event is generated. At this point, the CEP platform checks if there is still an existing subscriber for this specific high-level event. The subscribers are notified about the occurrence of the high-level event accordingly. Only this event is modeled in the business process, therefore it is also called business level event or business event. The whole event hierarchy is represented in the event abstraction rule and thus hidden from the process model, satisfying *R2: Representation of Event Hierarchies*.

In our example usecase, if there is a delay above a threshold, a LongDelay event is produced. In addition, the location of the source of delay should be ahead of the current GPS location of the truck. In Unicorn, event abstraction rules are created accordingly for the event LongDelay. Since Unicorn has the Esper engine at its core, we used Esper Event Processing Language (EPL) [44] for writing event abstraction rules. The event types can be defined in Unicorn as shown in [Listing 8](#).

Listing 8: Event type definitions for motivating example

```
CREATE schema Disruption
(latitude double, longitude double,
reason string, delay double);

CREATE schema CurrentLocation
(latitude double, longitude double,
destLat double, destLong double);

CREATE schema LongDelay
(reason string, delay double,
destLat double, destLong double);}
```

The rule for creating LongDelay is given in [Listing 9](#). Note that the function distance() is not defined in EPL, but has been implemented additionally to find out if the disruption is ahead of the truck or not. The function takes the latitude and longitude of a certain location and the destination as input parameters. With those values, it then calculates the distance between the specified location and the destination.

Listing 9: Event abstraction pattern for LongDelay

```
INSERT INTO LongDelay
SELECT d.reason as reason, d.delay as delay,
       l.destLat as destLat, l.destLong as destLong
FROM pattern[every d=Disruption-> l=CurrentLocation
WHERE distance(d.latitude, d.longitude, destLat, destLong)
      < distance(l.latitude, l.longitude, destLat, destLong)];
```


5.3.3 Implementation Concepts

This section elaborates on how the technical challenges for the implementation are handled in the integration framework. The following discussion proposes solutions to the requirements elicited as part of *R3: Implementation of Integration*.

EVENT BINDING. In [Chapter 3](#) we mentioned events with different properties: transitional events, external events, BPMN events. Event binding is the concept of mapping these different kinds of events to each other.

To map the external events to BPMN events, catching message event constructs are modeled in processes. The catching message events can be used as start event, normal intermediate event, boundary event, and in association with an event-based gateway. The process needs to subscribe to the events to catch them during execution. We extend the process models by *event annotations* that are used as event binding points. To enable the subscription, a subscription query is added for each event at design time. Only simple queries for subscribing to the business event are added in the model. More complex event queries to produce these high-level events are generated by abstraction rules inside the event processing platform, as per *R2*. For the current use-case, the external event `LongDelay` needs to be mapped to the BPMN event `Long delay`. The annotation for this event is “SELECT * FROM `LongDelay`” which abstracts from the complexity of event queries dealt in CEP platform.

*Event subscription
query*

The other event binding point needed is to map the transitional events, i.e., lifecycle transitions, to BPMN events. Since we know the location of the destination and we receive the regular GPS updates from the truck driver, we can match the latitude and longitude for both of them. If they match, we can conclude that the truck has reached its destination. This event matching specifies the binding point according to which the engine changes the state of the activity `Drive to destination` or `Drive along updated route` from *running* to *terminated*. The state transition points can be used to monitor the status of the process, and to automatically change the state of an activity instance [16]. For example, the *begin* and *termination* states of the activities `Load goods` or `Deliver goods` can be used to track the status of the shipment. Again, the cancellation of the activity `Drive to destination` might trigger postponing the estimated time of delivery due to delay. This will be an example of mapping transitional events to external events (i.e., external to the “updating delivery time” process).

EVENT SUBSCRIPTION & CORRELATION. Once the process is modeled, it is deployed to the process engine along with the event queries. The deployed model is then parsed and ready to be executed. The event types and queries are sent to the event processing platform. They are

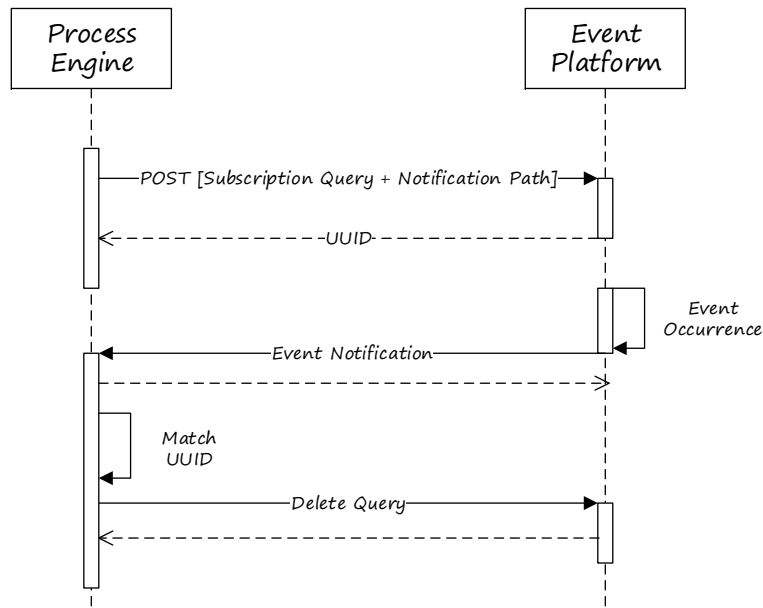


Figure 26: Event subscription and correlation within a process engine and an event platform.

registered and stored there. Once a matching event occurs, the process engine gets notified about it.

Process models serve as the blueprint for process instances [134]. Event subscription can be done for a process model or for a specific process instance [79]. For the basic integration, the start events are always subscribed to at the time of deployment. Thus, subscription for Transport plan received is done at process deployment. In our case, the truck driver might register to the mailing list of the logistics company to receive transport plans. The process gets instantiated with the occurrence of start event(s). Subscriptions for intermediate events are made once the control-flow reaches the event constructs, during instance level execution. In the example process, the annotation for the event binding point of Long delay is registered for each process instance separately once the activity Drive to destination begins.

Unsubscription from an intermediate event is done once the event is consumed or the associated activity is terminated in case of a boundary event. Accordingly, the driver stops listening to Long delay once she changes the route or reaches the destination. The start events are unsubscribed from only when the process is undeployed. Note that this is the subscription semantics as defined by BPMN. The work in this thesis further explores the complete subscription mechanism with other possible points in time when subscription and unsubscription can be done. Details about flexible (un)-subscription are found in later parts of the thesis (see [Chapter 6](#) & [Chapter 7](#)).

From a technical viewpoint, we extend the *execute*-method of the event nodes. When the process execution flow reaches this event node,

the subscription query and a notification path is sent to the event processing platform (EPP). The EPP responds with an Universally Unique Identifier (UUID) which is then stored in the process engine as a correlation key. Every time an event occurs, the EPP checks if there is an existing subscription for this event. If a matching query is found, then the notification is sent to the provided path along with the UUID. Now, upon receiving the notification, the process engine matches this UUID to the ones stored before and correlates the event to the correct process instance. Once the event is consumed, the *leave*-method of the event node performs a DELETE operation for unsubscription. The above sequence is depicted in [Figure 26](#).

EVENT CONSUMPTION. In several scenarios, reacting to an external event can only mean the occurrence of a BPMN event. The process specification according to BPMN semantics is simply followed to complete the reaction. The occurrence of a start event will always create a new instance of the process. If the Long delay event occurs in our usecase, following the semantics of BPMN interrupting boundary event, the associated Drive to destination activity will be aborted. An exception branch will be triggered additionally, which will lead to the activity Re-calculate route.

For using the event payload in further process execution, there should be provisions to store the event data. We suggest mapping the data contained in the notification to the attributes of a data object. This data object might already exist at the time the event notification is received. Otherwise it is created anew upon receiving the event. For each attribute of the data object, there exists a mapping that derives the attribute value from the event information and maps it to the outgoing data object.

There can also be a third kind of reaction to an event, such as changing a lifecycle state based on an external event information. This has already been discussed as event binding point.

5.4 SUMMARY & DISCUSSION

Business process management and complex event processing complement each other to monitor and control interacting devices, especially in an IoT related context. The work presented above assembles the necessary parts of integrating these two areas. We gather the requirements to build the framework with reference to a usecase from the logistics domain. Namely, we address the following major aspects:

- Separation of concerns between business process behavior and complex event processing techniques by enabling an event processing platform and a process engine to communicate with each other,

- Representation of event hierarchies in the event query (abstraction rule) while abstracting the complexity from the process model,
- Implementation challenges for event integration into business processes such as event binding and subscription, correlation, and consumption of events.

The conceptual framework and an integrated architecture are composed to this end, which enable the basic interaction between processes with the environment. Though the technical details are described with Gryphon, Chimera, and Unicorn; the framework is independent of any specific platform, language, or technology. The process specification follows BPMN semantics and realizes the reaction to an event according to the semantics defined in the standard. Our solution architecture handles the basic BPMN event constructs such as message or timer events, boundary events and event-based gateways. The event data can be stored in a data object to use further in decision making or execution of an activity. However, more complex event constructs like signal, error, or parallel events have not been considered in the current work and are yet to be implemented.

While exploring the scenarios where event-process communication plays a big role, we stumbled upon situations where the standard semantics for subscription management are not enough. These situations demand more flexibility with respect to the subscription lifetime to fit the needs of a distributed setup. The next chapter digs deeper into the need of flexibility for event subscription and proposes a flexible event handling model for business processes.

After an introduction to basic event handling, this chapter advances to the flexible event handling model that enables early subscription to the events. Event handling notions such as subscription, occurrence, consumption, and unsubscription are discussed individually from a process execution context. The points of subscription and unsubscription specify the possible milestones when a process can start and stop listening to an event along process execution timeline. The buffer policies instruct how to store and consume the relevant events for each execution. Finally, the semantic interdependencies among the above concepts are discussed. The work in this chapter has been partly published in “Events in Business Process Implementation: Early Subscription and Event Buffering” [81] and “A Flexible Event Handling Model for Business Process Enactment” [79].

6.1 MOTIVATION & OVERVIEW

The previous chapters establish the definition of events, event processing, how the events influence process execution, and how the process might react to an event. In this chapter, we focus specifically on the subscription management for the intermediate catching events received from the environment. Previously in [Chapter 5](#), we introduced a use-case from logistics domain. Let us consider an extended version of the transportation process, shown in [Figure 27](#), that motivates the need for flexible subscription.

The logistics company here coordinates over 200 trucks each day, and therefore, needs certain communication to be automated to make the business more efficient. In essence, the logistics company employs a process engine to control and monitor the transportations. The truck driver is contacted via email or text message, as earlier. The internal processes as well as the message exchanges between the truck driver and the logistics company are shown using a BPMN *collaboration diagram* [94]. The collaboration diagrams are used to show how two or more process participants without a central control interact with each other through message exchanges. Each participant is captured as a *pool*. In [Figure 27](#), the logistics company is represented as the pool named “Process Engine” and the “Truck Driver” is represented in a separate pool. Additionally, the process engine gets the external event information through a CEP Engine. The internal behavior of the CEP platform is hidden from the logistics company, but the message flow is visible.

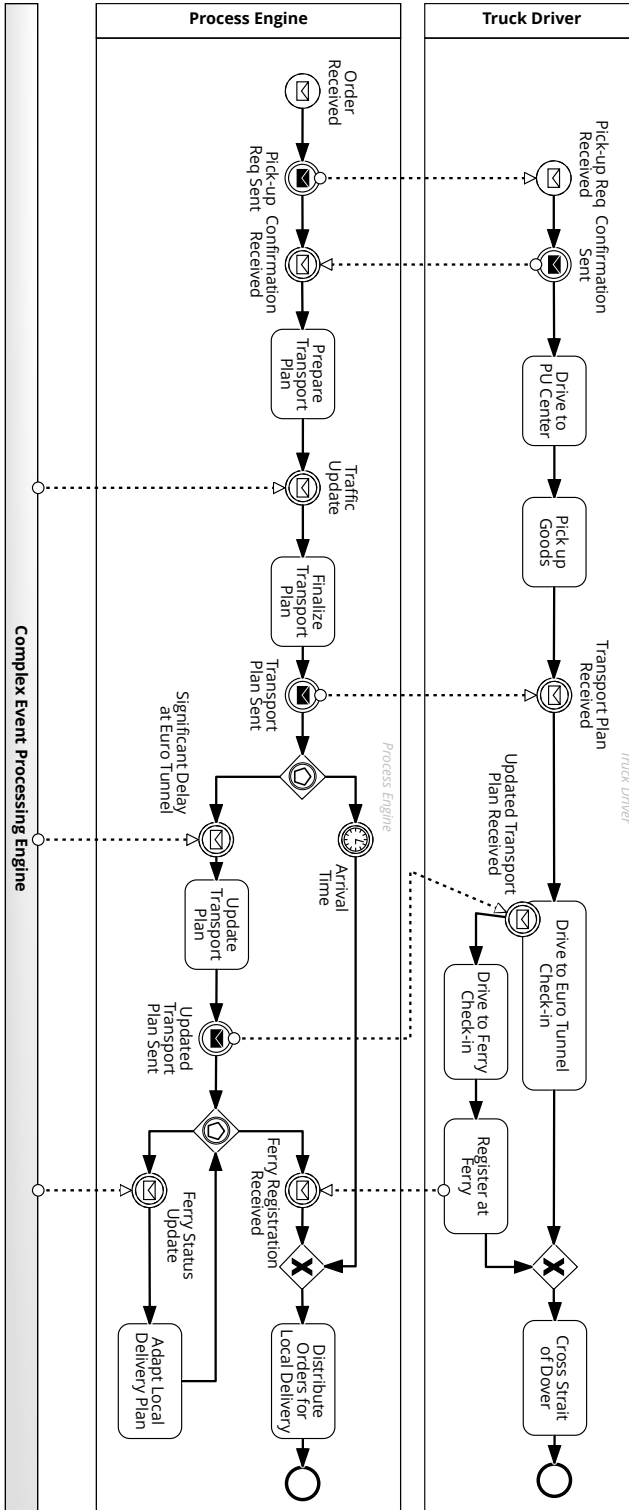


Figure 27: Collaboration diagram motivating the need for flexible subscription.

The transportation in the current example is multi-modal. The truck needs to ship goods from UK to continental Europe and for that, it needs to cross the Strait of Dover. Crossing Strait of Dover can be done using the train through Euro Tunnel, which is the preferred way since it is much faster. The other option is to use ferry, which is preferred only if there is a delay at Euro Tunnel due to accident or technical failure.

The whole collaboration starts when the logistics company receives an order to deliver at Europe. Once the truck driver confirms the pick-up request, she drives to the pick up center (PU Center) and loads goods. Meanwhile, the logistic company starts preparing the transport plan containing the specific Euro Tunnel connection to take. After the initial plan is done, the traffic update is considered to know the current situation of the route. Accordingly, the transport plan is finalized and sent to the driver. Following the plan, now the truck starts driving to the Euro Tunnel. While still driving, if there is a notification about a long delay at the Euro Tunnel, the logistic company updates the transport plan with a booking for the ferry and sends it to the driver. The truck now goes to the ferry instead of Euro Tunnel. Eventually, using Euro Tunnel or ferry, the truck crosses Strait of Dover. Once the route changes to ferry, the logistic company listens to the ferry status to update estimated time of arrival and adapts the local delivery plan for continental Europe. Finally, based on the ferry registration, the actual ETA can be calculated and accordingly, orders can be distributed among the local delivery partners.

Motivating example capturing interactions between process and environment

The CEP platform here plays the role of environment, influencing the shipment process. The process engine subscribes to the external events by registering a subscription query to the CEP platform, which in turn connects to the event sources such as the GPS sensor of the truck, public APIs for traffic flow information¹, the Euro Tunnel RSS feed², and notifications from the ferry operator. Operating on the event streams from these sources, the CEP platform then notifies the process engine about current traffic situation in the route, a significant delay at Euro Tunnel or the schedule of the ferry.

Now, let's take a closer look at the *intermediate message events* in the collaboration diagram, since these are the events used for representing communication with external world. The start event for the collaboration Order Received is sent by the customer (not shown in diagram). The exchange of pick-up request happen next which triggers the process for the truck driver. After the pick-up request is received, the confirmation is passed. In the process engine context, the event Confirmation Received can not occur before the Pick-up Req Sent event, since they are causally bound. However, the event Traffic Update can occur anytime during the preparation of transport plan or even before that. In reality, the traffic update is published at a regular interval. If the pro-

¹ <http://webtris.highwaysengland.co.uk>

² <http://www.eurotunnelfreight.com/uk/contact-us/travel-information/>

*Need for flexible
event subscription*

cess misses the last event, then it has to wait for the next occurrence. Moving on, the transport plan needs to be finalized before it is sent. But it does not depend on the truck driver's engagement in driving to the PU Center or loading goods. We can still assume that the logistic company is aware of the truck driver's internal process and communicates the transport plan in a synchronous way, i.e., when the truck driver is done with picking up goods. This assumption does not hold for the next catching event though. Events that inform about delays at the Euro Tunnel check-in are published by the environment at regular intervals and do not depend on the process execution status. On the contrary, the shipment process waits for respective events only after the transport plan has been sent. Thus, a relevant event that would have led to route diversion may have been missed. In contrast to the Euro Tunnel, events on the ferry status are not published at regular intervals, but solely upon operational changes with respect to the last notification. Clearly, as per BPMN event handling semantics, a process instance may miss the relevant event.

Let's consider a transport on a very busy weekday. A technical fault occurred in the tunnel earlier that day and the train runs 3 hours behind schedule since then. The last information on the RSS feed was published at 2:35 pm. After sending the transport plan at 2:38 pm, the process engine has started to listen to the delay event. Meanwhile, the driver starts driving towards Euro Tunnel check-in at 2:40 pm as she receives the plan already. The system publishes updated information again at 3:15 pm. Operations are still 2:30 h behind schedule, which is considered to be a significant delay. The message gets received through the process and updated plan is sent to the truck driver at 3:20 pm. The driver eventually takes the alternative route to the ferry, but only after heading to the Eurotunnel for 40 minutes. The late change of plans causes an unnecessary delay to the shipment.

The presented example illustrates the complexity of using events in business processes, especially when all possible event occurrence times are taken into consideration. It raises the need for considering environmental occurrences before the process might be ready to consume it. Differences have been pointed out as to how exactly the event is placed in the process, if it waits for a direct response to an earlier request or if the event occurrence is unrelated to the execution of that very process instance. Next in [Section 6.2](#), we explore the event handling notions focusing on the viewpoint of business process execution. However, we also show how these notions are handled in a CEP platform. BPMN event constructs with respect to their semantics are discussed, along with their possible state transitions during runtime. Based on those, in [Section 6.3](#) we address the above mentioned research questions. We propose points of subscription and unsubscription along the process execution timeline to answer the research questions *RQ1* and *RQ2*, respectively (cf. [Section 1.1](#)). Event buffering with specific buffer policies

address the questions raised by RQ_3 . The interdependencies among the proposed concepts are discussed further to gain insight about how the bits and pieces fit together. Finally, [Section 6.4](#) concludes the chapter.

The notion event buffering was conceived during Dagstuhl Seminar 2016 on “Integrating Process-Oriented and Event-Based Systems” (see 5-7, page 57, [48]), together with *Dr. Jan Sürmeli*. Extending the idea, the concepts of early subscription and event buffering are explored further in [81], published in BPM (Forum) 2017 in collaboration with *Prof. Dr. Matthias Weidlich*. The specific points of subscription and corresponding Petri net mappings (cf. [Chapter 7](#)) for the event constructs have been further conceptualized by the author and published in EDOC 2018 [79].

Acknowledgements

6.2 EVENT HANDLING NOTIONS

BPMN models use a single node to represent a catching event. At execution level, the mapping to Petri net also translates this event to a single transition. However, event handling from a business process perspective consists of multiple notions which are abstracted in this single node or transition. The four notions of event handling considered in this work are — event subscription, event occurrence, event consumption, and event unsubscription. This section explores the underlying dependencies among those notions and how they should be handled by a process engine as well as by an event processing platform.

6.2.1 Business Process View

The notions of event handling and the dependencies among them are adopted and extended from the causality proposed by Barros et al. [13], shown on the left side of [Figure 28](#). The extended version that we propose is portrayed on the right hand side of the figure. *Event subscription* is technically sending and registering the subscription query for a specific event. The process starts listening to the event once the subscription is made. *Event occurrence* can happen anytime, independent of an existing subscription. However, only after subscription, the occurrence of an event is notified to the subscriber, in our case, the process engine. Therefore, the *relevant occurrence* can take place only after a subscription has been registered.

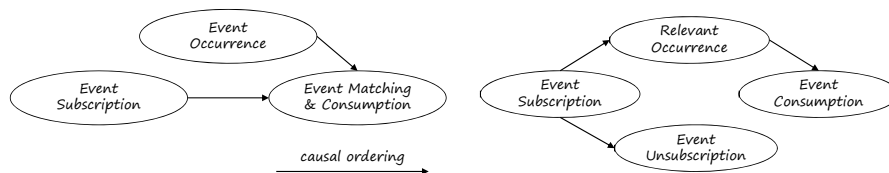


Figure 28: Dependencies among the event handling notions: event subscription, occurrence, consumption, and unsubscription.

The process engine can subscribe to an event processing platform or directly to an event source. If the subscription is directly registered at the event source, then the process engine is notified about the atomic events produced by the source. If the event streams are fed to an event processing platform, then the EPP can perform several operations on multiple event streams (cf. Section 3.4) to aggregate the raw atomic events and come up with the higher-level business event required by the process engine. The authors in [13] combine event matching and consumption in one step. However, as described in Chapter 5, the logic and control for matching an existing event subscription with the (aggregated) event occurrences and notifying the subscribers lie in the EPP. Thus, event handling notions such as event abstraction and matching are abstracted from the process engine view.

Note that event occurrence and event detection are not differentiated further, since event occurrence in this context already signifies the detection of the business level, most possibly complex event, by the event processing platform. The possible delay due to the communication channel between the event platform and the process engine is definitely possible [78], however, this is not considered in the scope of this thesis.

In the adopted version, work *event consumption* signifies the detection of the event in the process control flow and reacting to it as per process specification. This essentially means, once there is a subscription and a matching event has occurred, the event information can be used by the process for different purpose, such as performing a task, taking a decision, aborting an ongoing activity, initiating exception handling measures, and choosing an execution path to follow. *Event unsubscription* is done to stop listening to a particular event by deleting the subscription for it. Unsubscription is not mandatory, but recommended, since otherwise the process engine is fed with unnecessary events which might create overhead for the engine. Usually unsubscription is made after an event is consumed. But it can also be done before consumption, essentially anytime after an existing subscription, if somehow the event information is not relevant for the process any more. Next sections in this chapter will elaborate more on the possible points of (un)-subscription. Formally, the temporal dependencies can be expressed as $S_e < O_e < C_e \wedge S_e < U_e$ where S_e denotes the subscription of event e , O_e denotes the occurrence of e relevant for the consumer process, C_e denotes the consumption of e , and U_e denotes the unsubscription of e .

EVENT CONSTRUCTS. The categorization of event constructs according to their position in the process (start, intermediate, throwing), interaction mode (catching, throwing) and semantics (blank, timer, message) has been discussed in Section 2.2.2. In our work, external events are always represented as start or intermediate catching message event. We focus on intermediate events since they have more variances and can ac-

*Causal dependencies
among event
handling notions*

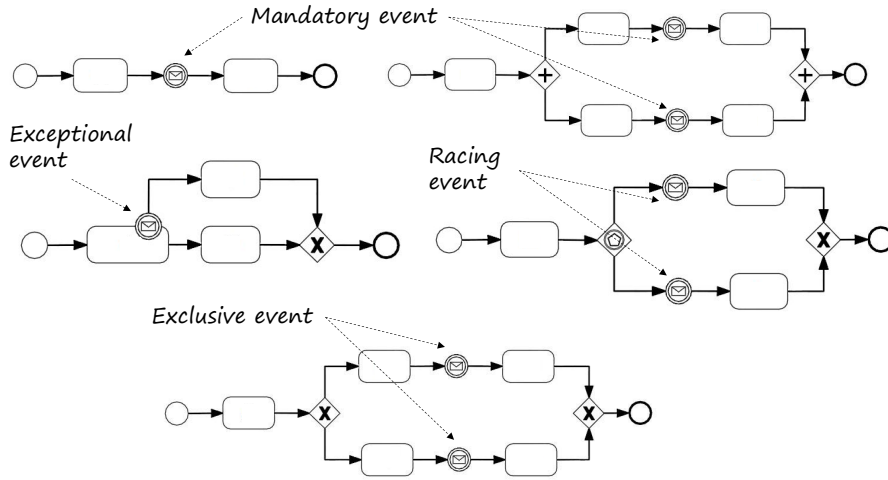


Figure 29: Event construct classification put in example process models.

commodate flexibility in terms of subscription and unsubscription. Depending on the necessary differences in event handling, we propose the design time classification of the intermediate catching message events as described below. Figure 29 visualizes the event constructs modeled in process skeletons.

Definition 6.1 (Mandatory Event).

For a process model \mathfrak{M} with the set of traces \mathcal{J} , an event $e \in E_{IC}$ is a mandatory event iff $\forall \sigma = t_1, t_2, \dots, t_n \in \mathcal{J}, \exists 1 \leq j \leq n, n \in \mathbb{N}$ such that $t_j = e$. Let $E^M \subseteq E_{IC}$ be the set of mandatory events in \mathfrak{M} . \blacklozenge

The mandatory events are those events that have to occur in order to complete the process execution. Based thereon, a mandatory event is an event in the main process flow or an event inside a parallel branch that is in the main process flow. In either way, the control flow will definitely reach the event construct at some point for all possible executions and the event will be awaited before the process execution can move further. Note that a start event is always a mandatory event since a start event needs to occur for each execution of a process.

Definition 6.2 (Exceptional Event).

An event $e \in E_{IC}$ is an exceptional event iff $e \in \bigcup \text{image}(\mathcal{B})$ where the function \mathcal{B} maps the activities to its associated boundary event(s). Therefore, $\text{image}(\mathcal{B})$ is the set of events associated with activities. Let $E^B \subseteq E_{IC}$ be the set of exceptional events in process model \mathfrak{M} . \blacklozenge

This is exactly the same as interrupting boundary event defined in BPMN. The BPMN specification says, the boundary event is always attached to an activity or a subprocess. Once the event occurs, the associated activity is canceled and an exceptional branch is triggered. The relevance of the event occurrence timestamp here is very important.

It has to be during the associated activity being in *running* state, i.e. the event must happen after the activity begins and before it terminates in order to follow the exceptional path. Since the scope of this work is only sound processes, we do not consider non-interrupting boundary events.

Definition 6.3 (Racing Event).

The events $e_1, e_2, \dots, e_n \in E_{IC}$ are racing events iff $\forall e_i, 1 < i \leq n \in \mathbb{N}, \exists g \in G_E$ such that $\bullet e_i = \{g\}$. Let $E^R \subseteq E_{IC}$ be the set of racing events in process model \mathfrak{M} . \blacklozenge

BPMN event-based gateway is a special gateway where instead of data, the decision is taken based on event occurrence. The gateway is immediately followed by several events and whichever event occurs first, the process takes the branch led by that event. This is why the events after an event-based gateway are supposedly in a race with each other.

Definition 6.4 (Exclusive Event).

An event $e \in E_{IC}$ is an exclusive event iff $\exists \sigma_1, \sigma_2$ such that $\sigma_1 = e_s \dots, g_1, n_1, n_2, \dots, n_m, g_2, \dots, e_e$ and $\sigma_2 = e_s \dots, g_1, n'_1, n'_2, \dots, n'_l, g_2, \dots, e_e$, where the start event $e_s \in E_S$, the end event $e_e \in E_E$, and the XOR gateways $g_1, g_2 \in G_X$ such that $\exists i \in [1, m] : n_i = e$ AND $\forall j \in [1, l] : n'_j \neq e$. Let $E^X \subseteq E_{IC}$ be the set of exclusive events in process model \mathfrak{M} . \blacklozenge

According to the above definition, an event e can only be in one of the paths between the XOR gateways g_1 and g_2 . For a specific process instance, only one of the paths after an exclusive gateway is followed. This makes the events on the branches after an XOR split and before an XOR join exclusive to each other, i.e. when one of the branches is chosen, the events in other branches are not required any more.

RUNTIME EVENT LIFECYCLE. The above classification of event constructs is done based on a static model with its process specification semantics. We consider the design time classification since we already specify the event handling configuration as annotations to the events in the process model. The configurations are followed by the process engine once the model is deployed. However, during runtime, the requisite of an event can vary depending on the process execution status. For example, we say the mandatory events are the ones on the regular control flow or inside a parallel branch. This is true at both design and runtime. On the contrary, if we consider the runtime situation for the exclusive events, once a particular branch is chosen after the XOR gateway, the events on that branch become mandatory for the process instance to complete execution. [Figure 30](#) shows the state transitions of event requisite from a process execution perspective.

Once a process model is deployed, the events are deployed to the engine too. In the course of process execution, the deployed events

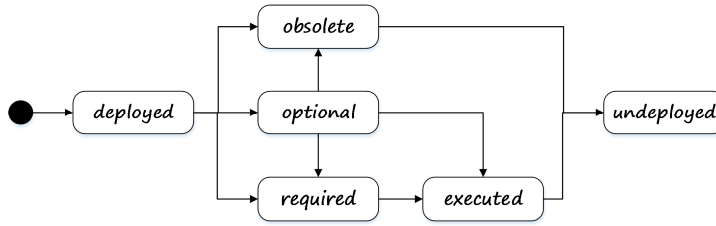


Figure 30: Lifecycle of an event w.r.t. requisite for a process execution.

can transit to either of the following three states depending on their desirability to the process: *Required* — when the event is absolutely necessary to the instance for a complete execution, *Optional* — when the event might occur, but does not have to, *Obsolete* — when the event is not necessary for the instance any more. If we try to map the event constructs described above to the runtime lifecycle, a mandatory event for a process model will be required for all instances of that model. All required events are executed eventually. An exceptional event is optional from the start of a process instance execution, such that it does not have to happen, but there is a provision to execute it if it happens. Once the associated activity is terminated, it goes to obsolete state. Similarly, the racing events are also optional and can be executed if they occur. But only one of the racing events is executed for a single instance, the rest change the state to obsolete. In case of the exclusive events, they are also optional until the branch they are situated on is chosen for the specific instance and they become required. If a different branch is chosen, they become obsolete instead. All the events are either executed or obsolete by the time the instance completes execution. Eventually, the events are undeployed along with the process model.

6.2.2 Event Processing View

The execution of event handling notions from an event processing platform’s perspective are shown using a transition system in [Figure 31](#). A formal definition of transition system is found later in this thesis, see [Definition 8.1](#) in [Section 8.2.1](#). The basic concepts behind transition system is that it abstracts from the internal behavior of a system and only represents the interactions of the system with its environment. Here, “?” signifies receiving of an event, whereas “!” represents sending of an event.

Initially, the event stream (ES) is empty and there is no registered event type (ET), subscription query (SQ), and event match (EM), as represented in state s_0 . Once the process is deployed, the event types (E1, E2) and the subscription queries (q1, q2) are registered to the EPP. As a result, in state s_1 , the ET list contains {E1, E2}, and the SQ has {q1, q2}. However, the ES and EM lists are still empty. Let’s assume the subscription query q1 listens to the occurrence of E1 where q1 is satisfied with the two consecutive occurrences of the lower-level event

$e1$. On the other hand, $q2$ is registered to get notification about $E2$ and needs the pattern $e1$ followed by $e2$. The event types and queries written in Esper EPL are given in Listing 10.

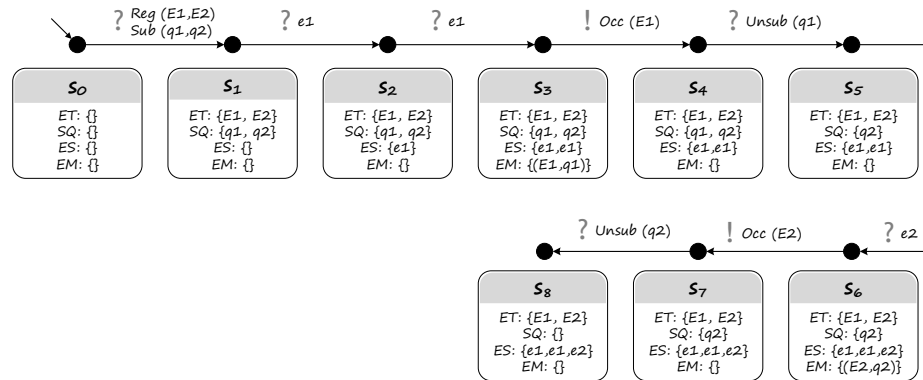


Figure 31: Interaction from event processing platform's perspective.

Listing 10: Example of event patterns

```
CREATE schema e1(id string);
CREATE schema e2(id string);

INSERT INTO E1
SELECT * FROM PATTERN[e1->e1];

INSERT INTO E2
SELECT * FROM PATTERN[e1->e2];
```

If the EPP receives an $e1$, it changes the state since the queries are partially matched now. The ES therefore has the event $e1$ stored; but the ET, SQ, and EM lists remain same for s_1 and s_2 . As soon as the next occurrence of $e1$ is received, shown as ES: $\{e1, e1\}$, the query $q1$ gets satisfied and the event $E1$ is generated. This is shown as the tuple $\{E1, q1\}$ in EM list. The EPP now notifies the process engine about the occurrence of $E1$. Assuming the process engine unsubscribes from an event query after consumption of the event, $q1$ is eventually deleted from the SQ list. At this point, if $e2$ happens, then $q2$ gets satisfied, and the notification is sent accordingly. The event stream now contains $\{e1, e1, e2\}$. Eventually, the process engine wants to stop listening to $E2$ and therefore unsubscribes from $q2$. As a result, the event processing platform ends up in state s_8 with no registered query.

6.3 FLEXIBLE SUBSCRIPTION MANAGEMENT

Process engines are responsible for performing, monitoring, and controlling business process execution. Either an integrated modeler or

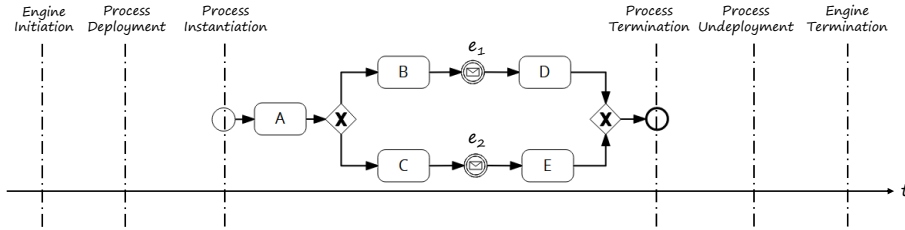


Figure 32: Process execution timeline.

a separate editor can be used to model the processes. At some point after the engine is initiated, the process model is deployed to the engine. BPMN processes are instantiated by start event(s). Every time the start event occurs, the process model has a new running instance. The process instances are executed according to the process definition. Different instances can have different execution traces based on the decisions taken or events occurred during process execution. Each instance is terminated with an end event. No running instance of a process model exist once the process model is undeployed. Once there is no running instance from any process model, the engine can be terminated. The milestones of process execution starting from the engine initiation until the engine termination is shown in [Figure 32](#).

6.3.1 Points of Subscription

If the occurrence of an event is not dependent on the process execution, then it can happen anytime along the timeline and even beyond that. However, we might need certain information to subscribe to an event. These information can be specific to a process instance, a process model or across processes. As soon as there is no unresolved data dependency, the process can start listening to an event it might need in future. The *Points of Subscription (POS)* listed below control at which point in time a certain event can be subscribed to. Since the semantics of subscription point might be different for different event constructs, a formal clarification is given for each POS.

POS1: AT EVENT ENABLEMENT. A subscription is made only when the event construct is enabled by the control-flow. This is completely consistent with BPMN semantics and should be implemented when subscription for an event can be done only after completing the previous activity. In the motivating example, subscription should be done at event enablement for the event Confirmation Received, since it can occur only after pick-up request is sent. Subscription at event enablement means the following for the event constructs described before:

- Given $e \in E_{IC} \setminus E^B$ and $x \in N_A \cup N_E$ such that $x \bullet = \{e\}$, subscribe to e when x terminates.

- Given $e \in E^B$ and $A \in N_A$ such that $A \rightarrow e$,
subscribe to e when A begins (A_b).

POS2: AT PROCESS INSTANTIATION. A subscription is made as soon as the process is instantiated, i.e., given $e \in E_{IC}$, subscribe to e when the start event $e_s \in E_S$ occurs.

This is required when the subscription is dependent on instance specific data, but the event can occur earlier than scheduled to be consumed in the process. For example, the transport plan is specific to each transport, therefore the truck driver cannot expect that before getting the pick-up request. However, once the truck gets confirmation and picks up the goods from the pick-up center, the transport plan can be ready anytime depending on how fast the logistics company works. The truck driver can listen to the event *Transport Plan Received* right after the instantiation of the process. In this case, the driver does that intuitively, such as being reachable via email or phone all the time and checking for the transport plan once done with loading goods.

POS3: AT PROCESS DEPLOYMENT. According to POS3, given $e \in E_{IC}$, subscribe to e at process deployment (*PD*).

This is always done for the start events, since they are needed for instantiating the process. Thus, *Order Received* will be subscribed at the time the transport process is deployment. Additionally, the subscription for the intermediate catching event, too, is created as soon as the process model is deployed if this POS is chosen. This should be implemented when all instances of a process might need the intermediate event information. The *Traffic Update* and *Significant Delay* at Euro Tunnel can be subscribed to at process deployment, since all the trucks following the same route will need to know the updates. These updates are not published to signify a big change, rather, they are updated in a regular interval. Therefore, the last update might already be useful for the process instead of waiting for the next one.

POS4: AT ENGINE INITIATION. A subscription is made at the time when the engine starts running, i.e., given $e \in E_{IC}$, subscribe to e at engine initiation (*EI*).

This is helpful in a situation where the engine already knows which events might be needed by the processes to be executed and subscribes to the events beforehand. In such scenarios, an event information is often shared by several processes. When one process starts executing, it can then immediately access the event information already stored by the engine. It is very probable for the logistic company in our example to have other processes running than the transportation process shown here. For example, they can own other transport vehicles that are not crossing the Euro Tunnel, but still driving on the same route. For all of those transports, they also need to monitor the traffic situation and sug-

gest detour in case of a congestion. In this context, subscribing to the event *Traffic Update* is preferred to be done at engine initiation and kept accessible for all the processes that might need the information.

6.3.2 Points of Unsubscription

Similar to the point when a process starts listening to an event, we also need to decide till which point the process keeps listening to it. In case the event occurs and the process consumes it, the control flow moves on to next nodes. There can also be scenarios when an event has not yet occurred, but becomes irrelevant for the process execution. Depending on different situations when a process might and should stop listening to an event, we define the following points of unsubscription.

POU1: AT EVENT CONSUMPTION. Unsubscription is done after the event is consumed by the process and the control flow has moved on to the next node, i.e., given $e \in E_{IC}$ and $y \in N_A \cup N_E$ such that $e \bullet = \{y\}$, unsubscribe from e when y begins. In the motivating example, once the confirmation from the truck is received, the logistic company does not wait for further information from the driver.

POU2: AT SEMANTIC RESOLUTION. Unsubscription is done as soon as the event becomes irrelevant for the process. This can happen in the following three situations:

- When the associated activity for a boundary event terminates, the boundary event is no longer relevant. This is formally defined as: given $e \in E^B$ and $A \in N_A$ such that $A \rightarrow e$, unsubscribe from e when A terminates (A_t).
- When a specific branch is chosen after an XOR gateway, the events in other branches are no longer relevant. For an exclusive event $e_i \in E^X$ between an XOR split gateway $g_1 \in G_X$ and an XOR join gateway $g_2 \in G_X$, where the branches after the XOR gateway start with the nodes $\{d_1, d_2, \dots, d_n\}$, i.e., $g \bullet = \{d_1, d_2, \dots, d_n\}$, unsubscribe from e_i when d_j begins, given that $i \neq j$.
- When an event has occurred after an event-based gateway, all the events following the gateway are no longer relevant. For a racing event $e_i \in E^R$ after an event-based gateway $g \in G_E$, where the events after the gateway are $\{e_1, e_2, \dots, e_n\}$, i.e., $g \bullet = \{e_1, e_2, \dots, e_n\}$, unsubscribe from $\{e_1, e_2, \dots, e_n\}$ when e_i occurs.

POU3: AT PROCESS UNDEPLOYMENT. Unsubscription is done when the process is undeployed and there is no running instance for that process. According to this POU, given $e \in E_{IC}$, unsubscribe from e at process undeployment (*PU*). The event *Significant Delay at Euro Tunnel* can be unsubscribed for all the instances once the process is undeployed.

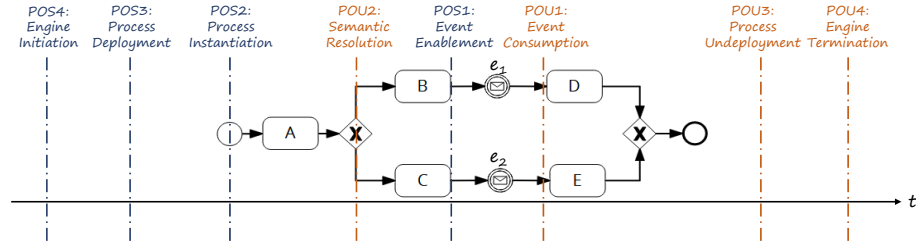


Figure 33: Points of (Un)-Subscription.

POU4: AT ENGINE TERMINATION. Unsubscription is done when the process engine is terminated and there is no running process any more. Therefore, given $e \in E_{IC}$, unsubscribe from e at engine termination (ET). The Traffic Update event can be unsubscribed at engine termination since all the processes that might need it will be terminated anyways. Figure 33 combines the points of subscription and unsubscription along the process execution timeline.

6.3.3 Event Buffering

As soon as the subscription is issued, the event occurrence is awaited. Once the matching event has occurred, the process engine gets notification about it. However, the process instance might not be ready to consume it yet, according to the motivating examples. To this end, an event buffer is proposed that will store the event temporarily. Later when the process instance is ready to consume the event information, it checks if a matching event already exists in the buffer. If the event occurrence has not happened yet, then the process instance waits for it to occur. If there exists a respective event then the process instance retrieves it. The event information is then available within the process instance context, i.e. the event can be consumed then.

However, to retrieve an event from a buffer, the occurrence cardinality of the events need to be accounted. There can be events that occur only once, such as receiving a pick-up request. In contrast, there can be events that occur continuously, forming an event stream. The traffic update or Euro Tunnel update are examples of such periodical events. When the process wants to consume an event and finds multiple events of same event type, it needs to decide which event shall be retrieved for the specific instance. To address this issue, buffer policies are configured when subscribing to an event source. The policies described below define how many events are stored in the buffer, which event is consumed by a process instance, and whether an event information is reused.

LIFESPAN POLICY. The lifespan policy specifies the subset of events received from CEP platform that should be stored in the buffer. Essentially, it defines the size of the buffer. It can also be interpreted as the

interval when the buffer is updated. Lifespan policy can be configured as:

- *Specific Length.* A specific number of events can be selected to be stored in the buffer. For example, only last 5 traffic updates are stored to analyze the current situation in a particular route.
- *Specific Time.* The subset of events can be selected using a temporal window. An example can be to consider only those events that occurred within the last 30 minutes after an accident, since the earlier events might include longer delay that is not relevant any more.
- *Keep All.* This configuration does not impose any restriction on event lifespan. All events received after a subscription is issued are thereby stored in the buffer.

RETRIEVAL POLICY. This policy specifies the event to be consumed by a specific process instance. The configurations include:

- *Last-In-First-Out.* This is suitable for situations when the latest event overwrites the required information carried by preceding events. The GPS data telling the location of a moving vehicle is an example when this configuration should be chosen.
- *First-In-First-Out.* On the other hand, there are situations when the first event is the most important among all. In a bidding context, the first vendor to bid a price below a certain threshold can be chosen as the one getting the contract.
- *Attribute-based.* Unlike the timestamp of the event in above two cases, the attribute values of the events can also be the determining factor for retrieval. In the bidding example, the contract might be assigned to the vendor who quotes the cheapest offer.

CONSUMPTION POLICY. Since an event can be interesting for more than one instance, or even for multiple processes, the information carried by an event can be considered to be reused. Consumption policy determines whether an event is removed from the buffer after it has been retrieved by a process instance. Such consumption policies are well-known to influence the semantics of event processing, see [46]. We consider the following configurations regarding event consumption for our buffer model:

- *Consume.* If the consumption policy is set to this then an event data is deleted from the buffer as soon as it is retrieved by a process instance. For example, a cancellation of order is relevant only for that specific order handling process instance.

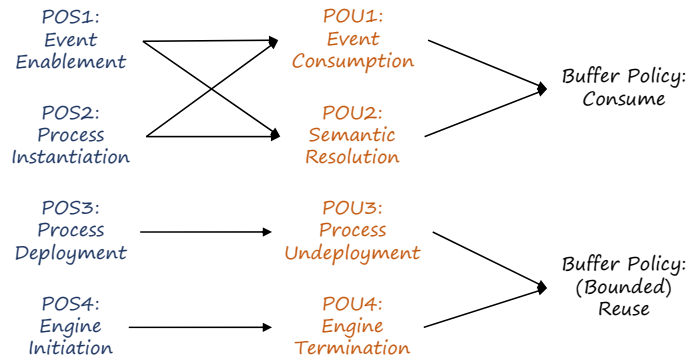


Figure 34: Interdependencies among the aspects of flexible event handling.

- *Reuse*. On the contrary, choosing this consumption policy ensures that the event data is reused across instances/processes. A strike in an airport most likely affect many processes that deal with flight delays and shuttle services.
- *Bounded Reuse*. In this case, a restriction is added to specify the number of times an event can be reused. Going back to the bidding example, offer from a certain vendor might be limited to be considered for a specific number of contracts, to avoid bias.

6.3.4 Semantic Interdependencies

The above sections describe the spectra of the event handling model. The aspects are technically independent of each other. However, from a semantic perspective, there are a few interdependencies that should be followed while using them together. For example, choosing a point of subscription for an intermediate event might have the constraint to select a specific point of unsubscription for that event. [Figure 34](#) visualizes these dependencies.

Subscription at *event enablement* and *process instantiation* are usually relevant for a specific process instance. In other words, these event information are not used after the instance have consumed it. Therefore, for both of them the unsubscription can be done at *event consumption*. In case the event information is obsolete even before consumption, then unsubscription can be done at the point of *semantic resolution*. The lifespan and retrieval policies are not dependent on the point of (un)-subscription anyway. But the consumption policy should be set to *consume* since no reuse is necessary.

On the other hand, subscription at *process deployment* is done so that the event can be accessed by all instances of that process. Even if a specific process instance does not need to listen to the event anymore after consumption or semantic resolution, unsubscription to the event will contradict the idea of reuse here. Therefore, unsubscription at *process undeployment* is recommended for this scenario. For the same reason,

the consumption policy for the buffer should be set to *reuse* or *bounded reuse*. Similarly, for subscription at *engine initiation*, the unsubscription should be done only at *engine termination*. The consumption policy should again be either *reuse* or *bounded reuse*.

6.4 SUMMARY & DISCUSSION

Languages such as BPMN, UML Activity diagrams, or WS-BPEL, do not offer flexible means for event handling. Specifically, the questions like when to subscribe to an event source, how long to keep the subscription, and how to retrieve an event for a process instance are severely ignored. Though the existing semantics for event subscription is adequate for a vast range of message exchanges, they fail to capture the flexibility required for processes communicating with external event sources in a distributed environment. The BPMN assumption of *an event occurrence only after the event construct is enabled* restricts the communication possibilities between event producers and consumers where the separate entities are not necessarily informed about each others internal status. This can lead to missing out on an event which has occurred but still relevant. Besides, waiting for an already occurred event can cause process delay, even deadlock. The need for advanced event handling has further been motivated with a shipment scenario from the domain of logistics.

To address these shortcomings, a flexible event handling model is proposed with points of subscription and unsubscription along the process execution timeline. The contributions presented in this chapter can be summarized as following:

- Investigating the role and usage of handling external events in business processes,
- Detecting limitations of common process specification languages when expressing complex event handling semantics,
- Proposing flexible subscription management system with specific points of (un)-subscription,
- Designing an event buffer to enable and manage early subscription efficiently.

The proposed concepts are formally defined in next chapter.

This chapter turns to the formal grounding of the event handling concepts discussed in Chapter 6. The generic notions of event subscription, occurrence, consumption, and unsubscription are detailed using Petri nets. Mapping steps from BPMN event constructs to Petri nets are given for each point of (un)-subscription. Further, buffer policies are defined using Coloured Petri Nets (CPN). The rich formalism provides unambiguous semantics and detailed guidelines for implementing flexible event handling configurations. “Events in Business Process Implementation: Early Subscription and Event Buffering” [81] and “A Flexible Event Handling Model for Business Process Enactment” [79] contain part of the formalism presented in this chapter.

7.1 MOTIVATION & OVERVIEW

The BPM lifecycle (see Section 2.1) instructs to come up with a fine-grained process model enriched with technical details to make it ready for execution. For using the event handling and CEP concepts for any technical solution, a formal model is recommended to guide correct implementation [93]. This chapter, therefore, gives a strong formal grounding to the concepts of event handling presented earlier, and brings them closer to implementation level.

We chose Petri nets for our formalism since this is a popular, well-established notation to model processes, and it gives more technical yet unambiguous semantics which is required for correct process execution [73, 122]. Being a formal language, there exist efficient analysis techniques and tools¹ for static semantic verification of Petri nets [137], which is not the case for BPMN models. Moreover, Petri nets are particularly suitable for capturing the communication between a process and its environment, as they support the composition of models, following the principles of loose coupling, and assuming asynchronous communication between the components.

A process model represented using BPMN or WS-BPEL can be easily and extensively transformed to a Petri net [37, 72]. Therefore, the transition of a process model from organizational level to operational level is not an additional challenge [134]. However, since the standard BPMN models do not include the flexible event handling notions introduced in this work, we define additional Petri net transformations necessary to implement the advanced event handling semantics.

¹ LoLA: A Low Level Petri Net Analyzer. <http://service-technology.org/lola/>

ASSUMPTIONS. Note that we use Petri net semantics in our work to capture the behavior of a process engine with respect to executing processes. We show the milestones such as engine initiation (*EI*) and process deployment (*PD*) (represented in Figure 32) as Petri net transitions. Therefore, the initial marking of the Petri net has one token in the input place of *EI*. In other words, we use BPMN process model excerpts along with the engine execution milestones as our representation level, and specify the corresponding Petri nets as the implementation level.

We consider only sound processes that get instantiated by a start event $e_s \in E_S$ and terminates with an end event $e_e \in E_E$. For each intermediate catching event, the temporal order $S_e < O_e < C_e \wedge S_e < U_e$ always holds. Further, we do not consider any loop structure in the process flow.

For simplicity, we focus on the semantics being discussed for the specific parts and abstract from the rest of the details. For example, we do not show the activity life cycle phases as separate transitions unless they are needed explicitly (e.g., for boundary events). As our focus is on intermediate catching events, we show only the occurrence for start and end events. In general, the subscription for start event has to happen at process deployment, since it is needed for process instantiation. Since the end event is a throwing event produced by the process, no subscription is needed for it. Note that we translate only the intermediate catching events, not the *receive tasks*.

We consider the interplay of the event handling notions only from business process view while mapping the points of (un)-subscription. The event processing aspects of event handling are discussed in the context of event buffering. Additionally, we assume that for all events, the timestamp when the event source produces the event (occurrence time) coincides with the timestamp when the event notification is received by the event processing platform (detection time).

The firing of the mapped Petri nets follow the standard semantics, i.e., once there are tokens at all the input places of a transition it can fire immediately, but does not have to. The reason behind deciding on this firing behavior over immediate firing is that it supports the activities to have certain duration which are needed to express the semantics of boundary events. Also, it will be impossible to decide on the branch to follow after an event-based gateway since the race among the events cannot be expressed using an immediate firing. We could use Generalized Stochastic Petri Nets (GSPN) [28] that allow to have both *timed* and *immediate* transitions. However, for the timed transitions in a GSPN, a random, exponentially distributed delay is assigned which is not desired in our scenario.

To avoid the unwanted delay after a transition is enabled, we assume that the transitions which can be solely carried out by the engine fires immediately. Essentially, the assumption is that the engine makes sure if there is no other dependency to execute a transition, it is executed im-

mediately upon enablement. The transitions such as issuing a subscription or an unsubscription, consuming an event when the control flow reaches the event construct and there is an existing event to consume, and cancelling an activity following a boundary event consumption fall under this category. On the contrary, the transitions where the engine needs to wait on the environment such as occurrence of an external event and executing a user task might take time to be fired even if they are enabled.

7.2 PETRI NET MAPPING

There are multiple layers of subscription and unsubscription in the complete event handling scenario, such as between the process engine and the event platform, and between the event platform and the actual event sources in the environment. They are unfolded step wise in the following. First, the event handling notions are transformed to Petri net transitions in [Section 7.2.1](#). In [Section 7.2.2](#), each point of subscription (POS) is mapped to corresponding Petri net modules, depending on the event construct configured with the POS; followed by the mapping of each point of unsubscription in [Section 7.2.3](#). Next, we switch to coloured Petri nets, since formalizing the buffer policies demand advanced expressiveness. [Section 7.2.4](#) discusses the event buffer along with the policies expressed by functions used in the arc inscriptions. [Section 7.3](#) summarizes and concludes the chapter.

7.2.1 Event Handling Notions

The Petri nets representing event handling notions presented in [Section 6.2](#) are given in [Figure 35](#). The event source is captured only by its interface to the process flow. Here, event source can be interpreted as either an individual event producer or the event processing platform.

We start with the description of the process execution level. The step of subscribing to an event e is captured by a dedicated transition S_e , which is triggered when the point of subscription is reached (place $P(POS, S_e)$). This transition produces a token in the place *sub* that passes the token to the event source. The process execution moves on to the next node, represented by the place $P(S_e, a)$ where a is the node following the point of subscription.

As soon as the matching event occurs, the event source sends it to the process engine, which is represented by having a token in the place e . Note that this place represents the event notification that is available only after a subscription has been made. Even if there are event occurrences before subscription, they are not yet relevant for the process execution and therefore, will not produce any token in this place.

The control flow enablement of the BPMN event construct in the process execution level is mapped to the consumption of the event, represented by the transition C_e . The predecessor and successor of C_e are x

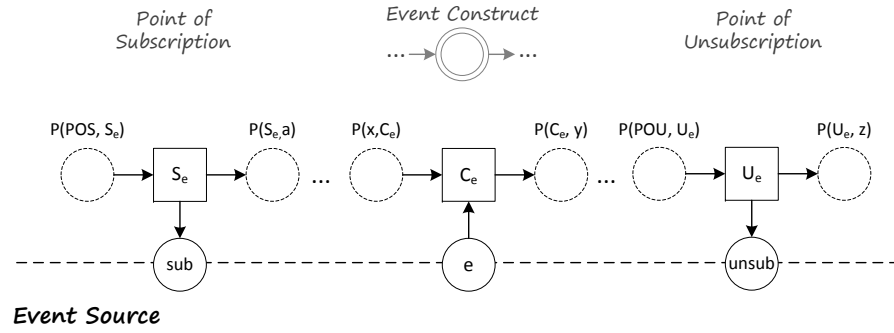
Process Execution

Figure 35: Event handling notions represented by Petri net.

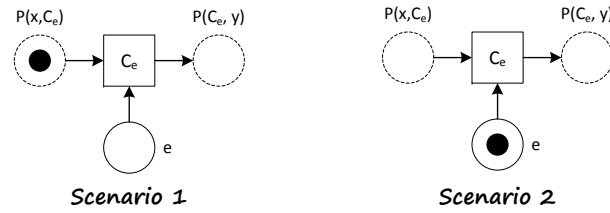


Figure 36: Scenarios considering the separate notions of event occurrence and event consumption.

and y , respectively. Since the event handling model considers the event occurrence and consumption with separate transitions independent of each other, there can be the following two scenarios:

- The control flow reaches the event construct, but there is no matching event that has occurred yet. This situation is mapped to a Petri net with a token in the place $P(x, C_e)$ and no token in the place e , as shown by *Scenario 1* in Figure 36.
- The matching event has already occurred, but the process control flow has not reached the event construct yet. This is represented by *Scenario 2* in Figure 36 as the Petri net having a token in the place e and no token in the place $P(x, C_e)$.

As soon as there are tokens in both the places $P(x, C_e)$ and e , the transition C_e is fired, producing a token in the place $P(C_e, y)$, thus passing the control flow to the next node in the process.

Down the control flow, another transition U_e is enabled to unsubscribe from the event source. This represents reaching the point of unsubscription along the process execution. Similar to POS, this transition produces a token in the place *unsub* that forwards the unsubscription request to the event source.

7.2.2 Points of Subscription

In this section, we tailor the event handling notions for each event construct and apply the points of subscription. We map each pair of event

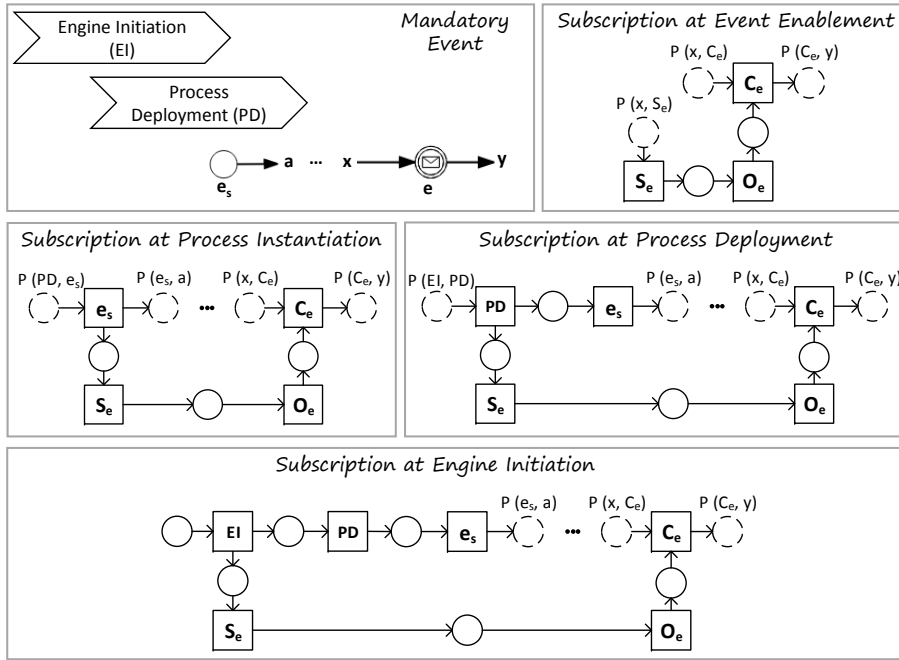


Figure 37: Petri net modules for mandatory event construct.

construct and point of subscription to corresponding Petri net modules. We take the BPMN to Petri net mapping prescribed by [37] (see [Section 2.4](#)) as basis and extend it to include subscription, occurrence, and consumption of the events. As explained by Dijkman et al., x denotes the predecessor node of event e , y denotes the successor node of e , and places with dashed borders mean they are not unique to one module. If additional process nodes are needed to be represented, they are taken from the set $\{a, b, z\}$. The gateways are represented by g . The nodes on i -th branch following a gateway are numbered as e_i, x_i, y_i , where $i \in \mathbb{N}_0$. Again, x_i and y_i denote the predecessor and successor node of event e_i , resp.

MAPPING MANDATORY EVENT. Following *subscription at event enablement*, these are the steps to map a mandatory event construct to the corresponding Petri net module:

1. Event e is mapped to three separate transitions: S_e — subscription to e , O_e — relevance occurrence of e detected by the process engine, and C_e — consumption of e .
2. S_e has one input place to link with x , the predecessor node of e .
3. C_e has one input place to link with x .
4. C_e has one output place to link with y , the successor node of e .
5. A flow is added from S_e to O_e .
6. A flow is added from O_e to C_e .

For other points of subscription, *Step2* is replaced as indicated in the following. The part of process structure containing mandatory event construct and resulting Petri nets are shown in [Figure 37](#).

- *Subscription at process instantiation*: A flow is added from the transition for start event e_s to S_e , where e_s has an output place to link it with a , the first node after process instantiation.
- *Subscription at process deployment*: A flow is added from the transition for process deployment (PD) to S_e .
- *Subscription at engine initiation*: A flow is added from the transition for engine initiation (EI) to S_e .

MAPPING BOUNDARY EVENT. For a boundary interrupting event, even if the subscription is created earlier, the event occurrence is relevant only during the running phase of the associated activity. Therefore, for this event construct, subscription is recommended to be registered only at event enablement. [Figure 38](#) shows the boundary event construct and the resulting Petri net module. The steps for mapping boundary event construct with subscription at event enablement are given below.

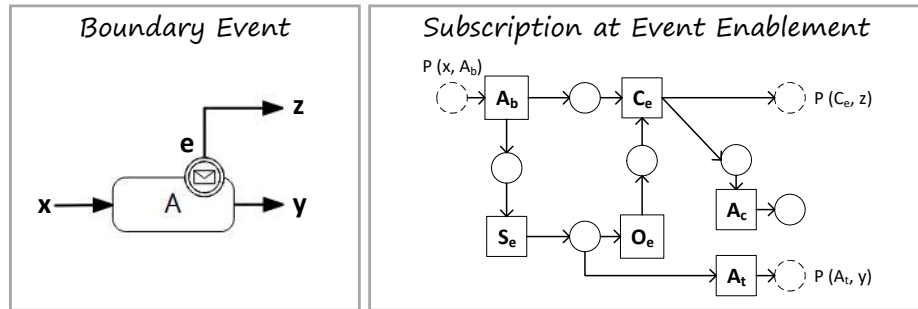


Figure 38: Petri net module for boundary event construct.

- The event e is mapped to transitions S_e , O_e , and C_e .
- The associated activity A is mapped to three transitions: A_b depicting the beginning of A , A_t depicting the termination of A , and A_c depicting the cancellation of A .
- A_b has one input place to link with x , the predecessor node of A .
- A_t has one output place to link with y , the successor node of A (normal flow).
- C_e has one output place to link with z , the successor node of e (exception branch).
- C_e has another output place to link it with A_c .
- A flow is added from A_b to S_e .
- Another flow is added from A_b to C_e .
- A flow is added from S_e to O_e .
- A flow is added from O_e to C_e .
- The input place before O_e is shared with A_t .

Here, the subscription is done as soon as the associated activity begins (A_b). After the subscription is done, a token is put in the place shared by the transitions A_t and O_e . If the event occurs before the activity terminates, i.e., the transition O_e is fired before the transition A_t , it consumes the token from the shared place and enables C_e . Having consumed the event by executing C_e , tokens are produced at the two output places – to pass on the control flow to the next node in the exceptional branch ($P(C_e, z)$) and to abort the ongoing activity (input place for A_c). Otherwise, the token from the shared place is consumed by the transition A_t and passed on to the normal flow of the process ($P(A_t, y)$).

MAPPING RACING EVENT. For racing events, subscription at event enablement means when the control-flow reaches the gateway, all the events following the gateway are subscribed. Once one of the events occur, the process flow takes the branch following that event. Note that in case of early subscription (POS_{2,3,4}), the process engine needs to check the timestamps of the events if there are more than one event available in order to consume the one that happened first and follow the path leading by it. The event-based gateway g with two racing events e_1 and e_2 is transformed to the corresponding Petri net modules in [Figure 39](#). The methodology for mapping a racing event construct to Petri net following *subscription at event enablement* is as following:

- The event e_i is mapped to two separate transitions, occurrence of e_i (O_{e_i}), and consumption of e_i (C_{e_i}).
- A single transition is introduced to represent the combined subscription to all the racing events e_1, e_2, \dots, e_n (S_{e_1, e_2, \dots, e_n}).
- The subscription transition has one input place to link with g , the event based gateway.
- Each consumption transition C_{e_i} has one output place to link with y_i , the successor node of e_i .
- A flow is added from subscription to each occurrence transition O_{e_i} .
- A flow is added from O_{e_i} to C_{e_i} .

For other points of subscription, each consumption transition C_{e_i} is connected with an input place $P(g, C_{e_i})$ to link it to g , the event based gateway. In contrast, the subscription transition S_{e_1, e_2, \dots, e_n} is not linked with the gateway anymore, rather *Step3* is replaced as indicated in the following. The racing event constructs following an event-based gateway and the resulting Petri nets are shown in [Figure 39](#).

- *Subscription at process instantiation:* A flow is added from the transition for start event e_s to the subscription transition.
- *Subscription at process deployment:* A flow is added from the transition for process deployment (*PD*) to the subscription transition.

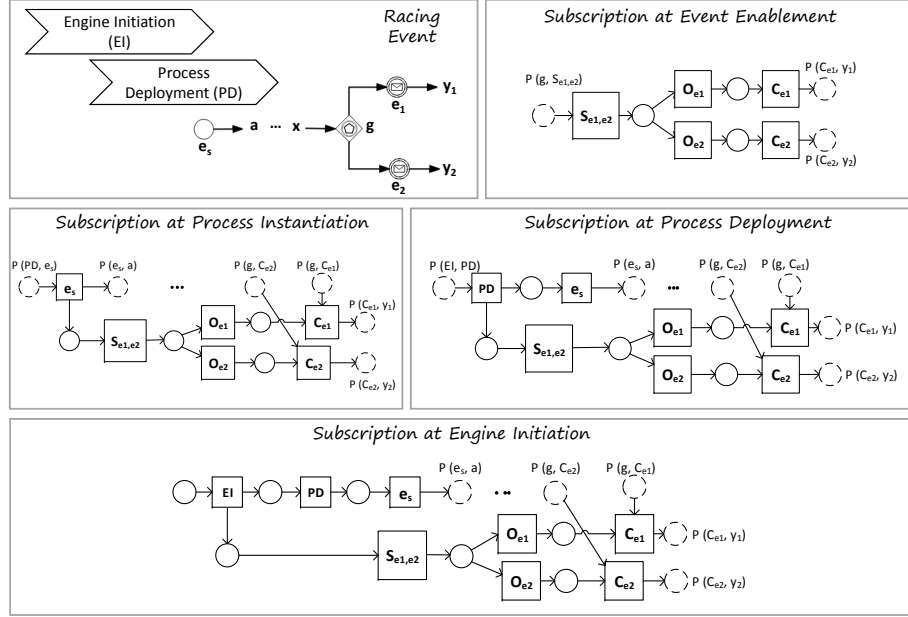


Figure 39: Petri net modules for racing event construct.

- *Subscription at engine initiation*: A flow is added from the transition for engine initiation (EI) to the subscription transition.

MAPPING EXCLUSIVE EVENT. As discussed in Section 6.2, an exclusive event is *optional* when a process is instantiated. However, it becomes *required* once the specific branch, on which the event is situated, is chosen during execution (see Figure 30). In essence, an exclusive event behaves like a mandatory event once the associated branch is enabled by control-flow. Therefore, the Petri net modules mapping exclusive event construct coincides with the Petri net modules for mandatory event construct for the same POS (see Figure 37). Nevertheless, we show the Petri net modules mapping an exclusive e_i situated on the i -th branch after the exclusive gateway g in Figure 40. The steps for mapping the exclusive event with *subscription at event enablement* are given below.

1. Event e_i is mapped to transitions S_{e_i} , O_{e_i} , and C_{e_i} .
2. S_{e_i} has one input place to link with x_i , the predecessor node of the event e_i .
3. C_{e_i} has one input place to link with x_i .
4. C_{e_i} has one output place to link it to y_i , the successor node of e_i .
5. A flow is added from S_{e_i} to O_{e_i} .
6. A flow is added from O_{e_i} to C_{e_i} .

For the remaining points of subscription, *Step2* is replaced as follows.

- *Subscription at process instantiation*: A flow is added from the transition for start event e_s to S_{e_i} .

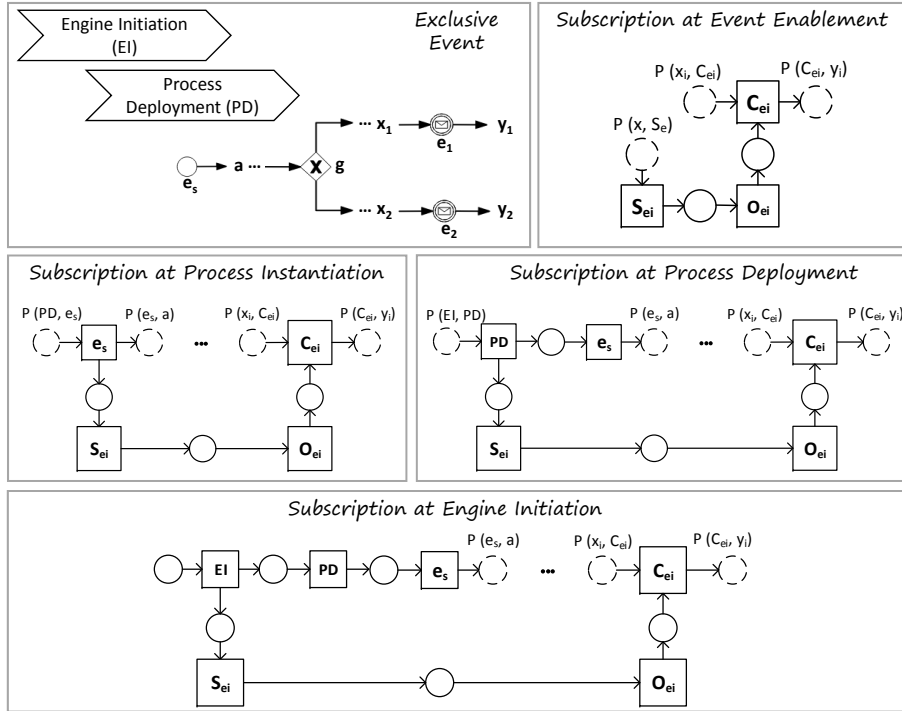


Figure 40: Petri net modules for exclusive event construct.

- *Subscription at process deployment*: A flow is added from the transition for process deployment (PD) to S_{ei} .
- *Subscription at engine initiation*: A flow is added from the transition for engine initiation (EI) to S_{ei} .

7.2.3 Points of Unsubscription

This section describes the Petri net mapping for points of unsubscription (POU), as described in Section 6.3.2. The unsubscription of event e is represented as transition U_e . Since unsubscription at event consumption (POU₁), at process undeployment (POU₃), and at engine termination (POU₄) are independent of the semantics of event construct, we describe a universal mapping for them. For unsubscription at semantic resolution (POU₂), separate mappings are given for boundary event, exclusive event, and racing event; since these are the only three event constructs where it is applicable. Figure 41 summarizes the Petri net modules for all points of unsubscription.

- *Unsubscription at event consumption*: The transition C_e , depicting the consumption of the event e , is connected to the transition U_e by an outgoing flow.
- *Unsubscription at process undeployment*: The transition PU , signifying process undeployment, has an outgoing flow to U_e .
- *Unsubscription at engine termination*: The transition ET , signifying engine termination, has an outgoing flow to U_e .

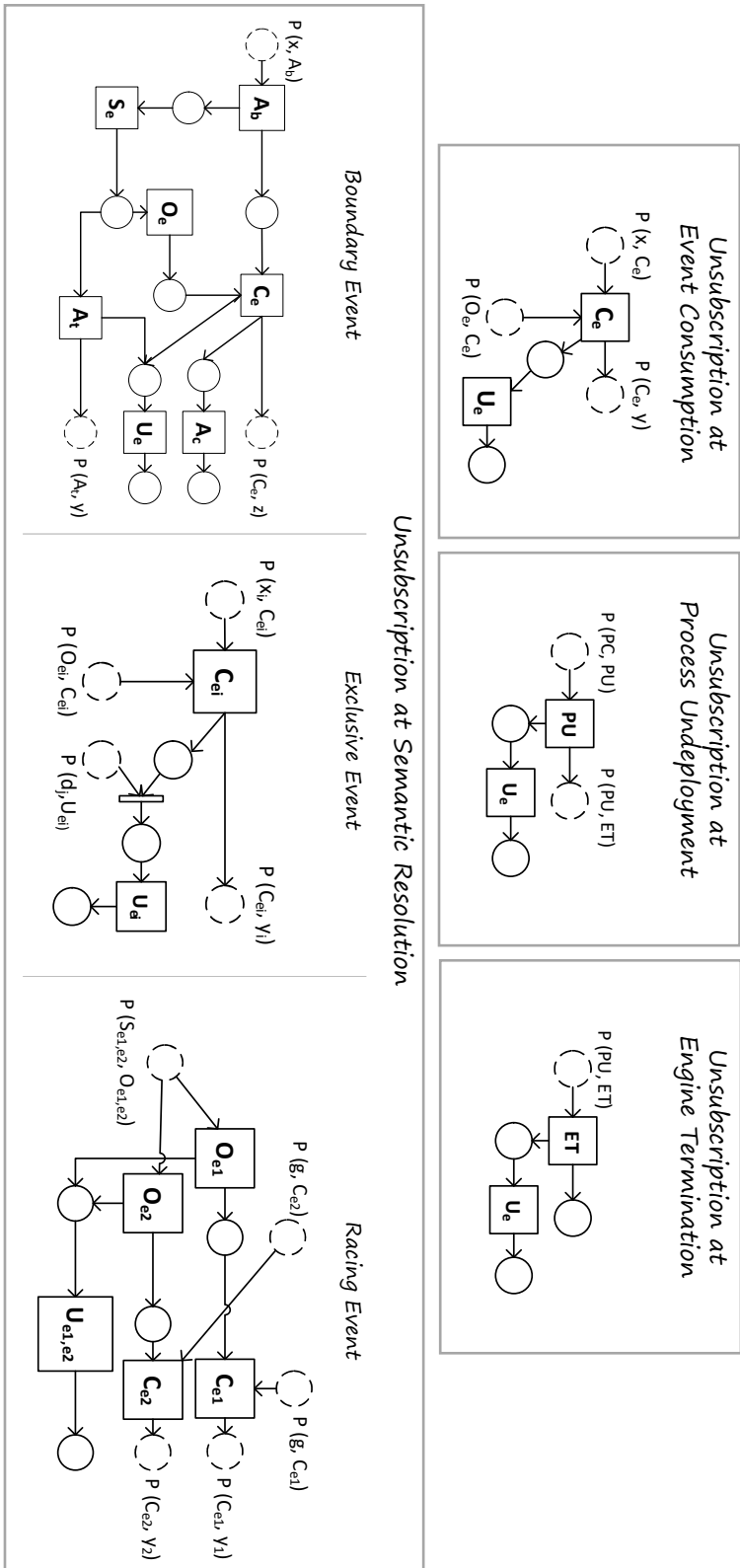


Figure 41: Petri net modules for points of unsubscription.

Semantic resolution in the context of unsubscription means the point in time when it is clear that certain event(s) is not required for the process execution any more. This signifies the event becoming an *obsolete* one (see Section 6.2, Figure 30). Mandatory event constructs are required for each process instance, hence they are named ‘mandatory’. But the other event constructs might become obsolete at specific points depending on their semantics, as described next.

SEMANTIC RESOLUTION: BOUNDARY EVENT. According to BPMN semantics, the boundary event is only relevant if it occurs in between the *begin* and *termination* of the associated activity. Thus, once the associated activity terminates, the boundary event becomes obsolete. Therefore, the unsubscription of a boundary event should be done either after consuming the event, or when the associated activity terminates. The Petri net mapping for unsubscription of boundary event therefore contains two incoming flows to the place before the U_e ; one from C_e , and the other one from A_t .

SEMANTIC RESOLUTION: EXCLUSIVE EVENT. If an exclusive event is subscribed at event enablement, then it is done after the particular branch is already chosen. However, if subscription is done earlier, then all the events situated in different optional branches can occur before the control flow reaches the XOR gateway. In this case, as soon as the decision is taken at the gateway and one branch is selected, the events in other branches become obsolete. This is shown in the Petri net module with the place $P(d_j, U_{e_i})$, where $i, j \in \mathbb{N}_0$, depicting the i -th and j -th branch after the XOR gateway, and $i \neq j$.

SEMANTIC RESOLUTION: RACING EVENT. Immediately after one of the events following an event-based gateway occurs, the process takes the path led by that event, and the other events become obsolete. This makes not only the event occurrence, but also the order of event occurrence significant for racing events. If subscription is done earlier, the temporal order of the event occurrences should be checked to make sure that the process consumes the event that occurred first. In Figure 41, an outgoing flow is added from each event occurrence to the transition representing combined unsubscription for all racing events $(U_{e_1, e_2, \dots, e_n})$. This ensures the correct order of event consumption, since the occurrence of an event passes the token to the input place for event consumption and at the same time, it also enables the unsubscription of all other events that were in race.

7.2.4 Event Buffering

This part of the chapter formalizes the concepts of event handling from an event processing platform’s perspective. This includes registering the subscription query, detecting a matching event, putting that in the

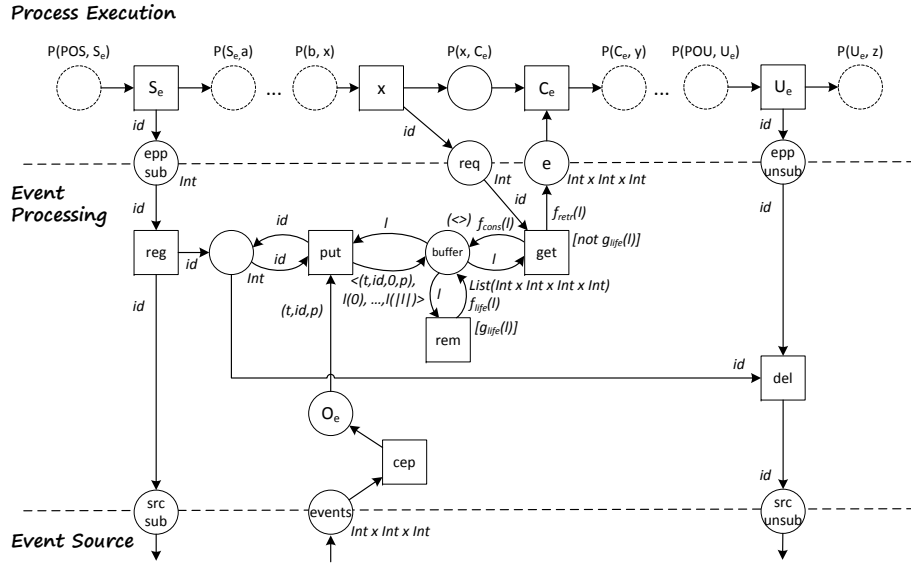


Figure 42: Petri net showing event matching, correlation, and buffering.

buffer until the process execution claims it, and deleting the event query when an unsubscription is issued. The semantics of our event model is defined using the CPN, as shown in Figure 42. The net is parameterized by functions used in the arc inscriptions to support the buffer policies that will be explained later in this section. In essence, the net illustrates the interplay of three levels: The business process execution, the event processing platform (EPP) including the buffer functionality, and the event source(s) in the environment. Similar to Figure 35 presented in the beginning of this chapter, the event source is captured only by its interface. However, now it communicates with the event processing layer instead of the process execution. The interaction points to the event source are given by three places: One capturing subscriptions forwarded by EPP, one modelling the events detected from the environment, and one passing the unsubscription from EPP to the event sources.

The process execution layer sends the subscription to the event processing layer, shown as the place *epp sub*. The subscription query is abstracted from the model. Eventually the EPP forwards the token to the actual event source in the environment, shown as the place *src sub*. The EPP also keeps the token internally to correlate with the tokens representing event notification. These tokens have a product colourset, which represents the event’s timestamp, identifier, and payload $\langle t, id, p \rangle$. We model all of them as integers ($\text{Int} \times \text{Int} \times \text{Int}$).

Once the events are received in EPP via the interface *events*, the complex event processing techniques are applied, represented as the transition *cep*. The events that match with a registered subscription query are accepted as occurrence of relevant events, represented by the place O_e .

This conforms with the causality that a subscription should exist before an event occurrence becomes relevant for the process execution.

Next, transition *put* stores the matching events in the buffer. Events are kept in place *buffer*, which has a list colourset over the colourset of stored events, initially marked with a token carrying an empty list (denoted by $\langle \rangle$). The buffer is extended with an additional integer signifying a consumption counter. Once fired, transition *put* adds the data of the token consumed from place O_e to this list.

Coming back to the process execution layer, at some point the control-flow reaches the event node and the process is now ready to consume the event stored in the buffer. This is done in a two fold manner. First, the process instance requests an event from the buffer, as shown by the token in the place *req*. This is done when the previous node of the event (x) is terminated. Then, the process actually receives the event, represented by a token in the place *e*. Whereas inside the event processing layer, transition *get* extracts the event from the buffer when requested. This ensures that consumption of an event is possible only when there is a subscription as well as the event has actually occurred.

When the process execution reaches the point of unsubscription, *epp unsub* is passed to the EPP along with the *id*. The transition *del* fires upon receiving the unsubscription request, and consumes the token produced by *req* for the same *id*, so that the event query is removed. Also, the unsubscription request is forwarded to the event source, as shown by the place *src unsub*. This satisfies the interdependency between subscription and unsubscription.

The transition *rem* in the buffer model controls which events shall be kept in the buffer. This transition may fire if the transition guard g_{life} evaluates to true. Upon being fired, it applies function f_{life} to the list of the token in place *buffer*. This models the lifespan policy of the buffer. The transition *get* is enabled as long as the transition to implement the lifespan policy is disabled. The arc inscriptions f_{cons} and f_{retr} model the consumption policy and the retrieval policy, respectively. The buffer policies implemented by instantiating the respective guards and functions are described in detail below.

LIFESPAN POLICY. This policy is realized via the guard condition g_{life} , and function f_{life} . The configurations can be defined as follows:

- *Specific Length:* Assuming that at most k events shall be kept in the buffer, the guard condition for the transition to remove events checks for the length of the respective list, i.e., g_{life} is defined as $|l| \geq k$. Then, function f_{life} selects only the k most recent events, i.e., $f_{\text{life}}(l) \mapsto \langle l(n - k + 1), \dots, l(|l|) \rangle$.
- *Specific Time:* Assuming that there is a global variable g in the CPN model that indicates the current time and a time window of k time units, the guard g_{life} checks whether some event fell

out of the window, $l(i) = (t, id, n, p)$ with $t < g - k$ for some $0 \leq i \leq |l|$. The respective events are removed, i.e., $f_{life}(l) \mapsto l'$ where $l'(j) = l(i) = (t, id, n, p)$ if $t \geq g - k$ and for $i - j$ events $l(m) = (t', id', n', p')$, $m < i$ it holds that $t' < g - k$.

- *Keep All*: In this case, the guard g_{life} is simply set to false, so that the function f_{life} does not have to be defined.

RETRIEVAL POLICY. The retrieval policy is connected to the function f_{retr} . The different configurations can be achieved as follows:

- *Last-In-First-Out*: The last event of the list is retrieved in this case, defined as $f_{retr}(l) \mapsto l(|l|)$.
- *First-In-First-Out*: Here, the head of the list of events is retrieved, considering the function $f_{retr}(l) \mapsto l(0)$.
- *Attribute-based*: With π as a selection predicate evaluated over the payload of events, the first of events that satisfies the predicate is retrieved, i.e., $f_{retr}(l) \mapsto (t, id, n, p)$, with $l(i) = (t, id, n, p)$, such that $\pi(p)$ holds true and for all $l(j) = (t', id', n', p')$, $j < i$, $\pi(p')$ is not satisfied.

CONSUMPTION POLICY. Function f_{cons} is used to implement the consumption policy. This can be defined as:

- *Consume*: The event retrieved from the buffer, assuming its position in the list of events l is i , is consumed, i.e., not written back to the buffer. This is captured by the following definition of the function implementing the consumption policy: $f_{cons}(l) \mapsto \langle l(1), \dots, l(i-1), l(i+1), \dots, l(|l|) \rangle$.
- *Reuse*: The event is not removed from the buffer, i.e., $f_{cons}(l) \mapsto l$.
- *Bounded Reuse*: Assuming that an event can be consumed k -times and with $l(i) = (t, id, n, p)$ being the retrieved events, the function to implement the consumption policy is defined as: $f_{cons}(l) \mapsto \langle l(1), \dots, l(i-1), l(i+1), \dots, l(|l|) \rangle$, if $n \geq k$, and $f_{cons}(l) \mapsto \langle l(1), \dots, l(i-1), (t, id, n+1, p), l(i+1), \dots, l(|l|) \rangle$, otherwise.

7.3 SUMMARY & DISCUSSION

The thorough formalization presented above reveals the internal behaviors of process execution and event processing when it comes to communicating via events. The Petri net mappings assign strong formal grounding and clear execution semantics to the conceptual framework introduced in [Chapter 6](#). The Petri net modules for event constructs, points of subscription, and points of unsubscription provide a complete

understanding of the event handling notions. Defining each buffer policy individually makes the event buffering efficient enough to suit a vast range of operations.

Both Petri nets and CPNs are intensely used for analyzing business processes [37, 137]. Also, there exist standard ways to transform CPNs to normal Petri nets [62]. The processes enhanced with event handling configurations are, therefore, ready and well-suited for formal analysis.

The formalization opens several possibilities to explore the significance of flexible event handling. Two such applications along with the impact of event handling configurations are described in the next part of the thesis (Chapter 8).

Part III

EVALUATION & CONCLUSIONS

The core concepts of this thesis have been introduced and elaborated in [Part II](#).

In this part, the concepts are evaluated. This chapter highlights two applications of the flexible event handling model introduced in [Chapter 6](#) to show the relevance of event handling configurations in process verification. The applications are based on the formal semantics defined in [Chapter 7](#). On one hand, execution trace analysis shows how the interdependencies among event handling notions play a role in correct process execution. Reachability analysis, on the other hand, shows the range of allowed behaviors for a communication model consisting of a process and its environment.

8.1 EXECUTION TRACE ANALYSIS

Process execution traces are used widely for conformance and correctness checking of business processes [26, 76, 139]. This part of the chapter discusses the correct process execution behavior considering the newly introduced event handling semantics. [Section 8.1.1](#) specifies the constraints for each event construct varying with precise event handling configurations which are needed to be true for correct execution of a process. The impact of choosing different configurations is explored in [Section 8.1.2](#). An initial version of the trace analysis constraints has been published in “A Flexible Event Handling Model for Business Process Enactment” [79].

8.1.1 Correctness Constraints

The correctness constraints are temporal constraints based on the formal semantics of event constructs, points of subscription, points of unsubscription, and the interdependencies among them. All of the traces conform to the basic temporal order $S_e < O_e < C_e \wedge S_e < U_e$ for subscription, occurrence, consumption, and unsubscription. In addition, the constraints according to the chosen subscription configurations are specified. The variation in the traces are highlighted whereas the paths that remain same are shown in gray. The Petri net modules sketched previously satisfy the constraints. For each event construct, a subset of the allowed behaviors defined by the corresponding Petri nets are given below.

TRACES FOR MANDATORY EVENT. The temporal constraints complying with correct execution behavior for a process containing a mandatory event e are given for each *POS*. Unsubscription is chosen to be done at event consumption (*POU1*). The combined Petri net for mandatory

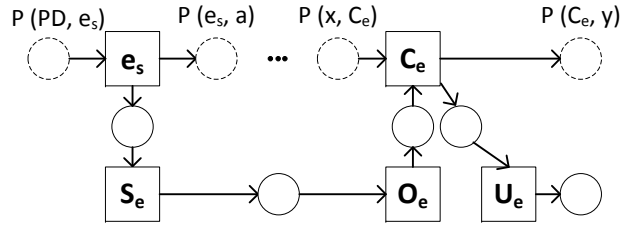


Figure 43: Petri net showing a *mandatory event construct with subscription at process instantiation and unsubscription at event consumption.*

events with subscription at process instantiation and unsubscription at event consumption is given in [Figure 43](#). For the rest of the Petri net modules, refer to [Figure 37](#) and [Figure 41](#).

Subscription at event enablement:

Subscription to event e should be done only after the previous transition x is executed, i.e., $x < S_e$ must hold.

$EI, PD, e_s, a, \dots, x, S_e, O_e, C_e, U_e, y, \dots, e_e$

Subscription at process instantiation:

Subscription should be done immediately after the start event has occurred; the event is consumed after x is executed, i.e., $e_s < S_e < x < C_e$ must hold.

$EI, PD, e_s, S_e, a, \dots, x, O_e, C_e, U_e, y, \dots, e_e$

$EI, PD, e_s, S_e, a, \dots, O_e, \dots, x, C_e, U_e, y, \dots, e_e$

$EI, PD, e_s, S_e, a, O_e, \dots, x, C_e, U_e, y, \dots, e_e$

$EI, PD, e_s, S_e, O_e, a, \dots, x, C_e, U_e, y, \dots, e_e$

Subscription at process deployment:

Here, the subscription is done after process deployment but before process instantiation. Thus, $PD < S_e < e_s < x < C_e$ must hold.

$EI, PD, S_e, e_s, a, \dots, x, O_e, C_e, U_e, y, \dots, e_e$

$EI, PD, S_e, e_s, a, \dots, O_e, \dots, x, C_e, U_e, y, \dots, e_e$

$EI, PD, S_e, e_s, a, O_e, \dots, x, C_e, U_e, y, \dots, e_e$

$EI, PD, S_e, e_s, O_e, a, \dots, x, C_e, U_e, y, \dots, e_e$

$EI, PD, S_e, O_e, e_s, a, \dots, x, C_e, U_e, y, \dots, e_e$

Subscription at engine initiation:

Subscription is done after engine initiation, before process deployment, i.e., $EI < S_e < PD < e_s < x < C_e$ must hold.

$EI, S_e, PD, e_s, a, \dots, x, O_e, C_e, U_e, y, \dots, e_e$

$EI, S_e, PD, e_s, a, \dots, O_e, \dots, x, C_e, U_e, y, \dots, e_e$

$EI, S_e, PD, e_s, a, O_e, \dots, x, C_e, U_e, y, \dots, e_e$

$EI, S_e, PD, e_s, O_e, a, \dots, x, C_e, U_e, y, \dots, e_e$

$EI, S_e, PD, O_e, e_s, a, \dots, x, C_e, U_e, y, \dots, e_e$

$EI, S_e, O_e, PD, e_s, a, \dots, x, C_e, U_e, y, \dots, e_e$

| CONFIGURATION | MANDATORY EVENT | BOUNDARY EVENT | RACING EVENT | EXCLUSIVE EVENT |
|--------------------------------------|--------------------------------------|----------------------------------|---|--|
| <i>Sub at event enablement</i> | $x < S_e$ | $x < A_b < S_e \wedge O_e < A_c$ | $g < S_{e_1, e_2, \dots, e_n}$ | $g < S_{e_j}$ |
| <i>Sub at process instantiation</i> | $e_s < S_e < x < C_e$ | - | $e_s < S_{e_1, e_2, \dots, e_n} < g < C_{e_i}$ | $e_s < S_{e_j} < g < C_{e_j}$ |
| <i>Sub at process deployment</i> | PD < $S_e < e_s$ < $x < C_e$ | - | PD < S_{e_1, e_2, \dots, e_n} < $e_s < g < C_{e_i}$ | PD < S_{e_j} < $e_s < g < C_{e_j}$ |
| <i>Sub at engine initiation</i> | EI < $S_e < PD$ < $e_s < x < C_e$ | - | EI < $S_{e_1, e_2, \dots, e_n} < PD$ < $e_s < g < C_{e_i}$ | EI < $S_{e_j} < PD < e_s$ < $g < C_{e_j}$ |
| <i>Unsub at event consumption</i> | $C_e < U_e$ | $C_e < U_e$ | $C_{e_i} < U_{e_1, e_2, \dots, e_n}$ | $C_{e_j} < U_{e_j}$ |
| <i>Unsub at semantic resolution</i> | - | $A_t < U_e$ | $C_{e_i} < U_{e_1, e_2, \dots, e_n}$ | $d_i < U_{e_j}$ |
| <i>Unsub at process undeployment</i> | PU < U_e | PU < U_e | PU < U_{e_1, e_2, \dots, e_n} | PU < U_{e_j} |
| <i>Unsub at engine termination</i> | ET < U_e | ET < U_e | ET < U_{e_1, e_2, \dots, e_n} | ET < U_{e_j} |

Table 1: Correctness constraints for event handling configurations

TRACES FOR BOUNDARY EVENT. For a process containing a boundary event e associated with an activity A , the following should hold for a correct execution with *subscription at event enablement* and *unsubscription at event consumption*: $\chi < A_b < S_e \wedge O_e < A_c$. If *unsubscription at semantic resolution* is chosen, the constraint will rather be: $\chi < A_b < S_e \wedge O_e < A_c \wedge A_t < U_e$. Traces below show the possible behaviors, conformant to the Petri net for the boundary event in [Figure 41](#):

$EI, PD, e_s, a, \dots, \chi, A_b, S_e, A_t, U_e, y, \dots, e_e$
 $EI, PD, e_s, a, \dots, \chi, A_b, S_e, O_e, C_e, A_c, U_e, z, \dots, e_e$

TRACES FOR RACING EVENT. Some example traces showing the possible behavior of a net containing two racing events e_1 and e_2 are given below. The *POU* is chosen as *unsubscription at semantic resolution*, as shown in [Figure 41](#). Note that *unsubscription at event consumption* results in the same set of traces in this case, since a common unsubscription is issued for all racing events as soon as one of them occurs. Therefore, for *POU1* and *POU2*, $C_e < U_{e_1, e_2, \dots, e_n}$ also holds.

Subscription at event enablement:

The traces should comply with $g < S_{e_1, e_2, \dots, e_n}$.

$EI, PD, e_s, a, \dots, g, S_{e_1, e_2}, O_{e_1}, U_{e_1, e_2}, C_{e_1}, y_1, \dots, e_e$
 $EI, PD, e_s, a, \dots, g, S_{e_1, e_2}, O_{e_2}, U_{e_1, e_2}, C_{e_2}, y_2, \dots, e_e$

Subscription at process instantiation:

Here, $e_s < S_{e_1, e_2, \dots, e_n} < g < C_e$ must hold.

$EI, PD, e_s, S_{e_1, e_2}, a, \dots, g, O_{e_1}, U_{e_1, e_2}, C_{e_1}, y_1, \dots, e_e$
 $EI, PD, e_s, S_{e_1, e_2}, a, \dots, g, O_{e_2}, U_{e_1, e_2}, C_{e_2}, y_2, \dots, e_e$
 $EI, PD, e_s, S_{e_1, e_2}, a, \dots, O_{e_1}, U_{e_1, e_2}, g, C_{e_1}, y_1, \dots, e_e$
 $EI, PD, e_s, S_{e_1, e_2}, a, \dots, O_{e_2}, U_{e_1, e_2}, g, C_{e_2}, y_2, \dots, e_e$
 $EI, PD, e_s, S_{e_1, e_2}, O_{e_1}, U_{e_1, e_2}, a, \dots, g, C_{e_1}, y_1, \dots, e_e$
 $EI, PD, e_s, S_{e_1, e_2}, a, \dots, O_{e_2}, U_{e_1, e_2}, \dots, g, C_{e_2}, y_2, \dots, e_e$

Similar to mandatory events, $PD < S_{e_1, e_2, \dots, e_n} < e_s < g < C_e$ must hold for *subscription at process deployment* and $EI < S_{e_1, e_2, \dots, e_n} < PD < e_s < g < C_e$ must hold for *subscription at engine initiation*.

TRACES FOR EXCLUSIVE EVENT. The following traces show a subset of allowed behaviors of the exclusive event construct e_1 and e_2 shown in [Figure 40](#). The temporal constraints for each *POS* are specified as well. For *unsubscription at event consumption*, the usual condition $C_{e_i} < U_{e_i}$ holds. In case *unsubscription at semantic resolution* is chosen, $d_j < U_{e_i}$ should be true, where d_j signifies the first node on j -th branch given that $i \neq j$.

Subscription at event enablement:

The traces for this *POS* should comply with $g < S_{e_i}$.

EI, PD, $e_s, a, \dots, g, d_1, \dots, x_1, S_{e_1}, O_{e_1}, C_{e_1}, U_{e_1}, y_1, \dots, e_e$

EI, PD, $e_s, a, \dots, g, d_2, \dots, x_2, S_{e_2}, O_{e_2}, C_{e_2}, U_{e_2}, y_2, \dots, e_e$

Subscription at process instantiation:

The constraint for this *POS* is $e_s < S_{e_j} < g < C_{e_j}$. In addition, we select *unsubscription at semantic resolution*. This means, once branch 1 is chosen, e_2 is unsubscribed following semantic resolution, as shown in [Figure 41](#). However, e_1 is only unsubscribed once it has occurred, i.e., at consumption. Therefore, $e_s < S_{e_j} < g < C_{e_j} \wedge d_i < U_{e_j}$ must be true here.

EI, PD, $e_s, S_{e_1, e_2}, a, \dots, g, d_1, U_{e_2}, \dots, x_1, O_{e_1}, C_{e_1}, U_{e_1}, y_1, \dots, e_e$

EI, PD, $e_s, S_{e_1, e_2}, a, \dots, g, d_2, U_{e_1}, \dots, x_2, O_{e_2}, C_{e_2}, U_{e_2}, y_2, \dots, e_e$

EI, PD, $e_s, S_{e_1, e_2}, a, O_{e_1}, \dots, g, d_1, U_{e_2}, \dots, x_1, C_{e_1}, U_{e_1}, y_1, \dots, e_e$

EI, PD, $e_s, S_{e_1, e_2}, O_{e_1}, a, \dots, g, d_2, U_{e_1}, O_{e_2}, \dots, x_2, C_{e_2}, U_{e_2}, y_2, \dots, e_e$

For *subscription at process deployment* and *subscription at engine initiation*, $PD < S_{e_j} < e_s < g < C_{e_j}$ and $EI < S_{e_j} < PD < e_s < g < C_{e_j}$ must hold, respectively.

[Table 1](#) summarizes the correctness constraints for each point of (un)-subscription, categorized by the event constructs. The event construct is detected at design time following the definitions provided in [Section 6.2](#). Depending on the usecase need and data dependency for subscription, the event handling configurations are chosen. The complete clause for correctness constraints is derived combining the constraints for *POS* and *POU* with a logical AND operator. To this end, the trace analysis is done to evaluate the process execution traces.

8.1.2 Impact of Event Handling

This section turns to apply the correctness constraints defined above to the motivating example illustrated in the motivating example presented in [Figure 27](#) in [Chapter 6](#). We consider only the logistics company's process, i.e., the process executed by "Process Engine". We identify the intermediate catching events "Confirmation Received", and "Traffic Update" as mandatory event constructs and the other events as racing event constructs. We limit the analysis to the point until the event "Traffic Update" is received, as shown in [Figure 44](#).

Now, we select the event handling configurations to be *subscription at event enablement* and *unsubscription at event consumption* for both the mandatory events. The resulting Petri net including transitions for engine initiation and process deployment is given in [Figure 45](#) (the labels of the process nodes are abbreviated).

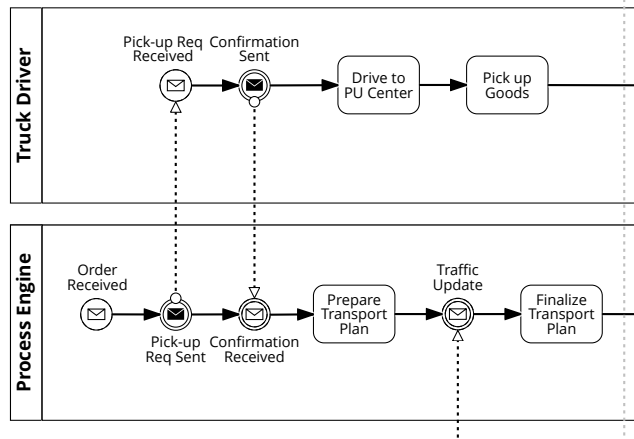


Figure 44: Excerpt from motivating example presented in Figure 27 in Chapter 6.

The correct trace with these configurations is:

$EI, PD, OR, PRS, S_{CR}, O_{CR}, C_{CR}, U_{CR}, PTP, S_{TU}, O_{TU}, C_{TU}, U_{TU}, FTP$

Next, we change the point of subscription to *subscription at process instantiation*. The point of unsubscription remains the same. The Petri net with the changed event handling configurations is shown in Figure 46.

The set of correct traces is:

$EI, PD, OR, S_{CR, TU}, PRS, O_{CR}, C_{CR}, U_{CR}, PTP, O_{TU}, C_{TU}, U_{TU}, FTP$
 $EI, PD, OR, S_{CR, TU}, PRS, O_{CR}, C_{CR}, U_{CR}, O_{TU}, PTP, C_{TU}, U_{TU}, FTP$
 $EI, PD, OR, S_{CR, TU}, PRS, O_{CR}, C_{CR}, O_{TU}, U_{CR}, PTP, C_{TU}, U_{TU}, FTP$
 $EI, PD, OR, S_{CR, TU}, PRS, O_{CR}, O_{TU}, C_{CR}, U_{CR}, PTP, C_{TU}, U_{TU}, FTP$
 $EI, PD, OR, S_{CR, TU}, PRS, O_{TU}, O_{CR}, C_{CR}, U_{CR}, PTP, C_{TU}, U_{TU}, FTP$
 $EI, PD, OR, S_{CR, TU}, O_{TU}, PRS, O_{CR}, C_{CR}, U_{CR}, PTP, C_{TU}, U_{TU}, FTP$

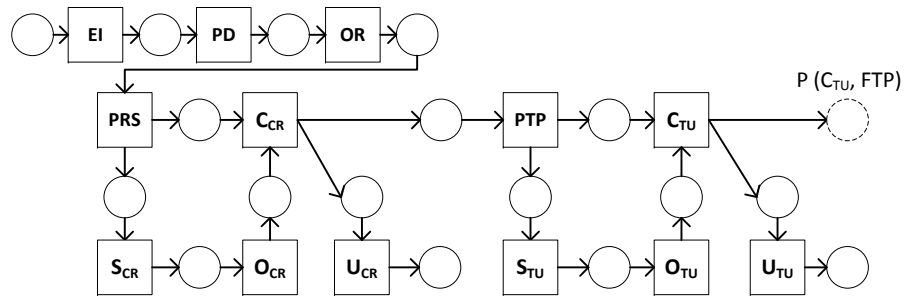


Figure 45: Petri net with *subscription at event enablement* and *unsubscription at event consumption*.

8.1.3 Discussion

From the traces presented above, it is evident that for the intermediate event construct “Confirmation Received”, even if the subscription

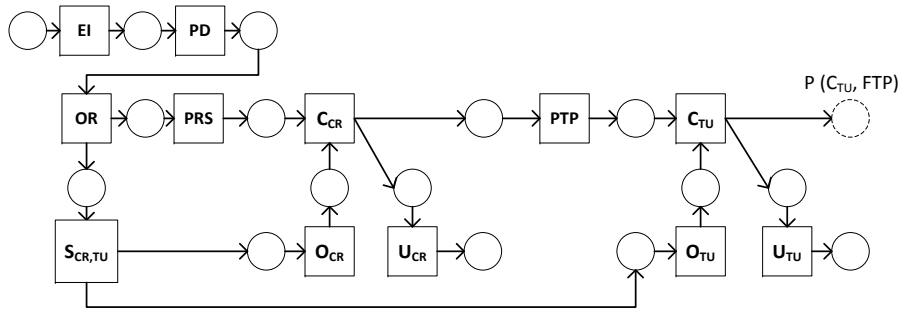


Figure 46: Petri net with *subscription at process instantiation* and *unsubscription at event consumption*.

is done at different points, the event CR will always take place after the previous node PRS is executed. The reason is the process causality explained in [Section 6.1](#) which restricts the temporal ordering of an event occurrence based on the completion of predecessor tasks. Existing BPMN semantics are perfectly applicable for these scenarios.

However, for the intermediate event construct “Traffic Update”, the traces show more variety with different event handling configurations. Depending on the time of subscription, the event integration gets more accommodating here. If the traffic update is published only when there is a significant change in the traffic situation, and that happens earlier than the process engine is ready to consume it, the information carried by the event can still be used. In case the traffic update events are published periodically and there are several update events by the time the logistics company actually needs the information to finalize the transport plan, they can either refresh the buffer to store just the latest event, or consume the last event stored in the buffer using the buffer policies. In this case, the use of a flexible event handling model decreases the probability of the process execution getting delayed or stuck forever due to the lag between event occurrence and the process being ready for consumption.

8.2 REACHABILITY ANALYSIS

Since process models are at the heart of business process management (see [Section 2.1](#)), techniques for workflow verification are extensively used for model checking [[3](#), [123](#)]. Especially *soundness* has been established as a general correctness criterion for process models, requiring the option to reach a final state, a proper characterization of the final state, and the absence of dead activities that cannot contribute to process execution. Most of the formal verification techniques, however, focus on control flow structure. There are some works that combine data values with control flow for soundness analysis or decision conformance in workflow [[15](#), [115](#)]. Nevertheless, none of these works

consider event handling information, although events have significant influence on process execution.

Acknowledgements

The verification problem we focus on is to analyze reachability for a certain state in a process, under a certain event handling configuration. This part of process verification is an ongoing work in collaboration with *Prof. Dr. Matthias Weidlich*.

To address the reachability problem, we choose a path in the process model \mathfrak{M} leading to an intermediate event execution, where the event is equipped with specific event handling configurations. As a next step, we try to find a set of paths in the environment that will ensure the process reaches a desired state, referred to as *corresponding path(s)*.

The event handling concepts described in [Chapter 6](#) constitute several configurations that can be used for an event subscription. For instance, subscription and unsubscription can be done at different points, events can be buffered or not, and a single event can be consumed once or multiple times. We scope the current verification with the following two configurations of event handling:

- Subscription configuration: by setting the point of subscription to *subscription at event enablement* (without buffering) or *early subscription* (with buffering).
- Consumption configuration: by selecting the consumption policy for the buffer as *consume* or *reuse*.

In [Section 8.2.1](#), we introduce the formal communication model representing the interaction behavior of a process and the environment. Later, in [Section 8.2.2](#), we discuss how event handling configurations might extend the possible behavior of the environment to complement the process model interaction.

8.2.1 Communication Model

The communication model is built with a process model and its corresponding environment model. The process listens to the environment by receiving events, and reacts on the environmental occurrences following the process specification for further activities, decisions, and the events generated during process execution, i.e., the sending events. The internal behavior of the environment is usually unknown to the process and vice versa. Since the current process verification considers only the interactions between process and environment, and is independent of the internal behaviors, a transition system represents the communication model efficiently.

Definition 8.1 (Transition System).

A *Transition System* is a tuple $TS = (S, T, s_0)$ with

- a set of states S ,

- a set of transitions T , and
- an initial state $s_0 \in S$.
- Let the set of events $E = \{a, \dots, z\}$, where $\{a, \dots, z\}$ are the labels of events.
- Let the set of interactions $IE = \{?, !\} \times E \cup \{\tau\}$, where $!$ is the interaction mode for sending an event, $?$ is the interaction mode for receiving an event, and τ is the time interval.
- The set of transitions $T \subseteq S \times IE \times S$.

◆

A motivating example for reachability analysis including event handling is given in Figure 47. Using the transition diagram notation, only the interaction points are shown and internal activities are abstracted. We model both the process and the environment with separate transition systems. The filled circles represent states and the arrows represent the transitions responsible for the state changes. The initial state is marked with an additional arrow that has no previous state. The question mark (?) signifies receiving an event, whereas the exclamation mark (!) depicts sending of an event. The transitions are labeled with the events along with the interaction mode. The transition showing the 10 min interval in the process model is an example of τ .

Syntax of transition system

The process model interaction shows that after sending two messages x and y , the process waits for z . After receiving z , the process sends w . If z does not occur within those 10 min, the system goes to an error state, represented by the event e . From the model alone, we cannot tell how probable it is to reach the error state.

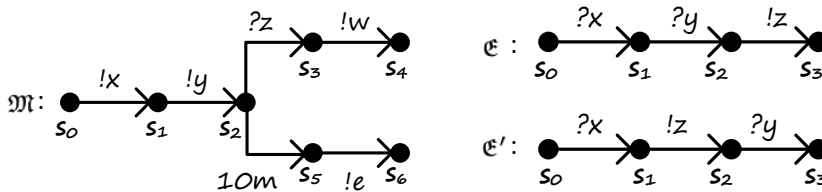


Figure 47: Transition diagram for a process model (\mathcal{M}) and corresponding environment models (\mathcal{E} , \mathcal{E}').

If we mirror the interactions of \mathcal{M} , we get \mathcal{E} , where upon receiving x followed by y , z is produced. Considering \mathcal{E} as the environment model, we see that z will definitely be published after y is received. If we assume that the transitions do not take time, i.e., z is published right after receiving y , the system will never reach the error state and the process remains always sound. However, the environment model does not necessarily map to process interaction order. For instance, \mathcal{E}' suggests another environment model where z is produced after receiving x , and only after that the environment expects y and z . Since z is already produced, the process misses it, leading to the error state for

sure. This situation can be avoided by issuing an early subscription [81]. For example, if subscription to z is made before or immediately after sending x , the process will not miss out on the event occurrence.

An early subscription in this context can be whenever the process has enough information to subscribe to the specific event source. This can be immediately after process instantiation or anytime afterwards until the event node is reached. According to the transition system, the point of subscription means a state in between initial state and the state that

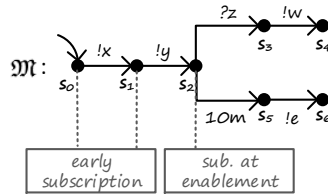


Figure 48: Different subscription configurations for receiving event z .

signifies the consumption of the event. In Figure 48, if we consider the subscription for event z in the process model \mathfrak{M} , subscription at event enablement will be represented by the state s_2 . On the other hand, early subscription will mean any of the states s_0 or s_1 . Since the

same event can be received more than once in a process, the state representing point of subscription for a specific event has to be on all the paths that lead to a transition carrying the same label as the event.

8.2.2 Impact of Event Handling

Let us consider the process model \mathfrak{M} visualized in Figure 49 where the process is instantiated at s_0 . Sending the event x takes the process into its next state s_1 . Now, two things can happen. The process might receive y and go to state s_2 . Otherwise, z is received twice in a row and the process goes to s_4 , via the intermediate state s_3 . The rest of the section describes how the set of corresponding paths differ for the states in \mathfrak{M} , depending on the different event handling configuration chosen for the events to be received by \mathfrak{M} .

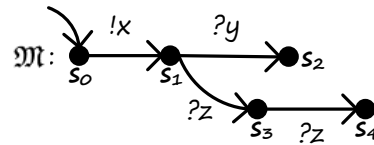


Figure 49: Process Model \mathfrak{M} communicating with environment.

SUBSCRIPTION CONFIGURATION. The reachability of the state s_3 is of concern for the first verification assignment. The path to satisfy $F(s_3)$ in \mathfrak{M} would be the sending of x followed by the receiving of z . We select the subscription configuration as *subscription at enablement*, (i.e., subscription is done at s_1) and the consumption configuration as *consume* to begin with. For this event handling configuration, we can simply mirror the path in \mathfrak{M} and find the corresponding path in \mathcal{E} . The corresponding path in this case would be $?x$ followed by $!z$, as shown in Figure 50. The environment can not emit z before receiving x for this scenario.

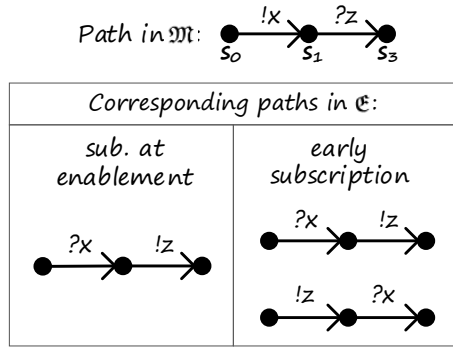


Figure 50: A chosen path in process and the set of corresponding paths in environment with varying *subscription configuration*.

Next, we change the subscription configuration to *early subscription*. The mirrored path still belongs to the set of corresponding paths. However, say we consider the early point of subscription as s_0 , and start buffering the events. With the current configuration, even if the environment emits z before it receives x , we can get the notification and store z in buffer for later consumption. This shows how moving the subscription to an earlier point adds more behavior of the environment as corresponding path, resulting in more flexibility for the process communication.

CONSUMPTION CONFIGURATION. The verification assignment now turns to the impact of consumption policy of the buffer. Here, the reachability of state s_4 is verified, i.e., $F(s_4)$. The path in \mathfrak{M} for this is sending of x , followed by receiving z two times consecutively. We set the subscription configuration to be *early subscription* to store the events in a buffer. Selecting the consumption configuration as *consume* will result in deleting the event from the buffer as soon as it is used in the process. Therefore, the environment needs to produce two events of type z to satisfy the corresponding path requirement. The set of corresponding paths for different consumption policies are shown in [Figure 51](#).

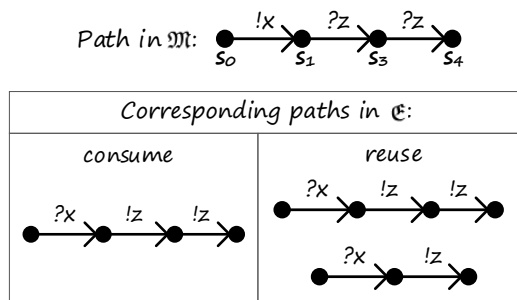


Figure 51: A chosen path in process and the set of corresponding paths in environment with varying *consumption configuration*.

To compare the impact, we now change the consumption configuration to *reuse*. The subscription remains the same as before. Since event information can be used more than once, it is sufficient if the environment produces z only once. This leads to the set of corresponding paths containing two options rather than one. Again, setting the event handling configuration differently extends the allowed environmental behavior for the same process path.

8.2.3 Discussion

The last section shows how different event subscription and consumption configurations in a process result in different sets of corresponding paths in the environment. Having shown the influence of event handling configurations on workflow reachability, this section turns to exploit further applications of the concepts. Later, the exceptions and further restrictions in applicability are discussed.

FURTHER APPLICATION IDEAS. The workflow verification with additional event handling concepts leads to further application areas such as *synthesis*. If process model and event handling configuration are known, assumptions on the environment can be synthesized. First, the *controllability* can be evaluated by checking if there exists at least one environment model that complements the process model. From the above discussion, it can be answered as below:

\forall paths of \mathfrak{M} , \exists corresponding set of paths P .

If $\exists p \in P$ in \mathfrak{E} , then \mathfrak{E} completes the composed process model.

This further leads to adapter synthesis to generate an environment that supports correct composition of a communication model. Another area of application can be to extend the formal properties that can be easily verified once the reachability analysis has been done for a process model. For example, it can be concluded that “*any corresponding path for subscription at event enablement is also a corresponding path for early subscription of the same event*”.

LIMITATIONS OF THE APPROACH. Though the event handling notions make workflow verification more precise and powerful, there are certain scenarios where a more careful application of the concepts is required. So far we have considered the event handling configurations only for subscription and consumption. In [Figure 52](#), a slightly different version of M is shown where instead of two z in a row, it needs z followed by w . For reachability of $F(s_4)$ with early subscription and reuse for both z and w , the set of corresponding paths are listed in the figure. Additionally, if we include the retrieval policy, then not only the event occurrences, but also the order of their occurrences will matter.

Let’s say we select the retrieval configuration as *FIFO*. According to the process semantics, when x is sent, \mathfrak{M} will wait for z . The cor-

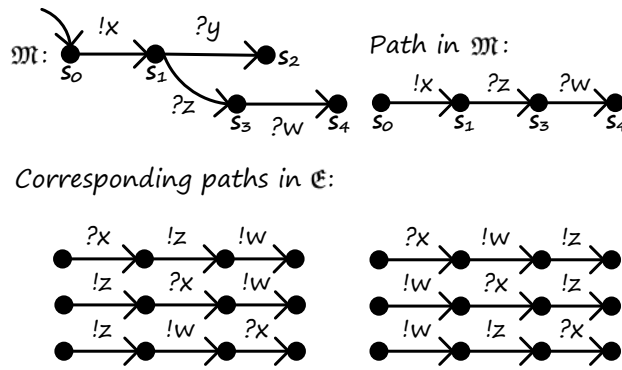


Figure 52: A chosen path in the process and the set of corresponding paths in environment with *early subscription* for z and w .

responding paths listed on the left will fit the need as whenever the process is done sending x , it can consume first z and then w from the buffer. Note that among the list of corresponding paths, there are three parts (listed on the right) where w occurs before z . In these cases, even if z is there in the buffer, it will be blocked by w . The process will thus get stuck due to using the wrong event handling configuration. On the contrary, selecting the retrieval configuration as *LIFO* should exclude the corresponding paths on the left side from correct process behavior.

In addition, depending on the modeling notation used to describe the workflow, there might be further restrictions on verification. Figure 53 shows a process modeled with BPMN [94]. After the sending task A , the control flow enables the subprocess consisting of tasks B and C . Once C is completed, the process executes E and reaches the end state. However, while the subprocess is running, the occurrence of the event e will abort the ongoing task and trigger the exceptional path leading to D . The corresponding transition diagram is visualized such that the occurrence of e after s_1 or s_2 will change the state to s_5 instead of s_3 .

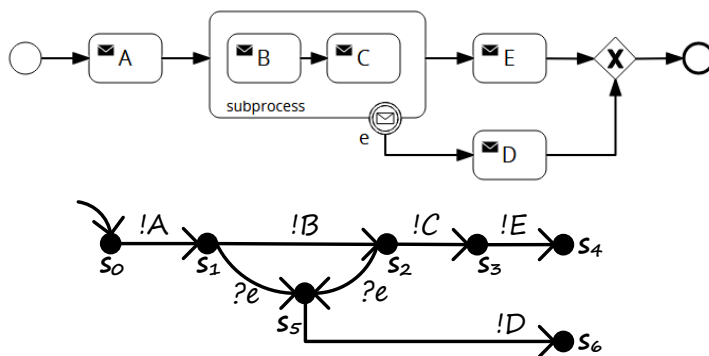


Figure 53: BPMN process model with boundary event and the corresponding transition system.

According to BPMN semantics for attached boundary events, the subscription is issued once the associated subprocess (or activity) is activated. Thus, the state representing subscription at enablement is s_1 . Using the flexible event handling configuration, the process can start listening to e earlier, for example already at s_0 . But this will violate the semantics of BPMN boundary event since the occurrence of the events is only valid during the running phase of the subprocess. Therefore, using event buffering in this case will add incorrect behavior to process execution.

8.3 SUMMARY

This chapter revisited two interesting and popular applications in process verification, namely, execution trace analysis to verify correctness, and reachability analysis to verify soundness. The applications are based on formal semantics of the event handling configurations introduced as the core concepts of the thesis. Both the applications discuss the impacts of event handling concepts with the help of example processes. Correctness criteria are given for each subscription and unsubscription point that ensure correct event handling semantics during execution of a process instance. Trace analysis based on those correctness criteria shows that choosing an early point of subscription increases the time window to get notifications about an event occurrence, and decreases the chance of a process missing out on a published event that is still relevant for process execution. The verification of process reachability is done by expressing a process as a transition system, choosing a specific path with specific event handling configurations, and finding the set of corresponding paths in the environment the process is interacting to. This enhanced reachability analysis shows that early subscription and reuse of event information allows for more behaviors of the environment to complement process execution. In essence, both the applications strengthen the claim that considering the event handling concepts explicitly adds flexibility to event-process communication.

PROOF-OF-CONCEPT IMPLEMENTATION

After discussing the significance of flexible event handling in process execution, this chapter now evaluates the feasibility of event handling concepts. First, we present the implementation details of the integration framework discussed in [Chapter 5](#). This includes individual descriptions of the components Gryphon (process modeler), Chimera (process engine), Unicorn (event processing platform); and the communication steps between them which realize the event-process integration. Later, we extend the basic framework with flexible event handling notions presented in [Chapter 6](#). For this part of the implementation we use the Camunda process engine to strengthen the generic applicability of our approach. Along with the papers publishing the concepts, the implementation work has been discussed in detail in the Master thesis “Flexible Event Subscription in Business Processes” [136]. Also, part of the work has been showcased in the demo papers “Unicorn meets Chimera: Integrating External Events into Case Management” [20], and “Testing Event-driven Applications with Automatically Generated Events” [126].

9.1 BASIC EVENT INTERACTION

We presented the system architecture in [Section 5.3](#) (refer to [Figure 25](#)). The generic architecture is realized with specific components and interfaces as shown in [Figure 54](#). Also, the sequence of integration is visualized with five steps. In the following we first discuss each of the components in detail. Further, we outline the steps of integration in a sequential manner.

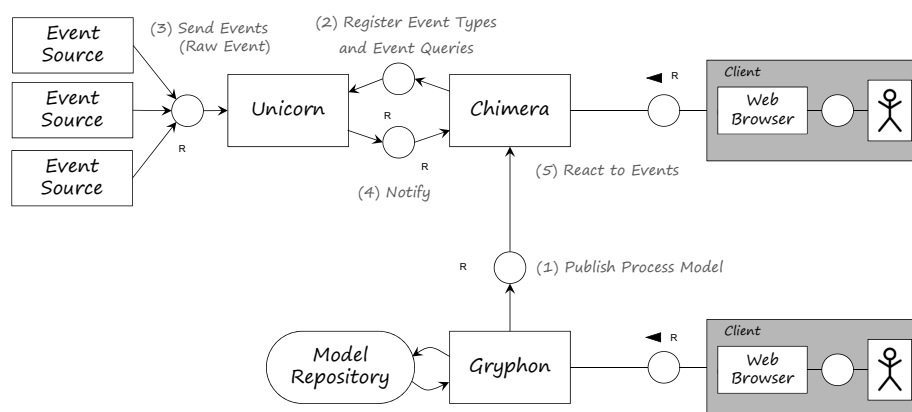


Figure 54: Detailed system architecture showing specific components and the sequence of integrating events into processes.

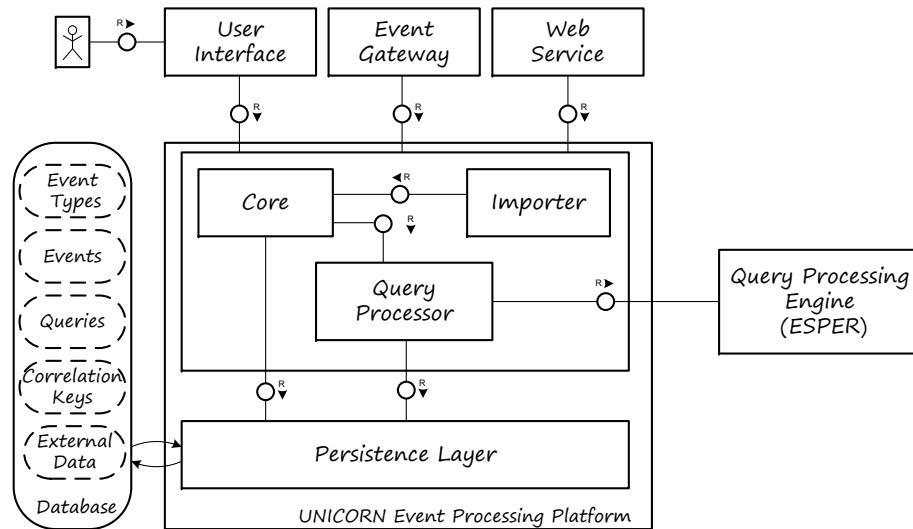


Figure 55: Architecture of event processing platform Unicorn.

9.1.1 Unicorn Event Processing Platform

Unicorn is a complex event processing platform built at the chair of Business Process Technology¹ at Hasso Plattner Institute, University of Potsdam. The first version of Unicorn was developed in the context of the *Green European Transport (GET) Service* project² [18], funded by European Union (October 2012 – September 2015). Since then, it has been extended to add more features needed for several research and student projects. The complete Unicorn project is open-source software, the current version is available under the GNU General Public License 2.0 on GitHub³.

Unicorn is a web-based platform, written in Java, using Wicket⁴ as a front-end framework. The event types, event queries, and notifications can be managed both via a web-based UI and a REST API⁵. The options available to receive notifications are via email, through the Java Message Service (JMS)⁶, by calling back registered REST endpoints, or by viewing them in the UI. For event processing, Unicorn is connected to the Esper engine⁷. Esper is a complex event processing engine, developed and marketed by EsperTech, and uses the expressive query language Esper EPL⁸.

¹ <https://bpt.hpi.uni-potsdam.de/Public/>

² <http://getservice-project.eu/>

³ <https://github.com/bptlab/Unicorn>

⁴ <https://wicket.apache.org/>

⁵ <https://restfulapi.net/>

⁶ <https://docs.spring.io/spring-integration/docs/2.0.0.M2/spring-integration-reference/html/ch19s06.html>

⁷ <http://www.espertech.com/products/esper.php>

⁸ <http://esper.espertech.com/release-7.1.0/esper-reference/html/index.html>

The internal architecture of Unicorn is modeled in [Figure 55](#). As seen in the *Fundamental Modeling Concepts (FMC) Diagram*⁹, there are several ways Unicorn can connect to event sources. Event sources can send events using Unicorn's REST API or publish them to a specific JMS channel which Unicorn listens to. This is feasible if the code of the event source can be changed or an intermediary gateway is used which collects events, for example from sensors, and forwards them to Unicorn. Historic events represented as comma-separated values (csv) or spreadsheets (xls) can also be parsed and imported as event streams using the Unicorn UI¹⁰.

Another way of getting events in Unicorn is active pulling. This is done by calling the web services or consuming RSS feeds in a periodical manner. For active pulling, adapters have to be configured separately for each event source. Unicorn provides a framework to extend the existing adapters with low amount of effort, which is found in the module *importer*. For testing event-driven applications, Unicorn offers a built-in event generator that uses value ranges and distributions to generate realistic events. More details about the event generator can be found in [\[126\]](#).

The *core* of Unicorn is responsible for triggering and managing operations such as event aggregation and composition based on the Esper rules. The *query processor* connects the core to the *Esper query processing engine*. Additionally, Unicorn maintains a database for storing event types, queries, correlation keys and more, which can be fetched by the core component through a *persistence layer*. The internal communication of Unicorn is discussed more while describing the steps of event integration, found later in this section.

9.1.2 Gryphon Case Modeler

Gryphon is a web-based modeler that builds on a Node¹¹ stack and uses bpmn.io¹². bpmn.js is an open-source BPMN modeler implemented in Javascript by Camunda¹³, while the other components are developed by BPT chair at HPI. Camunda modeler is suitable for modeling the core elements of BPMN, such as different kinds of tasks, sub-processes, split and join gateways, pools and lanes, and data objects. Start and end events, catching and throwing intermediate events like message, timer, signal, link, escalation, conditional, and compensation can also be modeled. Gryphon extends bpmn.io so as to create a data model, i.e., the specification of data classes and attributes used in a process model. The data classes are defined along with the states and valid state transitions for their instances, i.e., data objects at runtime, called object

⁹ <http://www.fmc-modeling.org/>

¹⁰ <https://bpt-lab.org/unicorn-dev/>

¹¹ <https://nodejs.org/>

¹² <http://bpmn.io/toolkit/bpmn-js/>

¹³ <https://camunda.com/>

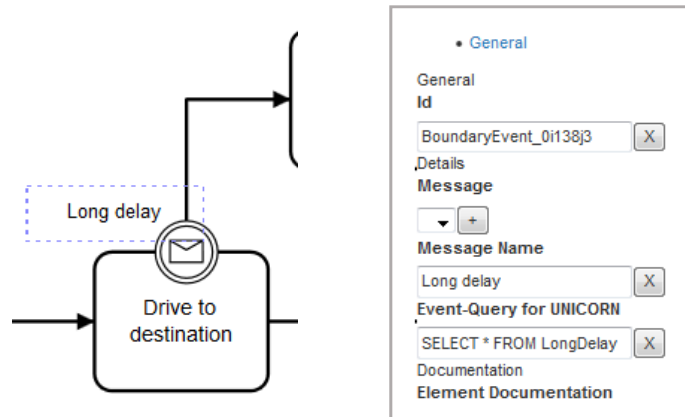


Figure 56: Modeling of subscription queries with extended field (in right) for event annotation in Gryphon.

life cycle. To realize the event integration, we enhanced Gryphon with the functionality to annotate process elements with event annotations and model event types.

The data model editor in Gryphon provides the option to distinguish between data classes and event types. Essentially, both of them are named sets of typed attributes. However, if they are specified as an event type then they are registered in Unicorn at the time of deployment of the process model. We decided to reuse the symbol for catching message events to model external events, since event notifications can be considered as messages. The catching message events can be annotated in Gryphon with event subscription queries which are registered in Unicorn. As Unicorn builds around Esper, the event queries are required to be written in EPL (`SELECT * FROM LongDelay`). These event annotations are used as event binding points, as described in Section 5.3.3. Figure 56 is a screenshot taken of Gryphon that shows an example of modeling event queries, represented as an event annotation. The detailed description of the features can be found here¹⁴ while the source code is published here in GitHub¹⁵.

In addition, for saving the event information carried by the events, we annotate the data object in the modeler. Technically, this mapping is achieved by representing event notifications in the JSON notation¹⁶ and giving a path expression for each attribute of the target data object. Writing data object values from event notifications is depicted in the screenshot of Gryphon in Figure 57 which shows the boundary event and the property editor for the data object. Here, the `LongDelay` event will be stored in the data object `Delay` in the state created and will consist of three attributes — reason, duration, and location.

¹⁴ <https://bpt.hpi.uni-potsdam.de/Gryphon/GettingStarted>

¹⁵ <https://github.com/bptlab/Gryphon>

¹⁶ <http://goessner.net/articles/JsonPath/>

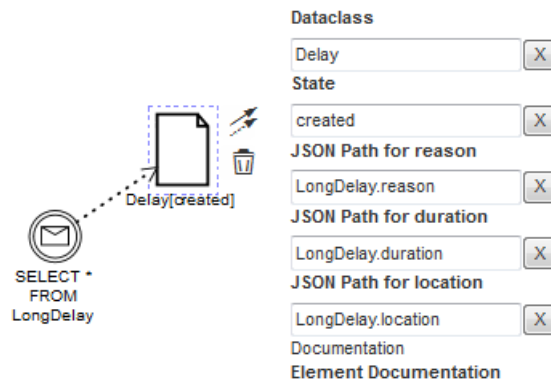


Figure 57: Event data from LongDelay is written into a newly created data object Delay using a JSON path expression.

9.1.3 Chimera Process Engine

The development of the Chimera process engine was initiated at the BPT chair at HPI during a project with Bosch Software Innovations GmbH (2014-2018) [55], focusing on fragment-based case management (fCM) [57]. Since then, it has been extended and adapted vastly for specific applications by students as well as by researchers. Since fCM is a conservative extension to BPMN, processes modeled using BPMN can be executed using Chimera as with any other standard process engine. The current version of Chimera source code can be found on GitHub¹⁷. The documentation and user guide can be accessed here¹⁸. The architecture of the Chimera process engine is presented in Figure 58.

Chimera has a *frontend* built in AngularJS¹⁹ that displays the deployed process models. The dashboard lets the user start a new instance or work on an existing instance. For an ongoing instance, the enabled activities are shown, which users can start and terminate. When terminating an activity, the user can add the values of data object attributes and change the state of a data object according to the data output set specified in the model. The *Activity Log*, *Data object Log*, and *Attribute Log* visualize the history of state changes for activities, data objects, and data attribute values, respectively. The frontend communicates with the engine using a REST endpoint.

The *backend* of the engine has several components. The *parser* fetches the models from the process editor, deserializes the BPMN elements, and stores them in the database. The execution of the process models is controlled by the *core* component following BPMN semantics (and the fCM specification, in case it is a case model). Termination of each activity makes the core recompute the next set of enabled activities. If the next activity is a web service, it is automatically executed by the engine.

¹⁷ <https://github.com/bptlab/chimera>

¹⁸ <https://bpt.hpi.uni-potsdam.de/Chimera/WebHome>

¹⁹ <https://angularjs.org/>

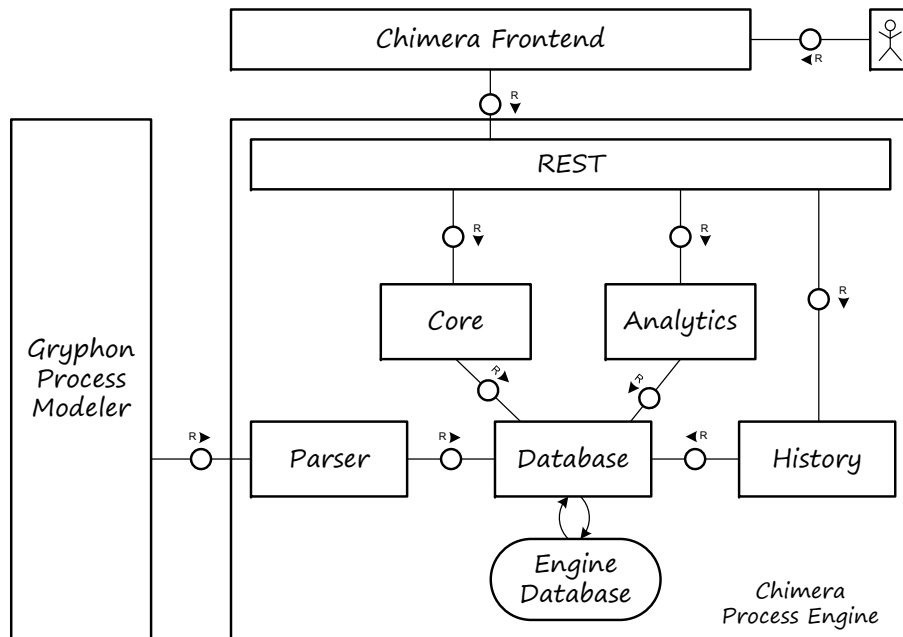


Figure 58: Architecture of the Chimera process engine .

The *History* module is responsible for storing the state changes of the activities and data objects, along with the values of the data object attributes. This produces the event log that can be used for process monitoring during the execution and process mining applications later. The analysis algorithms and methods can be accessed using the REST interface from the *analytics* module. Nevertheless, the history of a process execution can also directly be accessed from the History module and analyzed separately. The *database* is the access layer to the repository, which is MySQL²⁰ in our case.

9.1.4 Event Integration Sequence

After having introduced the components building the system architecture, this section turns to the interplay of them that realizes the integration in five steps. The steps are visualized with the architecture in Figure 54. Further, the sequence of communication is recorded in the sequence diagram shown in Figure 59.

(1) PUBLISH PROCESS MODEL. The process is modeled in Gryphon using BPMN syntax and semantics. The catching events are annotated with subscription queries. Corresponding data object lifecycles (OLC) are modeled such that the state changes modeled in the process are consistent with the state changes defined in the OLC. Once the model is completed, it is deployed in Chimera using the REST API, sent as a JSON file.

²⁰ <https://www.mysql.com/>

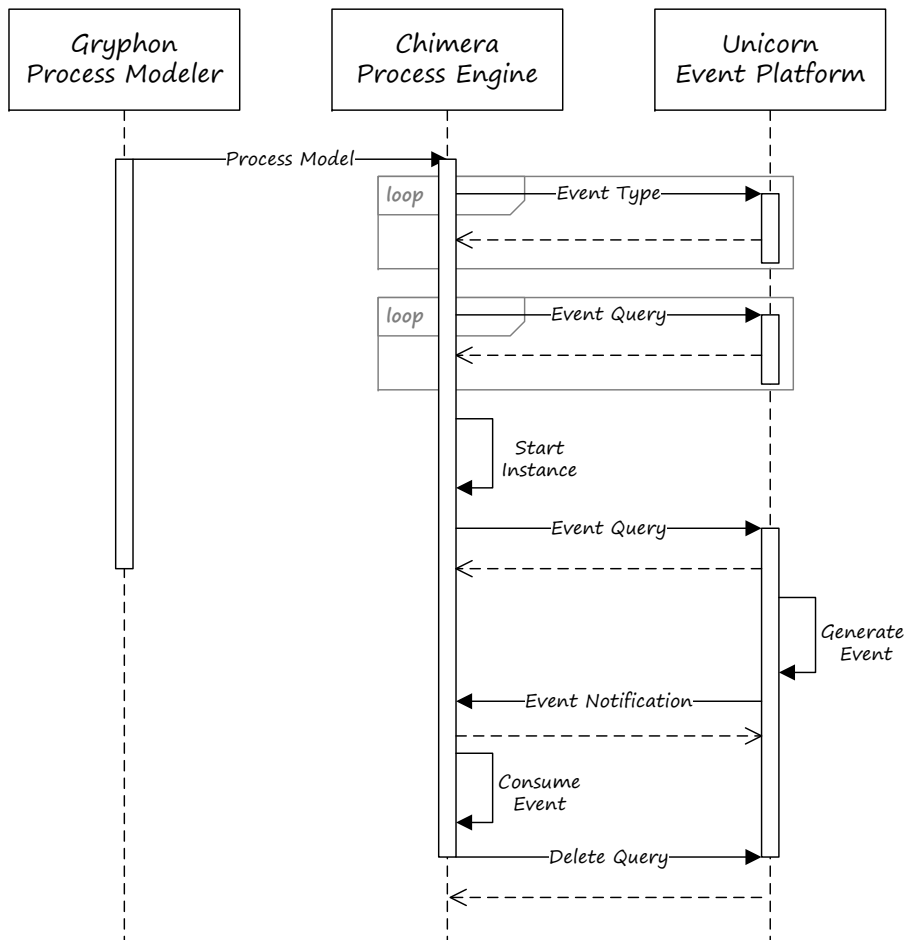


Figure 59: The sequence of communication between Gryphon, Chimera, and Unicorn for the integration of events into processes.

(2) REGISTER EVENT TYPES AND EVENT QUERIES. At deployment, the event types modeled in the process are registered in Unicorn, along with the REST callback path of Chimera. For the start event, the event query is also registered at this point. The start event remains registered until the process is deleted. A process instance can be started either by receiving an event from Unicorn or manually in Chimera. For the intermediate catching events, the queries are registered when the control flow reaches the event node during execution. These queries are unregistered from Unicorn when the event is consumed or skipped. If the abstraction rules for events are not already defined by the event engineer, this is done at this point. For each registered query, Unicorn sends back a unique ID to Chimera for further correlation.

(3) SEND EVENTS. Unicorn starts listening to the event sources and transforms the raw events according to the abstraction rules. The event sources might stream the events to the REST endpoint or JMS channel of Unicorn, or Unicorn can proactively pull the information from a

Web API periodically. Unicorn keeps evaluating abstraction rules upon receiving each event to check whether any of the rules has been satisfied completely or partially.

(4) NOTIFY. Once all the information necessary for one higher level event is available, Unicorn generates the business event. The event is then sent to the REST callback path provided by Chimera.

(5) REACT TO EVENTS. Upon receiving a start event, an instance of the corresponding process is started. For an intermediate event, the associated BPMN semantics are followed, e.g., to consume the event, to abort an ongoing activity, or to decide on the further execution branch. If the event attributes are mapped to a JSON path expression during modeling, the attribute values are stored in a data object. Once consumed, a DELETE request is sent to Unicorn that unsubscribes Chimera from further notification of the event. In case of boundary events that did not occur, the DELETE request is sent once the associated activity terminates. For start events, the unsubscription is done at process undeployment.

9.2 FLEXIBLE EVENT SUBSCRIPTION WITH BUFFERING

Having illustrated the basic integration architecture for event-process communication, this section extends the implementation framework presented above to enable flexible event handling. The implementation follows the concepts presented in [Chapter 6](#) and demonstrates the proof-of-concept of flexible subscription. Unicorn is again used as the event processing platform. Instead of Chimera, the open-source process engine Camunda²¹ has been used to show that an off-the-shelf process engine that follows BPMN semantics, can be extended with flexible event handling configurations with only minor changes. First, the necessary extensions made to BPMN are elaborated in [Section 9.2.1](#). Next, the adjustments made in Unicorn ([Section 9.2.2](#)) and Camunda ([Section 9.2.3](#)) are described.

This part of the implementation has been majorly done by *Dennis Wolf* in context of his Master thesis [136], written at the chair of Business Process Technology, supervised by the author. In the Master thesis, an extended version of the BPMN+X model published in [81] has been presented. Further, the following five event occurrence scenarios (EOS) have been identified:

1. EOS₁: The event occurs while the catch element is enabled
2. EOS₂: The event does not occur at all
3. EOS₃: The event occurs between process instantiation and the enabling of the BPMN event

Acknowledgements

²¹ <https://camunda.com/products/bpmn-engine/>

4. *EOS4*: The event occurs between process deployment and process instantiation
5. *EOS5*: The event occurs before the deployment of the process in the process engine

To accommodate all of the above scenarios Camunda has been extended with a Process Engine Plugin and the source code of Unicorn has been adapted (cf. [Figure 62](#)). The downloadable code for the extended architecture can be found on GitHub²².

9.2.1 BPMN Extension

BPMN 2.0 offers an extension mechanism to add new features while still being compliant to the standard (see *BPMN 2.0, Section 8.2.3* [94]). We formalize the extension as a BPMN+X model, a UML profile [95] defined for using the BPMN extensibility concepts *ExtensionDefinition* and *ExtensionAttributeDefinition*. The extension is developed around the addition of subscription-related attributes to the BPMN *Message* type, as shown in [Figure 60](#) using green color. Based on the additional information, subscription and unsubscription for message events are handled by the associated process engine. Extending the Message reflects the adaptations to intermediate catching message events as well as the receive tasks. If a single message is used multiple times in a process, the subscription information has to be provided only once.

As per the specification, the Message element contains an attribute *name* — the name of the message, and *itemRef* — the reference to a BPMN *ItemDefinition* specifying the Message structure. Additionally, it inherits all attributes from the BPMN *RootElement* (see [94], Section 8.2.5). All subscription information is incorporated in a model element *SubscriptionDefinition*, which is added to the *extensionDefinitions* of the Message element. The added attributes of *SubscriptionDefinition* are described in the following.

The subscription information for an event contains the event query, the platform address, and (optionally) authorization information of the CEP platform. This extension assumes that only one event engine is in use, with the result that access information can be configured in a central configuration store for the current process execution environment and not redundantly for each message. The event query, on the other hand, needs to be specified for every message and is added to the model as an extension attribute *eventQuery* of type String, which should contain the subscription query as interpretable by the CEP platform (see [Figure 56](#)). Only *eventQuery* is a mandatory parameter, all others will fall back to default values if not provided. Using the BPMN+X model and conforming to BPMN's *ExtensionDefinition* and *ExtensionAttributeDefinition*, the following formal XSD-Schema is generated.

Attribute *subscriptionTime* defines the points of subscription, i.e., when the subscription should be registered. It can take one of the following

²² <https://github.com/dennis-wolf/camunda-explicit-early-subscription>

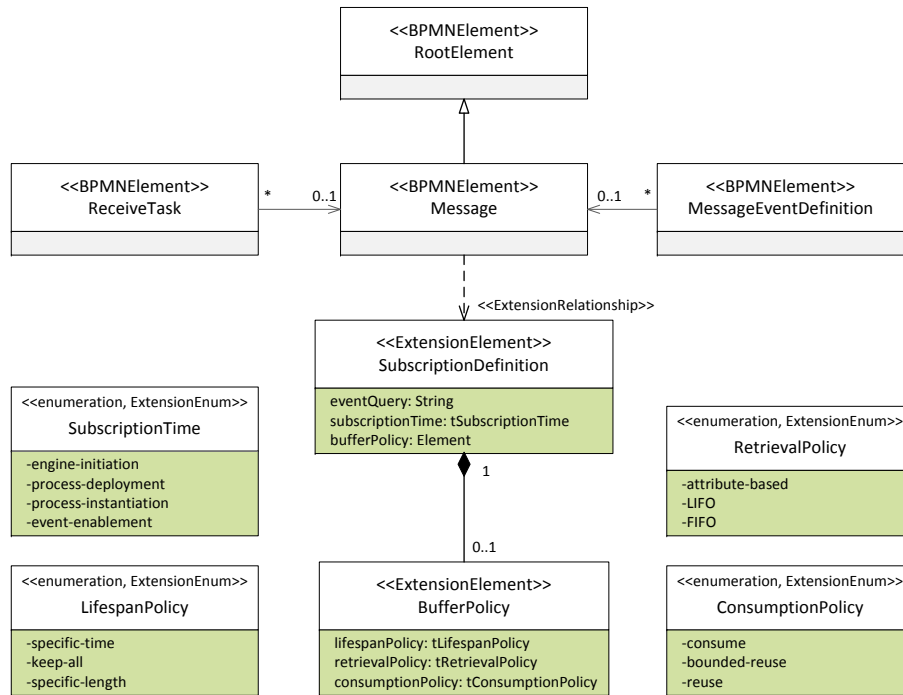


Figure 60: BPMN+X model showing extension of BPMN Message element.

values: engine-initiation, process-deployment, process-instantiation, or event-enablement. The last option is the default option, representing the standard BPMN semantics in case no other point of subscription is specified. The required time of subscription necessary for the events are defined at design time according to the use case. The subscription is then executed automatically by the process engine based on the information given in the BPMN model. Further information on the detailed execution flow is provided in [Section 9.2.3](#). SubscriptionDefinition is composed of another element (complex type) *bufferPolicy*, which influences the behavior of the related event buffers. The XML representation of the BPMN+X model is given in [Listing 11](#).

Listing 11: XML interpretation of BPMN+X model

```

<xsd:element name="SubscriptionDefinition">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element type="xsd:string" name="eventQuery"
        minOccurs="1" maxOccurs="1" />
      <xsd:element type="tSubscriptionTime" name="subscriptionTime"
        minOccurs="0" maxOccurs="1" default="event-enablement"/>
      <xsd:element name="bufferPolicy"
        minOccurs="0" maxOccurs="1">
        <xsd:complexType>
          <xsd:sequence>

```



```

    <xsd:element type="tLifespanPolicy" name="
      lifespanPolicy"
      minOccurs="0" maxOccurs="1" default="keep-all"/>
    <xsd:element type="tRetrievalPolicy" name="
      retrievalPolicy"
      minOccurs="0" maxOccurs="1" default="FIFO"/>
    <xsd:element type="tConsumptionPolicy" name="
      consumptionPolicy"
      minOccurs="0" maxOccurs="1" default="reuse"/>
  </xsd:sequence>
</xsd:complexType>
</xsd:element>
</xsd:sequence>
</xsd:complexType>
</xsd:element>

<xsd:simpleType name="tSubscriptionTime">
<xsd:restriction base="xsd:string">
<xsd:enumeration value="engine-initiation"/>
<xsd:enumeration value="process-deployment"/>
<xsd:enumeration value="process-instantiation"/>
<xsd:enumeration value="event-enablement"/>
</xsd:restriction>
</xsd:simpleType>

<xsd:simpleType name="tLifespanPolicy">
<xsd:restriction base="xsd:string">
<xsd:enumeration value="specific-length(n)"/>
<xsd:enumeration value="specific-time(ISO time-span format)"/>
<xsd:enumeration value="keep-all"/>
</xsd:restriction>
</xsd:simpleType>

<xsd:simpleType name="tRetrievalPolicy">
<xsd:restriction base="xsd:string">
<xsd:enumeration value="attribute-based(filter criteria)"/>
<xsd:enumeration value="LIFO"/>
<xsd:enumeration value="FIFO"/>
</xsd:restriction>
</xsd:simpleType>

<xsd:simpleType name="tConsumptionPolicy">
<xsd:restriction base="xsd:string">
<xsd:enumeration value="consume"/>
<xsd:enumeration value="bounded-reuse(n)"/>
<xsd:enumeration value="reuse"/>
</xsd:restriction>
</xsd:simpleType>
</xsd:schema>

```

An example process is shown in [Figure 61](#) where activity Do Something is followed by an intermediate catching event TestEvent. We write

the eventQuery as "select * from TestEvent" and choose the subscriptionTime as "process-instantiation". The XML interpretation of the SubscriptionDefinition is given in Listing 12.

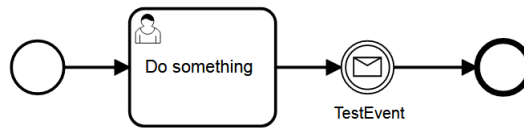


Figure 61: An example process with embedded subscription definition for the intermediate catching message event.

Listing 12: Excerpt from the XML interpretation of the BPMN process modeled in Fig. 59 showing process structure and extended elements enabling flexible subscription

```

...
<bpmn:process id="flexsub.camunda.demoprojects.simple.
  SubscribeOnInstantiation" name="SubscribeOnInstantiation"
  isExecutable="true">

% ***** process structure definition ***** %

  <bpmn:startEvent id="StartEvent_1">
    <bpmn:outgoing>SequenceFlow_0wjap3k</bpmn:outgoing>
  </bpmn:startEvent>
  <bpmn:sequenceFlow id="SequenceFlow_owjap3k"
    sourceRef="StartEvent_1" targetRef="Task_1um2wc3" />

  <bpmn:intermediateCatchEvent
    id="IntermediateThrowEvent_16k343v" name="TestEvent">
    <bpmn:incoming>SequenceFlow_1ndzqus</bpmn:incoming>
    <bpmn:outgoing>SequenceFlow_15imq89</bpmn:outgoing>
    <bpmn:messageEventDefinition messageRef="Message_1gdrmt" />
  </bpmn:intermediateCatchEvent>

  <bpmn:endEvent id="EndEvent_1n7dnmz">
    <bpmn:incoming>SequenceFlow_15imq89</bpmn:incoming>
  </bpmn:endEvent>
  <bpmn:sequenceFlow id="SequenceFlow_15imq89"
    sourceRef="IntermediateThrowEvent_16k343v"
    targetRef="EndEvent_1n7dnmz" />
  <bpmn:sequenceFlow id="SequenceFlow_1ndzqus"
    sourceRef="Task_1um2wc3"
    targetRef="IntermediateThrowEvent_16k343v" />

  <bpmn:userTask id="Task_1um2wc3" name="Do something">
    <bpmn:incoming>SequenceFlow_0wjap3k</bpmn:incoming>
    <bpmn:outgoing>SequenceFlow_1ndzqus</bpmn:outgoing>
  </bpmn:userTask>

```

```

</bpmn:process>

% ***** flexible subscription extension ***** %

<bpmn:message id="Message_1gdrckt" name="Message_OnInstantiation">
<bpmn:extensionElements>
  <flexsub:subscriptionDefinition>
    <flexsub:eventQuery>select * from TestEvent
    </flexsub:eventQuery>
    <flexsub:subscriptionTime>process-instantiation
    </flexsub:subscriptionTime>
  </flexsub:subscriptionDefinition>
</bpmn:extensionElements>
</bpmn:message>

...

```

After presenting the BPMN extension, the next sections describe the extensions of the CEP platform and the process engine needed for executing the information passed through the BPMN Message elements.

9.2.2 Unicorn Extension

The applied architecture with extensions in both Camunda and Unicorn is visualized in [Figure 62](#). The modules colored in green are the added components, whereas orange depicts modification.

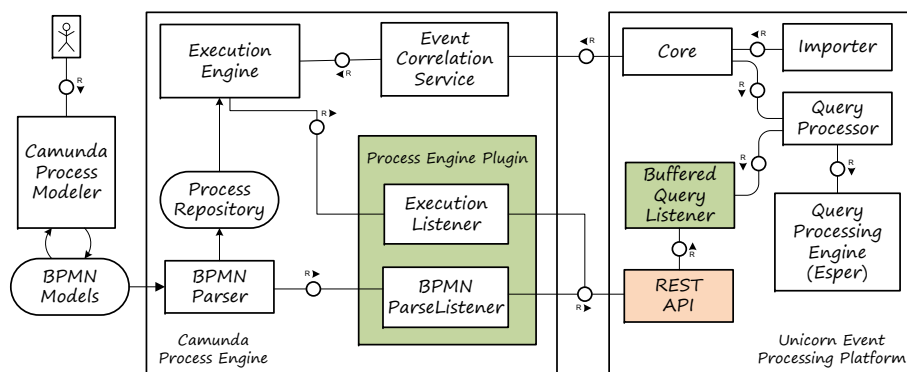


Figure 62: Extended architecture for flexible event handling.

For Unicorn, the major extension is adding the buffering functionality to allow delayed delivery of events to the process engine. The design decision to integrate the event buffer into the CEP platform has the following reasons:

- The CEP platform registers the subscriptions and notifies about the occurrences. The only modification needed was to handle the time of subscription and notification more flexibly. This is controlled by the process engine according to the extended process

specification. Thus, adding the buffer to Unicorn does not change the responsibilities of the CEP platform and the process engine. This helps the extended architecture to stay consistent with the requirement *separation of concerns*.

- The CEP platform anyway receives and stores the events. By implementing the event buffering module isolated from the process engine, we reuse the storage of the CEP platform and ensure that the performance of the process engine is not influenced.
- Having the event buffer in the CEP platform makes it accessible for both Chimera and Camunda process engines, as well as other possible event consumers.

The buffering technicalities are managed by the newly added module *Buffered Query Listener*. Esper offers the *UpdateListener* interface to enable the callback function for a new query. Unicorn implements this as *LiveQueryListener* to react on new event occurrences and registers the event query to Esper. Whenever an event matches that query, *LiveQueryListener* is notified. Originally, the query listener would notify all subscribers for that query and then drop the event. For implementing event buffering, a new class *BufferedLiveQueryListener* is created which extends the behavior of the standard query listener.

The module is built around the class *EventBuffer*. Objects of the class are managed by the *BufferManager* and represent a single buffer entity, containing events of one query. The query output from Esper is stored in a list in *EventBuffer*. The buffer behaves according to the value of its policies, which influence the items to store based on time window or count, the order in which items are retrieved from the list, and how many times an item is used. The lifespan policy, which requires that events are deleted from the buffer after a certain time, is ensured by a maintenance thread that runs from the *BufferManager* class and iterates over all *EventBuffer* objects in a specified time interval.

REST API extension

Besides the architectural modifications, the REST API of Unicorn is also extended to make use of the buffering module. So far, the Unicorn REST API is comprised of the basic functionality such as query registration, query deletion and obtaining query strings by the subscription identifier, as described in [Section 9.1](#). The additional methods introduced to the Unicorn Webservice for flexible event handling are described below. The methods use the path `<platform>/BufferedEventQuery/REST` to reach Unicorn.

REGISTER QUERY. This is used by the *BufferManager* to create a new *EventBuffer* object. The payload for the API call is a JSON object that requires an event query and optionally buffer policies. A unique identifier to detect the query and associated buffer is returned.

```
% *** Register Query *** %
POST to /BufferedEventQuery
returns queryId
Payload: JSON (eventQuery[, bufferPolicies])
```

SUBSCRIBE. This method adds a new recipient to the selected query, i.e., a new subscription is created using the `queryId` returned from the register query method. As a result, a notification is issued based on the current buffer content and whenever an event matches the query. The notification path contains the message name that supports Camunda's event correlation service to match the issued notification to the right message.

```
% *** Subscribe *** %
POST to /BufferedEventQuery/{queryId}
returns subscriptionId
Payload: JSON (notificationPath) with
notificationPath: (notificationAddress, messageName)
```

UNSUBSCRIBE. Using this method removes the specified subscription from the list of subscriptions of the selected query. The specific recipient therefore does not receive any further notification. Note that the buffer and query instance remain intact, so that other recipients can still subscribe.

```
% *** Unsubscribe *** %
DELETE to /BufferedEventQuery/{queryId}/{subscriptionId}
```

REMOVE QUERY. Finally, this method deletes the query and the associated buffer altogether.

```
% *** Remove Query *** %
DELETE to /BufferedEventQuery/{queryId}
```

9.2.3 Camunda Extension

As seen in the FMC diagram, Camunda has a similar architecture as Chimera. The BPMN process models are created using *Camunda modeler* and deployed to the process engine. The *Execution Engine* controls process instance execution, whereas the *Event Correlation Service*²³ manages receiving of events and correlating them with the correct process instance. Camunda offers the concept of *Process Engine Plugin* (PEP)²⁴ to intercept significant engine operations and introduce custom code. This is a separate software module activated by adding a plugin entry in the process engine configuration that implements the `ProcessEnginePlugin` interface of Camunda. The PEPs enable adding *Execution*

²³ <https://docs.camunda.org/manual/7.9/reference/bpmn20/events/message-events/#using-the-runtime-service-s-correlation-methods>

²⁴ <https://docs.camunda.org/manual/7.7/user-guide/process-engine/process-engine-plugins/>

*Listeners*²⁵ programmatically. These execution listeners trigger the execution of custom code during a process execution. To customize the semantics of the process specification, we implement a *BPMNParseListener* that is executed after a BPMN element is parsed, as explained in the GitHub project²⁶. The *BPMNParseListener* interface allows to react to the parsing of single elements based on their type by applying a separate method for every BPMN element available.

In essence, the subscriptions are managed automatically by the process engine triggering the execution listeners according to the subscription configuration provided in the extended message element extracted by the BPMN parse listener. An alternative way of extending Camunda could be to directly adapt the source code to integrate additional behavior to the process elements. However, implementing a PEP allows a clearer, more understandable approach to adapt the extension behavior and facilitates maintainability and reusability of the extended code. Figure 63 shows the class diagram of the PEP, elaborated in the following.

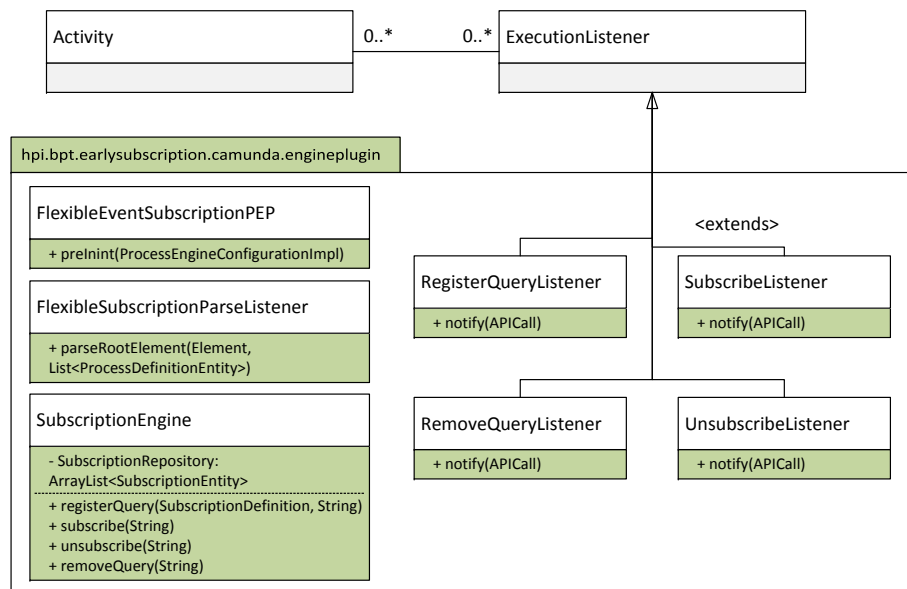


Figure 63: UML Class diagram of Camunda process engine plugin.

FLEXIBLE EVENT SUBSCRIPTION PEP. The Process Engine Plugin enables executing custom Java code at predefined points during engine execution. An entry point to the modification of the execution behavior is provided through the implementation of the *ProcessEnginePlugin* interface. It allows to intercept at three different points during the engine bootstrapping: *preInit*, *postInit*, and *postProcessEngineBuild*. We chose to provide the custom implementation for the *preInit* method.

25 <https://docs.camunda.org/manual/7.7/reference/bpmn20/custom-extensions/extension-elements/>

26 <https://github.com/camunda/camunda-bpm-examples/tree/master/process-engine-plugin/bpmn-parse-listener>

SUBSCRIPTION ENGINE. A new class *SubscriptionEngine* has been introduced that encapsulates the functionality needed to communicate with the API of the event engine, i.e., the methods for API-calls: *registerQuery*, *subscribe*, *unsubscribe*, and *removeQuery*. The class has another important functionality, namely, the *SubscriptionRepository* that handles the correlation from the engine side. This contains a list of all available query and subscription identifiers and a mapping to the related process definitions or instances. As described above, active queries and subscriptions can only be deleted using their unique identifiers. Upon issuing a subscription or registering a query, these identifiers are stored in the repository until the removal call is executed. Based on the repository information, the *SubscriptionEngine* matches the subscription identifiers and execute the remove and unsubscribe operations on the event engine.

EXECUTION LISTENERS. In total, four execution listeners are added, one for each API-call, which are attached to the process nodes (Activity) through the BPMN parse listener. Execution listeners are triggered by the internal transition events of Camunda, such as *terminate* or *begin* of an activity. Each of the listeners is a separate class implementing the *ExecutionListener* interface. There is only one method *notify* included in the listeners, which is called by the engine when the corresponding transition event fires. Using the method, each listener class issues an API-call, as defined by the *SubscriptionManager*.

FLEXIBLE SUBSCRIPTION PARSE LISTENER. The *BPMNParseListener* extracts all relevant xml elements from the deployed model, based on the element names. This results in a list of intermediate and boundary message catch events and receive-tasks, along with the messages they reference. Depending on the specified subscription time, *ExecutionListeners* are added at the transitional events during process execution. These transitional events are determined according to the points of subscription specification, as defined in [Section 7.2.2](#). For example, if the specified subscription time is at process deployment (POS₃), the provided query is registered immediately using the *SubscriptionManager*. On the other hand, if the subscription is set at process instantiation (POS₂), the following is executed:

- An instance of *RegisterQueryListener* is added to the start event of the process. The *subscriptionDefinition* is provided to that listener.
- The *SubscribeListener* is attached to the start event of the xml node representation of the intermediate message event.
- To the same node, *UnsubscribeListener* and *RemoveQueryListener* are attached to be triggered by the termination of the node.

9.3 SUMMARY

The proof-of-concept implementation shows the feasibility of both the basic concepts of event integration into business processes and the advanced event handling possibilities required for flexible subscription. The architecture consists of a BPMN modeler, an event processing platform, and a process engine. The design decisions and implementation techniques are mostly independent of a particular choice of engine, component, or platform. This is shown by using two different process engines, one built for academic purposes and the other one being used in real-world BPM solutions. Nevertheless, system specific adaptations might be required in case different components are used.

The basic architecture supports separation of concerns by enabling the CEP platform to handle the event processing logic and giving the process execution control to the process engine. The flexible subscription follows separation of logic by splitting the buffer maintenance and subscription handling between the CEP platform and the process engine, respectively. Throughout, intermediate catching message events are used to represent communication received from the environment. The BPMN extension of a Message element provides additional subscription definitions needed to switch between points of subscription and buffer policies. This enables implementing flexible subscription model while being grounded in BPMN semantics.

Altogether, the enhanced CEP platform and business process engine enable the automatic handling of information provided through the BPMN extension for flexible event subscription. The event engine exposes functionality for buffered event handling which is accessed through execution listeners during the process execution in Camunda. Given these extended features, process designers can conveniently incorporate subscription information for external message events in their executable BPMN models.

CONCLUSIONS

*“In literature and in life we ultimately pursue, not conclusions, but beginnings.” – the wise words said by Sam Tanenhaus in his book *Literature Unbound*¹ are applicable to research too. With this chapter, this thesis is now coming to an end, which opens the beginning for several other research directions. The first part of the chapter, [Section 10.1](#), summarizes the results of the thesis. Later in [Section 10.2](#) the limitations and future research possibilities are discussed.*

10.1 SUMMARY OF THESIS

The work in this thesis started with the notion of integrating the relevant contextual information into business processes to make the process execution aware of environmental occurrences and enable the process to react to a situation in a timely manner. To this end, the concepts from complex event processing area seemed to be beneficial for event abstraction hierarchy that was needed to extract the right level of granularity of the information to be consumed by the processes. Therefore, the system architecture was set up to integrate CEP and BPM with the means of communication between a process modeler, a process engine, and an event platform. While applying the basic interaction framework on different real life usecases, the research gap of having a clear and flexible subscription management model was evident. The research thus investigated the issues related to event subscription from a business process execution perspective and eventually a formal event handling model was developed. In the following, the detailed results of the thesis are listed:

- *Requirements for integrating real-world events into business processes.* Based on literature review, examples of real-world usecases and extensive discussion sessions with domain experts from both academia and industry, a list of requirements for integrating external events into business process execution is presented in [Chapter 5](#). These requirements cover conceptual as well as technical aspects of the integration. Further, the requirement for having flexible points of subscription and unsubscription are supported using motivation examples from the logistics domain.
- *Integrated framework enabling event-process communication.* The integrated framework satisfied the requirements and enabled a smooth

¹ <https://www.goodreads.com/book/show/1070640>

communication between the process engine and the event platform (ch. [Chapter 5](#)). Assigning the responsibility for controlling the business logic to the process engine and event processing techniques to the CEP platform establishes separation of concern and facilitates reuse of events. The event hierarchies are hidden from the process view and are dealt with by the event abstraction rules. The higher-level event is then mapped to the business event needed for the execution. The process engine subscribes to the start events at deployment and to the intermediate events at event enablement. The CEP platform connects to the event sources, operates on the event streams according to the event query, and notifies the process engine when a matching event is detected. The process engine then follows the process specification to react to the event.

- *Subscription management facilitating flexible event handling.* The subscription management is further detailed in the flexible event handling model presented in [Chapter 6](#). Along the process execution timeline, execution states are specified when a subscription and an unsubscription can be issued. While unsubscription is optional, subscription is mandatory to receive notification about an event occurrence. Depending on the use of event structure in a process, specific semantics are assigned to these points of (un)-subscription. For instance, boundary events should only be subscribed once the associated activity has started. An event in the normal flow, on the contrary, can be subscribed to at enablement, at process instantiation, at process deployment, or at engine initiation; depending on the availability of information needed for subscription. An unsubscription can be done at event consumption, at semantic resolution, at process undeployment, or at engine termination. Semantic resolution in this context means coming to the point during process execution when the event becomes obsolete for that particular instance. The definition of semantic resolution differs for boundary event, exclusive event, and racing events. The proof-of-concept implementation provides basic technicalities to enable flexible subscription by extending BPMN notation, Unicorn event platform, and Camunda process engine.
- *Event buffer complementing early subscription.* Early subscription of events demand a temporary storage for the events to make it available till the process is ready to consume. The concept of an event buffer is proposed in [Chapter 6](#) to address this. The buffer comes with a set of buffer policies that offer alternative configurations for the time and quantity based lifespan of the events, the most suitable event to consume in case several occurrences of the same event type are available, and the reusability of an event information. In the prototype implementation discussed in [Chap-](#)

ter 9, the buffering functionalities are added to the event platform, whereas the control of flexible subscription remains with the process engine.

- *Formal translation of the event handling model.* Finally, the formal model assigns clear semantics to the event handling concepts in Chapter 7. The points of subscription and points of unsubscription are mapped to Petri net modules according to the event constructs. The buffer policies are expressed formally as coloured Petri nets. This formal translation makes formal process verification methods applicable to the process execution enriched with event handling configurations. Trace analysis to verify correct process execution and reachability analysis to find allowed environmental behaviors are presented as examples of such applications.

10.2 LIMITATIONS AND FUTURE RESEARCH

We claim our work to be a significant contribution in the current context of IoT and distributed business process execution. Nevertheless, the work presented in the thesis has certain limitations that need to be addressed in future research related to this topic, as discussed below.

- *Roles of process designer & event engineer.* First of all, the whole concept of the thesis is based on the assumption that the process execution is aware of the event sources and event abstraction rules behind the business events relevant for the processes. This is possible only when there is clear distribution of responsibilities in an organization, such that there are dedicated process designers and event engineers who will decide on the configurations for event subscription from a business process perspective and an event processing perspective, respectively. In practice, this is often not the case. Also, the thesis does not focus on the specification about who is responsible for which design decision(s). In those situations, it might be difficult to exploit the flexibility offered by the event handling model to the fullest. However, the event handling model gives clear semantics for the points of (un)-subscription and the buffer policies. Assuming the process participants have enough domain knowledge, these notions are easy to grasp and apply to a certain usecase.
- *Advanced event types.* The integration architecture with Gryphon, Chimera, and Unicorn implements the basic use of events as process artifacts such as message and timer events, boundary events, and event-based gateways. More advanced event constructs like signal, compensation, parallel event types are not implemented since that has been considered out of scope for this thesis. The external events are always represented as catching intermediate

events (and start events). We argue that most of the business scenarios can be captured using message events in a normal flow, in an exceptional flow (boundary event), and after a decision point (event-based gateway) [88]. Yet, event types such as error, escalation, and signal events offer specific event handling semantics that are most appropriate to capture an event occurrence in certain situations. In future, the subscription configurations should be extended for all event types and their usage in a process.

- *Complex process structures.* So far, processes with loops are not considered in our work. The loops make event correlation highly complicated, as described in the following example. For an online payment procedure, the customer is given the option to update the card details if the payment is not successful at the first place. This is done with a simple process taking the customer details as a receiving message event. If the buffer policy is set to LIFO and reuse for the event, the first execution of the loop can consume the latest card information residing in the buffer. However, the next execution of the loop happens only when the payment is failed. Having an event in buffer here leads to the consumption of the same information, though an updated card details is intended. Situations like that need to be investigated further and the event handling configurations might be extended to accommodate versatile scenarios.
- *Smart Buffer Manager.* Let us think of a scenario where we know that an event is published in regular interval of 30 min. The process reaches the point of consuming the event when the 30 min window is towards the end, e.g., the last event has been published 28 min before and the next event occurrence is expected in 2 min. In this case, waiting for 2 more minutes for the updated information might make more sense than having the last event available in the buffer. A smart buffer manager reflects on the idea of having those context information available to the buffer and thus suggesting the process participant the alternative options to consume events.
- *Further applications.* In [Chapter 8](#), we documented two applications based on the formal notations of the event handling model. Some possible extensions of those applications are also hinted in the discussion sections at the end of the chapter. However, the application concepts are not tested with real data yet. For instance, building up a concrete adapter based on the corresponding paths is an interesting future work. Using a set of processes to implement the application ideas for the adapter synthesis and trace analysis is in our agenda for extending the work.

- *User-friendly visualization.* Lastly, the current implementation for flexible event subscription is developed to run in the background and does not provide any visual cockpit yet. Though the subscription configurations are part of the XML description of the processes, they are not offered to the process designers in a user-friendly way. An idea to address this could be to design configuration panels for intermediate events where drop-down lists for point of subscription, point of unsubscription, and buffer policies are added. This approach will also be helpful to enforce the semantic interdependencies between the event handling notions discussed in [Section 6.3.4](#).

BIBLIOGRAPHY

- [1] Wil M. P. Aalst. *Process Mining: Discovery, Conformance and Enhancement of Business Processes*, volume 136. 01 2011. ISBN 978-3-642-19344-6. doi: 10.1007/978-3-642-19345-3. (Cited on pages 14, 27, and 49.)
- [2] Activiti. Activiti BPM Platform. <https://www.activiti.org/>. (Cited on page 47.)
- [3] Nabil R. Adam, Vijayalakshmi Atluri, and Wei-Kuang Huang. Modeling and analysis of workflows using petri nets. *Journal of Intelligent Information Systems*, 10(2):131–158, Mar 1998. ISSN 1573-7675. doi: 10.1023/A:1008656726700. URL <https://doi.org/10.1023/A:1008656726700>. (Cited on page 107.)
- [4] Marcos K. Aguilera, Robert E. Strom, Daniel C. Sturman, Mark Astley, and Tushar D. Chandra. Matching events in a content-based subscription system. In *Proceedings of the Eighteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '99, pages 53–61, New York, NY, USA, 1999. ACM. ISBN 1-58113-099-6. doi: 10.1145/301308.301326. URL <http://doi.acm.org/10.1145/301308.301326>. (Cited on page 48.)
- [5] A. Akbar, F. Carrez, K. Moessner, and A. Zoha. Predicting complex events for pro-active iot applications. In *2015 IEEE 2nd World Forum on Internet of Things (WF-IoT)*, pages 327–332, Dec 2015. doi: 10.1109/WF-IoT.2015.7389075. (Cited on pages 47 and 53.)
- [6] A. Al-Fuqaha, M. Guizani, M. Mohammadi, M. Aledhari, and M. Ayyash. Internet of things: A survey on enabling technologies, protocols, and applications. *IEEE Communications Surveys Tutorials*, 17(4): 2347–2376, Fourthquarter 2015. ISSN 1553-877X. doi: 10.1109/COMST.2015.2444095. (Cited on pages 42 and 49.)
- [7] Stefan Appel, Sebastian Frischbier, Tobias Freudenreich, and Alejandro Buchmann. Event Stream Processing Units in Business Processes. In *BPM*, pages 187–202. Springer, Berlin, Heidelberg, 2013. doi: 10.1007/978-3-642-40176-3_15. (Cited on pages 44, 46, and 49.)
- [8] Arvind Arasu, Shivnath Babu, and Jennifer Widom. The cql continuous query language: Semantic foundations and query execution. *The VLDB Journal*, 15(2):121–142, June 2006. ISSN 1066-8888. doi: 10.1007/s00778-004-0147-z. URL <http://dx.doi.org/10.1007/s00778-004-0147-z>. (Cited on page 49.)
- [9] Kevin Ashton. That ‘internet of things’ thing. 2009. URL <http://www.rfidjournal.com/articles/view?4986>. (Cited on page 3.)
- [10] Michael Backmann, Anne Baumgrass, Nico Herzberg, Andreas Meyer, and Mathias Weske. Model-Driven Event Query Generation for Business Process Monitoring. In *Service-Oriented Computing – ICSOC 2013 Workshops*, pages 406–418. Springer, Cham, December 2013. doi: 10.1007/978-3-319-06859-6_36. (Cited on pages 15, 44, and 49.)

- [11] Thomas Baier, Claudio Di Ciccio, Jan Mendling, and Mathias Weske. Matching events and activities by integrating behavioral aspects and label analysis. *Software & Systems Modeling*, 17(2):573–598, May 2018. ISSN 1619-1374. doi: 10.1007/s10270-017-0603-z. URL <https://doi.org/10.1007/s10270-017-0603-z>. (Cited on page 32.)
- [12] Roberto Baldoni, Leonardo Querzoni, and Antonino Virgillito. Distributed event routing in publish/subscribe communication systems: a survey. Technical report, 2005. (Cited on page 48.)
- [13] A. Barros, G. Decker, and A. Grosskopf. Complex events in business processes. In *BIS*. Springer, 2007. (Cited on pages 44, 45, 47, 49, 56, 69, and 70.)
- [14] Rémi Bastide, Ousmane Sy, David Navarre, and Philippe Palanque. A formal specification of the corba event service. In Scott F. Smith and Carolyn L. Talcott, editors, *Formal Methods for Open Object-Based Distributed Systems IV*, pages 371–395, Boston, MA, 2000. Springer US. ISBN 978-0-387-35520-7. (Cited on page 48.)
- [15] Kimon Batoulis, Stephan Haarmann, and Mathias Weske. Various notions of soundness for decision-aware business processes. In Heinrich C. Mayr, Giancarlo Guizzardi, Hui Ma, and Oscar Pastor, editors, *Conceptual Modeling*, pages 403–418, Cham, 2017. Springer International Publishing. ISBN 978-3-319-69904-2. (Cited on page 107.)
- [16] A. Baumgrass, N. Herzberg, A. Meyer, and M. Weske. BPMN Extension for Business Process Monitoring. In *EMISA, Lecture Notes in Informatics. Gesellschaft fuer Informatik (GI)*, 2014. (Cited on pages 32 and 61.)
- [17] Anne Baumgraß, Mirela Botezatu, Claudio Di Ciccio, Remco M. Dijkman, Paul Grefen, Marcin Hewelt, Jan Mendling, Andreas Meyer, Shaya Pourmirza, and Hagen Völzer. Towards a methodology for the engineering of event-driven process applications. In *Business Process Management Workshops - BPM 2015, 13th International Workshops, Innsbruck, Austria, August 31 - September 3, 2015, Revised Papers*, pages 501–514, 2015. doi: 10.1007/978-3-319-42887-1_40. URL https://doi.org/10.1007/978-3-319-42887-1_40. (Cited on pages 46, 47, and 49.)
- [18] 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.*, pages 75–79, 2015. URL <http://ceur-ws.org/Vol-1418/paper16.pdf>. (Cited on pages 46, 54, and 116.)
- [19] Izak Benbasat and Robert W. Zmud. Empirical research in information systems: The practice of relevance. *MIS Q.*, 23(1):3–16, March 1999. ISSN 0276-7783. doi: 10.2307/249403. URL <http://dx.doi.org/10.2307/249403>. (Cited on pages xiii and 8.)
- [20] 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)*, volume 1789 of *CEUR Workshop Proceedings*, pages 67–72. CEUR-WS.org, 2016. URL <http://ceur-ws.org/Vol-1789/bpm-demo-2016-paper13.pdf>. (Cited on pages 54 and 115.)
- [21] L. Brenna, A. Demers, J. Gehrke, M. Hong, J. Ossher, B. Panda, M. Riedewald, M. Thatte, and W. White. Cayuga: a high-performance event processing engine. In *SIGMODConf2007*, pages 1100–1102. ACM, 2007. (Cited on page 48.)
- [22] Jan vom Brocke and Jan Mendling. *Frameworks for Business Process Management: A Taxonomy for Business Process Management Cases*, pages 1–17. 01 2018. ISBN 978-3-319-58306-8. doi: 10.1007/978-3-319-58307-5_1. (Cited on page 13.)
- [23] Cristina Cabanillas, Claudio Di Ciccio, Jan Mendling, and Anne Baumgrass. Predictive Task Monitoring for Business Processes. In Shazia Sadiq, Pnina Soffer, and Hagen Völzer, editors, *BPM*, number 8659, pages 424–432. Springer International Publishing, September 2014. ISBN 978-3-319-10171-2 978-3-319-10172-9. doi: 10.1007/978-3-319-10172-9_31. (Cited on pages 45 and 49.)
- [24] Camunda. camunda BPM Platform. <https://www.camunda.org/>. (Cited on pages 5 and 47.)
- [25] Filip Caron, Jan Vanthienen, and Bart Baesens. Comprehensive rule-based compliance checking and risk management with process mining. *Decision Support Systems*, 54(3):1357 – 1369, 2013. ISSN 0167-9236. doi: <https://doi.org/10.1016/j.dss.2012.12.012>. URL <http://www.sciencedirect.com/science/article/pii/S0167923612003788>. (Cited on page 14.)
- [26] Federico Chesani, Paola Mello, Marco Montali, Fabrizio Riguzzi, Maurizio Sebastianis, and Sergio Storari. Checking compliance of execution traces to business rules. In Danilo Ardagna, Massimo Mecella, and Jian Yang, editors, *Business Process Management Workshops*, pages 134–145, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg. ISBN 978-3-642-00328-8. (Cited on page 101.)
- [27] Michele Chinosi and Alberto Trombetta. Bpmn: An introduction to the standard. *Computer Standards Interfaces*, 34(1):124 – 134, 2012. ISSN 0920-5489. doi: <https://doi.org/10.1016/j.csi.2011.06.002>. URL <http://www.sciencedirect.com/science/article/pii/S0920548911000766>. (Cited on page 4.)
- [28] G. Chiola, M. A. Marsan, G. Balbo, and G. Conte. Generalized stochastic petri nets: a definition at the net level and its implications. *IEEE Transactions on Software Engineering*, 19(2):89–107, Feb 1993. doi: 10.1109/32.214828. (Cited on page 84.)
- [29] Mariano Cilia, Alejandro Buchmann, and Tu-Darmstadt K Moody. An active functionality service for open distributed heterogeneous environments. 04 2019. (Cited on page 48.)

- [30] Carlo Combi, Barbara Oliboni, and Francesca Zerbatò. Modeling and handling duration constraints in BPMN 2.0. In *Proceedings of the Symposium on Applied Computing, SAC 2017, Marrakech, Morocco, April 3-7, 2017*, pages 727–734, 2017. doi: 10.1145/3019612.3019618. URL <https://doi.org/10.1145/3019612.3019618>. (Cited on page 50.)
- [31] G. Cugola, E. Di Nitto, and A. Fuggetta. Exploiting an event-based infrastructure to develop complex distributed systems. In *Proceedings of the 20th International Conference on Software Engineering*, pages 261–270, April 1998. doi: 10.1109/ICSE.1998.671135. (Cited on page 48.)
- [32] Gianpaolo Cugola and Alessandro Margara. TESLA: a formally defined event specification language. In *Proceedings of the Fourth ACM International Conference on Distributed Event-Based Systems, DEBS 2010, Cambridge, United Kingdom, July 12-15, 2010*, pages 50–61, 2010. doi: 10.1145/1827418.1827427. URL <https://doi.org/10.1145/1827418.1827427>. (Cited on page 49.)
- [33] Michael Daum, Manuel Götz, and Jörg Domaschka. Integrating cep and bpm: How cep realizes functional requirements of bpm applications (industry article). In *DEBS*, pages 157–166. ACM, 2012. (Cited on page 47.)
- [34] Alexandre de Castro Alves. New event-processing design patterns using cep. In Stefanie Rinderle-Ma, Shazia Sadiq, and Frank Leymann, editors, *Business Process Management Workshops*, pages 359–368, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg. ISBN 978-3-642-12186-9. (Cited on page 38.)
- [35] H de Man. Case Management: A Review of Modeling Approaches. *BPTrends*, pages 1–17, January 2009. (Cited on page 50.)
- [36] Gero Decker and Jan Mendling. Process instantiation. *Data Knowledge Engineering*, 68(9):777–792, 2009. doi: 10.1016/j.datak.2009.02.013. (Cited on page 48.)
- [37] Remco M. Dijkman, Marlon Dumas, and Chun Ouyang. Semantics and analysis of business process models in BPMN. *Information and Software Technology*, 50(12):1281 – 1294, 2008. ISSN 0950-5849. doi: <https://doi.org/10.1016/j.infsof.2008.02.006>. (Cited on pages 10, 22, 24, 83, 87, and 97.)
- [38] Remco M. Dijkman, B. Sprenkels, T. Peeters, and A. Janssen. Business models for the internet of things. *Int J. Information Management*, 35(6): 672–678, 2015. doi: 10.1016/j.ijinfomgt.2015.07.008. URL <https://doi.org/10.1016/j.ijinfomgt.2015.07.008>. (Cited on pages 42 and 49.)
- [39] Remco M. Dijkman, Geoffrey van IJzendoorn, Oktay Türetken, and Meint de Vries. Exceptions in business processes in relation to operational performance. *CoRR*, abs/1706.08255, 2017. URL <http://arxiv.org/abs/1706.08255>. (Cited on page 43.)
- [40] Marlon Dumas and Matthias Weidlich. *Business Process Analytics*, pages 1–8. Springer International Publishing, Cham, 2018. ISBN 978-3-319-63962-8. doi: 10.1007/978-3-319-63962-8_85-1. URL https://doi.org/10.1007/978-3-319-63962-8_85-1. (Cited on pages 42 and 49.)

- [41] Marlon Dumas, Wil M. van der Aalst, and Arthur H. ter Hofstede. *Process Aware Information Systems: Bridging People and Software Through Process Technology*. Wiley-Interscience, New York, NY, USA, 2005. ISBN 0471663069. (Cited on page 14.)
- [42] Marlon Dumas, Marcello La Rosa, Jan Mendling, and Hajo A. Reijers. *Fundamentals of Business Process Management*. Springer, 2013. ISBN 978-3-642-33142-8. doi: 10.1007/978-3-642-33143-5. URL <http://dx.doi.org/10.1007/978-3-642-33143-5>. (Cited on page 13.)
- [43] Marius Eichenberg. *Event-Based Monitoring of Time Constraint Violations*. Master thesis, Hasso Plattner Institute, 2016. (Cited on page 15.)
- [44] EsperTech. Esper Event Processing Language EPL. <http://www.espertech.com/esper/release-5.4.0/esper-reference/html/>. (Cited on pages 49 and 60.)
- [45] Antonio Estruch and José Antonio Heredia Álvaro. Event-Driven Manufacturing Process Management Approach. In *BPM*, pages 120–133. Springer, Berlin, Heidelberg, September 2012. doi: 10.1007/978-3-642-32885-5_9. (Cited on pages 44, 45, and 49.)
- [46] Opher Etzion and Peter Niblett. *Event Processing in Action*. Manning Publications, 2010. ISBN 9781935182214. (Cited on pages 3, 27, 29, 31, 53, and 79.)
- [47] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Comput. Surv.*, 35(2):114–131, June 2003. ISSN 0360-0300. doi: 10.1145/857076.857078. URL <http://doi.acm.org/10.1145/857076.857078>. (Cited on page 48.)
- [48] David Eysers, Avigdor Gal, Hans-Arno Jacobsen, and Matthias Weidlich. Integrating Process-Oriented and Event-Based Systems (Dagstuhl Seminar 16341). *Dagstuhl Reports*, 6(8):21–64, 2017. ISSN 2192-5283. doi: 10.4230/DagRep.6.8.21. URL <http://drops.dagstuhl.de/opus/volltexte/2017/6910>. (Cited on page 69.)
- [49] Dirk Fahland and Christian Gierds. Analyzing and completing middleware designs for enterprise integration using coloured petri nets. In *Advanced Information Systems Engineering - 25th International Conference, CAiSE 2013, Valencia, Spain, June 17-21, 2013. Proceedings*, pages 400–416, 2013. doi: 10.1007/978-3-642-38709-8_26. URL http://dx.doi.org/10.1007/978-3-642-38709-8_26. (Cited on page 48.)
- [50] Mohammad Ali Fardbastani, Farshad Allahdadi, and Mohsen Sharifi. Business process monitoring via decentralized complex event processing. *Enterprise Information Systems*, 12:1–28, 09 2018. doi: 10.1080/17517575.2018.1522453. (Cited on pages 47 and 49.)
- [51] J. Friedenstab, C. Janiesch, M. Matzner, and O. Muller. Extending bpmn for business activity monitoring. In *2012 45th Hawaii International Conference on System Sciences*, pages 4158–4167, Jan 2012. doi: 10.1109/HICSS.2012.276. (Cited on pages 14 and 15.)
- [52] A. Gal, A. Senderovich, and M. Weidlich. Online temporal analysis of complex systems using iot data sensing. In *2018 IEEE 34th International*

- Conference on Data Engineering (ICDE)*, pages 1727–1730, April 2018. doi: 10.1109/ICDE.2018.00224. (Cited on page 15.)
- [53] Samuel Greengard. *The internet of things*. MIT Press, 2015. (Cited on pages 42 and 49.)
- [54] Christian W. Günther and Wil M. P. van der Aalst. Mining activity clusters from low-level event logs. 2006. (Cited on page 32.)
- [55] 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.*, pages 110–114, 2015. URL <http://ceur-ws.org/Vol-1418/paper23.pdf>. (Cited on pages 54 and 119.)
- [56] N. Herzberg, A. Meyer, and M. Weske. An event processing platform for business process management. In *EDOC*. IEEE, 2013. (Cited on pages 3, 45, 49, and 58.)
- [57] Marcin Hewelt and Mathias Weske. A hybrid approach for flexible case modeling and execution. In *BPM*, 2016. (Cited on pages 50 and 119.)
- [58] Annika Hinze and Alejandro P. Buchmann. *Principles and applications of distributed event-based systems*. Hershey, PA : Information Science Reference, 2010. (Cited on pages 31 and 47.)
- [59] Richard Hull, Vishal S. Batra, Yi-Min Chen, Alin Deutsch, Fenno F. Terry Heath III, and Victor Vianu. Towards a shared ledger business collaboration language based on data-aware processes. In Quan Z. Sheng, Eleni Stroulia, Samir Tata, and Sami Bhiri, editors, *Service-Oriented Computing*, pages 18–36, Cham, 2016. Springer International Publishing. ISBN 978-3-319-46295-0. (Cited on page 15.)
- [60] Hans-Arno Jacobsen, Vinod Muthusamy, and Guoli Li. The PADRES Event Processing Network: Uniform Querying of Past and Future Events. *it - Information Technology*, 51(5):250–260, May 2009. (Cited on pages 48 and 49.)
- [61] Christian Janiesch, Agnes Koschmider, Massimo Mecella, Barbara Weber, Andrea Burattin, Claudio Di Ciccio, Avigdor Gal, Udo Kanningesser, Felix Mannhardt, Jan Mendling, Andreas Oberweis, Manfred Reichert, Stefanie Rinderle-Ma, WenZhan Song, Jianwen Su, Victoria Torres, Matthias Weidlich, Mathias Weske, and Liang Zhang. The internet-of-things meets business process management: Mutual benefits and challenges. 09 2017. (Cited on pages 43 and 50.)
- [62] Kurt Jensen and Lars Kristensen. *Coloured Petri Nets: Modelling and Validation of Concurrent Systems*. 01 2009. ISBN 978-3-642-00283-0. doi: 10.1007/b95112. (Cited on page 97.)
- [63] Kurt Jensen and Lars Michael Kristensen. *Coloured Petri Nets - Modelling and Validation of Concurrent Systems*. Springer, 2009. ISBN 978-3-642-00283-0. doi: 10.1007/b95112. URL <http://dx.doi.org/10.1007/b95112>. (Cited on pages 21 and 22.)

- [64] John Jeston and Johan Nelis. Business process management: Practical guidelines to successful implementations. 01 2008. (Cited on page 13.)
- [65] S. Kaisler, F. Armour, J. A. Espinosa, and W. Money. Big data: Issues and challenges moving forward. In *2013 46th Hawaii International Conference on System Sciences*, pages 995–1004, Jan 2013. doi: 10.1109/HICSS.2013.645. (Cited on page 3.)
- [66] Kathrin Kirchner, Nico Herzberg, Andreas Rogge-Solti, and Mathias Weske. Embedding Conformance Checking in a Process Intelligence System in Hospital Environments. In *ProHealth/KR4HC*, LNCS, pages 126–139. Springer, 2012. (Cited on page 14.)
- [67] Julian Krumeich, Benjamin L Weis, Dirk Werth, and Peter Loos. Event-driven business process management: where are we now?: A comprehensive synthesis and analysis of literature. *Business Proc. Manag. Journal*, 20:615–633, 2014. (Cited on pages 32 and 44.)
- [68] Steffen Kunz, Tobias Fickinger, Johannes Prescher, and Klaus Spengler. Managing complex event processes with business process modeling notation. In Jan Mendling, Matthias Weidlich, and Mathias Weske, editors, *Business Process Modeling Notation*, pages 78–90, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg. ISBN 978-3-642-16298-5. (Cited on pages 44 and 49.)
- [69] Matthias Kunze and Mathias Weske. *Behavioural Models - From Modelling Finite Automata to Analysing Business Processes*. Springer, 2016. ISBN 978-3-319-44958-6. doi: 10.1007/978-3-319-44960-9. (Cited on pages 13 and 20.)
- [70] Andreas Lanz, Manfred Reichert, and Peter Dadam. Robust and flexible error handling in the aristaflow bpm suite. In *CAiSE Forum 2010*, volume 72 of *LNBP*, pages 174–189. Springer, 2011. (Cited on page 50.)
- [71] Guoli Li, Vinod Muthusamy, and Hans-Arno Jacobsen. A distributed service-oriented architecture for business process execution. *ACM Transactions on the Web (TWEB)*, 4(1):2, 2010. (Cited on page 47.)
- [72] Niels Lohmann. A feature-complete petri net semantics for WS-BPEL 2.0. In *Web Services and Formal Methods, 4th International Workshop, WS-FM 2007, Brisbane, Australia, September 28-29, 2007. Proceedings*, pages 77–91, 2007. doi: 10.1007/978-3-540-79230-7_6. URL http://dx.doi.org/10.1007/978-3-540-79230-7_6. (Cited on pages 4 and 83.)
- [73] Niels Lohmann, Eric Verbeek, and Remco Dijkman. *Petri Net Transformations for Business Processes – A Survey*, pages 46–63. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009. ISBN 978-3-642-00899-3. doi: 10.1007/978-3-642-00899-3_3. URL https://doi.org/10.1007/978-3-642-00899-3_3. (Cited on page 83.)
- [74] David Luckham. *Event Processing for Business: Organizing the Real-Time Enterprise*. 01 2012. (Cited on pages 3 and 32.)
- [75] David C. Luckham. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Addison-Wesley, 2010. ISBN 0201727897. (Cited on pages 18 and 56.)

- [76] Linh Thao Ly, Stefanie Rinderle, and Peter Dadam. Semantic correctness in adaptive process management systems. In Schahram Dustdar, José Luiz Fiadeiro, and Amit P. Sheth, editors, *Business Process Management*, pages 193–208, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg. ISBN 978-3-540-38903-3. (Cited on page 101.)
- [77] Alexander Lübbe. *Tangible Business Process Modeling*. Dissertation, Universität Potsdam, 2011. (Cited on page 13.)
- [78] Sankalita Mandal. Events in BPMN: the racing events dilemma. In *Proceedings of the 9th Central European Workshop on Services and their Composition (ZEUS)*, volume 1826 of *CEUR Workshop Proceedings*, pages 23–30. CEUR-WS.org, 2017. URL <http://ceur-ws.org/Vol-1826/paper5.pdf>. (Cited on pages 7, 50, and 70.)
- [79] Sankalita Mandal and Mathias Weske. A flexible event handling model for business process enactment. In *22nd IEEE International Enterprise Distributed Object Computing Conference, EDOC*, pages 68–74. IEEE Computer Society, 2018. doi: 10.1109/EDOC.2018.00019. URL <https://doi.org/10.1109/EDOC.2018.00019>. (Cited on pages 62, 65, 69, 83, and 101.)
- [80] Sankalita Mandal, Marcin Hewelt, and Mathias Weske. A framework for integrating real-world events and business processes in an iot environment. In *On the Move to Meaningful Internet Systems. OTM 2017 Conferences - Confederated International Conferences: CoopIS, C&TC, and ODBASE, Proceedings, Part I*, volume 10573 of *Lecture Notes in Computer Science*, pages 194–212. Springer, 2017. doi: 10.1007/978-3-319-69462-7_13. URL https://doi.org/10.1007/978-3-319-69462-7_13. (Cited on pages 7, 53, and 54.)
- [81] Sankalita Mandal, Matthias Weidlich, and Mathias Weske. Events in business process implementation: Early subscription and event buffering. In *Business Process Management Forum - BPM Forum*, volume 297 of *Lecture Notes in Business Information Processing*, pages 141–159. Springer, 2017. doi: 10.1007/978-3-319-65015-9_9. URL https://doi.org/10.1007/978-3-319-65015-9_9. (Cited on pages 50, 65, 69, 83, 110, and 122.)
- [82] Sankalita Mandal, Marcin Hewelt, Maarten Oestreich, and Mathias Weske. A classification framework for iot scenarios. In *Business Process Management Workshops - BPM 2018 International Workshops*, volume 342 of *Lecture Notes in Business Information Processing*, pages 458–469. Springer, 2018. doi: 10.1007/978-3-030-11641-5_36. URL https://doi.org/10.1007/978-3-030-11641-5_36. (Cited on pages 7, 42, 49, and 53.)
- [83] Felix Mannhardt, Massimiliano de Leoni, Hajo A. Reijers, Wil M. P. van der Aalst, and Pieter J. Toussaint. Guided process discovery - a pattern-based approach. *Inf. Syst.*, 76:1–18, 2018. (Cited on page 32.)
- [84] R. Meier and V. Cahill. Steam: event-based middleware for wireless ad hoc networks. In *Proceedings 22nd International Conference on Distributed Computing Systems Workshops*, pages 639–644, July 2002. doi: 10.1109/ICDCSW.2002.1030841. (Cited on page 48.)

- [85] Giovanni Meroni. *Artifact-driven Business Process Monitoring*. PhD thesis, 06 2018. (Cited on pages 46 and 49.)
- [86] A. Metzger, P. Leitner, D. Ivanović, E. Schmieders, R. Franklin, M. Carro, S. Dustdar, and K. Pohl. Comparing and combining predictive business process monitoring techniques. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, 45(2):276–290, Feb 2015. ISSN 2168-2216. doi: 10.1109/TSMC.2014.2347265. (Cited on page 15.)
- [87] A Meyer, A Polyvyanyy, and M Weske. Weak Conformance of Process Models with respect to Data Objects. In *Services and their Composition (ZEUS)*, 2012. (Cited on page 14.)
- [88] M Muehlen and J Recker. How Much Language is Enough? Theoretical and Practical Use of the Business Process Modeling Notation. In *Advanced Information Systems Engineering*, pages 465–479. Springer, 2008. (Cited on page 136.)
- [89] Max Muhlhauser and Iryna Gurevych. *Ubiquitous Computing Technology for Real Time Enterprises*. Information Science Reference - Imprint of: IGI Publishing, Hershey, PA, 2007. ISBN 1599048353, 9781599048352. (Cited on pages 30 and 31.)
- [90] Jorge Munoz-Gama. Conformance checking and diagnosis in process mining. In *Lecture Notes in Business Information Processing*, 2016. (Cited on page 14.)
- [91] S. Nechifor, A. Petrescu, D. Damian, D. Puiu, and B. Târnaucă. Predictive analytics based on cep for logistic of sensitive goods. In *2014 International Conference on Optimization of Electrical and Electronic Equipment (OPTIM)*, pages 817–822, May 2014. doi: 10.1109/OPTIM.2014.6850965. (Cited on pages 47 and 53.)
- [92] OASIS. Web Services Business Process Execution Language, Version 2.0, April 2007. (Cited on page 4.)
- [93] Michael Offel, Han van der Aa, and Matthias Weidlich. Towards net-based formal methods for complex event processing. In *Proceedings of the Conference "Lernen, Wissen, Daten, Analysen", LWDA 2018, Mannheim, Germany, August 22-24, 2018.*, pages 281–284, 2018. URL <http://ceur-ws.org/Vol-2191/paper32.pdf>. (Cited on page 83.)
- [94] OMG. Business Process Model and Notation (BPMN), Version 2.0, January 2011. (Cited on pages 3, 5, 18, 54, 65, 113, and 123.)
- [95] OMG. Unified Modeling Language (UML), Version 2.5, 2012. (Cited on pages 5 and 123.)
- [96] OMG. Decision Model and Notation (DMN), Version 1.1, June 2016. (Cited on page 16.)
- [97] M. Pesic and W. M. P. van der Aalst. A declarative approach for flexible business processes management. In Johann Eder and Schahram Dustdar, editors, *Business Process Management Workshops*, pages 169–180, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg. ISBN 978-3-540-38445-8. (Cited on page 15.)

- [98] Paul Pichler, Barbara Weber, Stefan Zugal, Jakob Pinggera, Jan Mendling, and Hajo A. Reijers. Imperative versus declarative process modeling languages: An empirical investigation. In Florian Daniel, Kamel Barkaoui, and Schahram Dustdar, editors, *Business Process Management Workshops*, pages 383–394, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. ISBN 978-3-642-28108-2. (Cited on page 15.)
- [99] P. R. Pietzuch and J. M. Bacon. Hermes: a distributed event-based middleware architecture. In *Proceedings 22nd International Conference on Distributed Computing Systems Workshops*, pages 611–618, 2002. doi: 10.1109/ICDCSW.2002.1030837. (Cited on page 48.)
- [100] Louchka Popova-Zeugmann. *Time and Petri Nets*. 11 2013. ISBN 978-3-642-41114-4. doi: 10.1007/978-3-642-41115-1. (Cited on page 21.)
- [101] Luise Pufahl, Sankalita Mandal, Kimon Batoulis, and Mathias Weske. Re-evaluation of decisions based on events. In *Enterprise, Business-Process and Information Systems Modeling - 18th International Conference, BPMDS*, volume 287 of *Lecture Notes in Business Information Processing*, pages 68–84. Springer, 2017. doi: 10.1007/978-3-319-59466-8_5. URL https://doi.org/10.1007/978-3-319-59466-8_5. (Cited on pages 7 and 46.)
- [102] Vivek Ranadive and Kevin Maney. *The Two-Second Advantage: How We Succeed by Anticipating the Future—Just Enough*. Crown Business, 2011. ISBN 0307887650. (Cited on page 3.)
- [103] Manfred Reichert, Stefanie Rinderle-Ma, and Peter Dadam. Flexibility in process-aware information systems. *TPNOC*, 5460:115–135, 2009. (Cited on page 50.)
- [104] Wolfgang Reisig. *Understanding Petri Nets - Modeling Techniques, Analysis Methods, Case Studies*. Springer, 2013. (Cited on page 21.)
- [105] Leonard Richardson and Sam Ruby. *Restful Web Services*. O’Reilly, first edition, 2007. ISBN 9780596529260. (Cited on page 5.)
- [106] Pedro H. Piccoli Richetti, Fernanda Araujo Baião, and Flávia Maria Santoro. Declarative process mining: Reducing discovered models complexity by pre-processing event logs. In Shazia Sadiq, Pnina Soffer, and Hagen Völzer, editors, *Business Process Management*, pages 400–407, Cham, 2014. Springer International Publishing. ISBN 978-3-319-10172-9. (Cited on page 32.)
- [107] A Rogge-Solti, N Herzberg, and L Pufahl. Selecting Event Monitoring Points for Optimal Prediction Quality. In *EMISA*, pages 39–52, 2012. (Cited on page 15.)
- [108] Sherif Sakr, Zakaria Maamar, Ahmed Awad, Boualem Benatallah, and Wil M. P. Van Der Aalst. Business process analytics and big data systems: A roadmap to bridge the gap. *IEEE Access*, PP:1–1, 11 2018. doi: 10.1109/ACCESS.2018.2881759. (Cited on pages 43 and 49.)
- [109] Matthias J. Sax, Guozhang Wang, Matthias Weidlich, and Johann-Christoph Freytag. Streams and tables: Two sides of the same coin. In *Proceedings of the International Workshop on Real-Time Business Intelligence*

and Analytics, BIRTE 2018, Rio de Janeiro, Brazil, August 27, 2018, pages 1:1–1:10, 2018. doi: 10.1145/3242153.3242155. URL <https://doi.org/10.1145/3242153.3242155>. (Cited on page 49.)

- [110] Helen Schonenberg, Ronny Mans, Nick Russell, Nataliya Mulyar, and Wil van der Aalst. Process flexibility: A survey of contemporary approaches. In Jan L. G. Dietz, Antonia Albani, and Joseph Barjis, editors, *Advances in Enterprise Engineering I*, pages 16–30, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. ISBN 978-3-540-68644-6. (Cited on page 50.)
- [111] Stefan Schönig, Lars Ackermann, Stefan Jablonski, and Andreas Ermer. An integrated architecture for iot-aware business process execution. In Jens Gulden, Iris Reinhartz-Berger, Rainer Schmidt, Sérgio Guerreiro, Wided Guédria, and Palash Bera, editors, *Enterprise, Business-Process and Information Systems Modeling*, pages 19–34, Cham, 2018. Springer International Publishing. ISBN 978-3-319-91704-7. (Cited on page 42.)
- [112] Stefan Schönig, Ana Paula Aires, Andreas Ermer, and Stefan Jablonski. *Workflow Support in Wearable Production Information Systems*, pages 235–243. 06 2018. ISBN 978-3-319-92900-2. doi: 10.1007/978-3-319-92901-9_20. (Cited on pages 42 and 49.)
- [113] Ronny Seiger, Christine Keller, Florian Niebling, and Thomas Schlegel. Modelling complex and flexible processes for smart cyber-physical environments. *Journal of Computational Science*, 10:137 – 148, 2015. ISSN 1877-7503. doi: <https://doi.org/10.1016/j.jocs.2014.07.001>. URL <http://www.sciencedirect.com/science/article/pii/S1877750314000970>. (Cited on page 47.)
- [114] Arik Senderovich, Andreas Rogge-Solti, Avigdor Gal, Jan Mendling, and Avishai Mandelbaum. The road from sensor data to process instances via interaction mining. In Selmin Nurcan, Pnina Soffer, Marko Bajec, and Johann Eder, editors, *Advanced Information Systems Engineering*, pages 257–273, Cham, 2016. Springer International Publishing. ISBN 978-3-319-39696-5. (Cited on page 32.)
- [115] Natalia Sidorova, Christian Stahl, and Nikola Trčka. Workflow soundness revisited: Checking correctness in the presence of data while staying conceptual. In Barbara Pernici, editor, *Advanced Information Systems Engineering*, pages 530–544, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg. ISBN 978-3-642-13094-6. (Cited on page 107.)
- [116] H Smith and P Fingar. *Business Process Management: The Third Wave*. 01 2006. (Cited on page 13.)
- [117] Pnina Soffer, Annika Hinze, Agnes Koschmider, Holger Ziekow, Claudio Di Ciccio, Boris Koldehofe, Oliver Kopp, Arno Jacobsen, Jan Sürmeli, and Wei Song. From event streams to process models and back: Challenges and opportunities. *Information Systems*, 01 2018. (Cited on pages 44 and 50.)
- [118] Kunal Suri. *Modeling the Internet of Things in Configurable Process Models*. PhD thesis, 02 2019. (Cited on page 45.)

- [119] Niek Tax, Natalia Sidorova, Reinder Haakma, and Wil M. P. van der Aalst. Event abstraction for process mining using supervised learning techniques. *CoRR*, abs/1606.07283, 2016. (Cited on page 32.)
- [120] Niek Tax, Ilya Verenich, Marcello La Rosa, and Marlon Dumas. Predictive business process monitoring with lstm neural networks. In Eric Dubois and Klaus Pohl, editors, *Advanced Information Systems Engineering*, pages 477–492, Cham, 2017. Springer International Publishing. ISBN 978-3-319-59536-8. (Cited on page 14.)
- [121] UNICORN. Complex Event Processing Platform. <https://bpt.hpi.uni-potsdam.de/UNICORN/WebHome>. (Cited on page 58.)
- [122] W. M. P. van der Aalst. Verification of workflow nets. In Pierre Azéma and Gianfranco Balbo, editors, *Application and Theory of Petri Nets 1997*, pages 407–426, Berlin, Heidelberg, 1997. Springer Berlin Heidelberg. ISBN 978-3-540-69187-7. (Cited on page 83.)
- [123] W. M. P. van der Aalst, K. M. van Hee, A. H. M. ter Hofstede, N. Sidorova, H. M. W. Verbeek, M. Voorhoeve, and M. T. Wynn. Soundness of workflow nets: classification, decidability, and analysis. *Formal Aspects of Computing*, 23(3):333–363, May 2011. ISSN 1433-299X. doi: 10.1007/s00165-010-0161-4. URL <https://doi.org/10.1007/s00165-010-0161-4>. (Cited on page 107.)
- [124] Wil van der Aalst. *Process Mining: Data Science in Action*. Springer Publishing Company, Incorporated, 2nd edition, 2016. ISBN 3662498502, 9783662498507. (Cited on pages 3, 13, and 15.)
- [125] Wil M. P. van der Aalst, Niels Lohmann, Peter Massuthe, Christian Stahl, and Karsten Wolf. Multiparty contracts: Agreeing and implementing interorganizational processes. *Comput. J.*, 53(1):90–106, 2010. doi: 10.1093/comjnl/bxn064. URL <http://dx.doi.org/10.1093/comjnl/bxn064>. (Cited on page 47.)
- [126] Maximilian Völker, Sankalita Mandal, and Marcin Hewelt. Testing event-driven applications with automatically generated events. In *Proceedings of the BPM Demo Track and BPM Dissertation Award co-located with 15th International Conference on Business Process Modeling*, volume 1920 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2017. URL http://ceur-ws.org/Vol-1920/BPM_2017_paper_182.pdf. (Cited on pages 115 and 117.)
- [127] Rainer von Ammon, C. Silberbauer, and C. Wolff. Domain specific reference models for event patterns - for faster developing of business activity monitoring applications. 2007. (Cited on pages 38, 45, and 49.)
- [128] Rainer von Ammon, Christoph Emmersberger, Torsten Greiner, Florian Springer, and Christian Wolff. Event-driven business process management. In *Fast Abstract, Second International Conference on Distributed Event-Based Systems, DEBS 2008, Rom, Juli 2008*, 2008. URL <https://epub.uni-regensburg.de/6829/>. (Cited on page 44.)
- [129] Barbara Weber, Jakob Pinggera, Stefan Zugal, and Werner Wild. Handling events during business process execution: An empirical test. In *ER-POIS@CAiSE*, 2010. (Cited on page 43.)

- [130] M Weidlich, G Decker, A Groß kopf, and M Weske. BPEL to BPMN: The Myth of a Straight-Forward Mapping. In *On the Move to Meaningful Internet Systems*, pages 265–282. Springer, 2008. (Cited on page 4.)
- [131] Matthias Weidlich. *Behavioural profiles: a relational approach to behaviour consistency*. PhD thesis, University of Potsdam, 2011. (Cited on pages 20 and 47.)
- [132] Matthias Weidlich, Holger Ziekow, Jan Mendling, Oliver Günther, Mathias Weske, and Nirmal Desai. Event-based monitoring of process execution violations. In *BPM*, pages 182–198. Springer, 2011. (Cited on pages 14, 44, and 49.)
- [133] Matthias Weidlich, Holger Ziekow, Avigdor Gal, Jan Mendling, and Mathias Weske. Optimizing event pattern matching using business process models. *IEEE Trans. Knowl. Data Eng.*, 26(11):2759–2773, 2014. doi: 10.1109/TKDE.2014.2302306. URL <https://doi.org/10.1109/TKDE.2014.2302306>. (Cited on pages 47 and 49.)
- [134] Mathias Weske. *Business Process Management - Concepts, Languages, Architectures, 2nd Edition*. Springer, 2012. ISBN 978-3-642-28615-5. doi: 10.1007/978-3-642-28616-2. (Cited on pages 3, 13, 15, 62, and 83.)
- [135] Mathias Weske. *Business Process Management - Concepts, Languages, Architectures, Third Edition*. Springer, 2019. ISBN 978-3-662-59431-5. doi: 10.1007/978-3-662-59432-2. URL <https://doi.org/10.1007/978-3-662-59432-2>. (Cited on page 21.)
- [136] Dennis Wolf. *Flexible event subscription in business processes*. Dissertation, Universität Potsdam, 2017. (Cited on pages 115 and 122.)
- [137] Karsten Wolf. Petri net model checking with lola 2. In Victor Khomenko and Olivier H. Roux, editors, *Application and Theory of Petri Nets and Concurrency*, pages 351–362, Cham, 2018. Springer International Publishing. ISBN 978-3-319-91268-4. (Cited on pages 83 and 97.)
- [138] Honguk Woo, Aloysius K. Mok, and Deji Chen. Realizing the potential of monitoring uncertain event streams in real-time embedded applications. *IEEE Real-Time and Embedded Technology and Applications*, 2007. (Cited on page 47.)
- [139] R. Worzberger, T. Kurpick, and T. Heer. Checking correctness and compliance of integrated process models. In *2008 10th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, pages 576–583, Sep. 2008. doi: 10.1109/SYNASC.2008.10. (Cited on page 101.)
- [140] C. Zang and Y. Fan. Complex event processing in enterprise information systems based on rfid. *Enterprise Information Systems*, 1(1):3–23, 2007. doi: 10.1080/17517570601092127. URL <https://doi.org/10.1080/17517570601092127>. (Cited on page 47.)

All links were last followed on 20.09.2019.

DECLARATION

I hereby confirm that I have authored this thesis independently and without use of others 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
December 2019

Sankalita Mandal

