

Artem Polyvyanyy *Editor*

# Process Querying Methods



Springer

# Process Querying Methods

Artem Polyvyanyy  
Editor

# Process Querying Methods

 Springer

*Editor*

Artem Polyvyanyy  
School of Computing  
and Information Systems  
The University of Melbourne  
Melbourne, VIC, Australia

ISBN 978-3-030-92874-2                      ISBN 978-3-030-92875-9 (eBook)  
<https://doi.org/10.1007/978-3-030-92875-9>

© Springer Nature Switzerland AG 2022

Chapter “Celonis PQL: A Query Language for Process Mining” is licensed under the terms of the Creative Commons Attribution 4.0 International license (<http://creativecommons.org/licenses/by/4.0/>). For further details see license information in the chapter.

This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.

The use of general descriptive names, registered names, trademarks, service marks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.

The publisher, the authors and the editors are safe to assume that the advice and information in this book are believed to be true and accurate at the date of publication. Neither the publisher nor the authors or the editors give a warranty, expressed or implied, with respect to the material contained herein or for any errors or omissions that may have been made. The publisher remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

This Springer imprint is published by the registered company Springer Nature Switzerland AG  
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

# Foreword

We have come a long way. Finally, thanks to the thought leadership of Artem Polyvyanyy, this book on process querying has become available. Artem describes the research area of business process querying as the study concerned with methods for automatically managing repositories of business process models [9]. Every bachelor student in computer science knows SQL, the structured query language that allows us to efficiently formulate SELECT statements for retrieving the data from a database that is currently of interest to us. So why it took so long until we had something similar available for working efficiently with business process repositories?

Let us go back to the year 2005 and let me share why I have always been excited about process querying. Business process modeling was just awfully heterogeneous at that time. The workflow patterns had just recently been published [14], which was the first step towards overcoming the Babylonian language confusion of process modeling. I saw the need for an integrated metamodel for process modeling [6], but the topic turned out to be too complicated for a PhD thesis. Several layers had to be disentangled: heterogeneous process modeling languages with heterogeneous semantics, heterogeneous process modeling tools, heterogeneous process model interchange formats, and industry was already working on what later became the BPMN standard. It was several years later that we finally saw satisfactory solutions, one in an initiative by Gero Decker, Hagen Overdick, and Mathias Weske on Oryx [2], which would later become Signavio, and another driven by Marcello La Rosa together with Hajo Reijers, Wil van der Aalst, Remco Dijkman, myself, Marlon Dumas, and Luciano García-Bañuelos [5], laying the foundations for Apromore.

But it was not only the metamodel of a language-independent process modeling repository that was missing. In 2005, we hardly understood how we could query the behavior of a process model. There had been first proposals for process query languages by Momotko and Subieta [8] as well as Klein and Bernstein [4], but the key challenge was still how to query for the behavioral semantics and not syntactic structures of a process model. Together with several colleagues, I investigated how we can calculate the behavioral similarity between two process models together with Boudewijn van Dongen, Wil van der Aalst, Remco Dijkman, Marlon Dumas, and Reina Käärrik [3, 7], but we were not the only ones. My move from Brisbane,

Australia, to Berlin, Germany, in 2008 turned out to be particularly fertile. Mathias Weske's team at HPI Potsdam had several talented PhD students with whom I got to collaborate. Artem Polyvyanyy was one of them. Together with Ahmed Awad, he worked on semantic querying of BPMN process models [1]. Step by step, we developed the formal foundations based on these efforts. Behavioral abstractions such as the behavioral profiles driven by Matthias Weidlich [15] turned out to be the key mechanisms for querying—a work that Artem further extended and generated towards the 4C spectrum of fundamental behavioral relations for concurrent systems [10].

With Artem's move to join the BPM group at QUT Brisbane in 2012, and more recently the University of Melbourne, everything finally fell into its place. The research on Apromore provided the ideal testbed for experimenting and implementing process querying, first APQL [13] and more recently PQL [12]. Artem's work on a generic query architecture [11] has become the de-facto standard in this area. This book is a testament to these inspiring developments around research and practice of process querying. Already for a while, querying for process mining has become a natural extension of the original work that focused on models. Maybe, as the formal foundations have been defined, tool implementations are available, and now also a book is published, it is time to start standardizing process querying like SQL. Enjoy this book and join the efforts towards further advancing process querying!

Berlin, Germany  
October 2021

Jan Mendling

## References

1. Awad, A., Polyvyanyy, A., Weske, M.: Semantic querying of business process models. In: 2008 12th International IEEE Enterprise Distributed Object Computing Conference, pp. 85–94. IEEE (2008)
2. Decker, G., Overdick, H., Weske, M.: Oryx—an open modeling platform for the bpm community. In: International Conference on Business Process Management, pp. 382–385. Springer (2008)
3. Dijkman, R.M., Dumas, M., van Dongen, B.F., Käärik, R., Mendling, J.: Similarity of business process models: Metrics and evaluation. *Inf. Syst.* **36**(2), 498–516 (2011). <https://doi.org/10.1016/j.is.2010.09.006>
4. Klein, M., Bernstein, A.: Toward high-precision service retrieval. *IEEE Internet Comput.* **8**(1), 30–36 (2004)
5. La Rosa, M., Reijers, H.A., van der Aalst, W.M.P., Dijkman, R.M., Mendling, J., Dumas, M., García-Bañuelos, L.: APROMORE: an advanced process model repository. *Expert Syst. Appl.* **38**(6), 7029–7040 (2011). <https://doi.org/10.1016/j.eswa.2010.12.012>

6. Mendling, J., de Laborda, C.P., Zdun, U.: Towards an integrated BPM schema: Control flow heterogeneity of PNML and BPEL4WS. In: Althoff, K., Dengel, A., Bergmann, R., Nick, M., Roth-Berghofer, T. (eds.) *Professional Knowledge Management, Third Biennial Conference, WM 2005, Kaiserslautern, Germany, April 10–13, 2005, Revised Selected Papers. Lecture Notes in Computer Science*, vol. 3782, pp. 570–579. Springer (2005). [https://doi.org/10.1007/11590019\\_65](https://doi.org/10.1007/11590019_65)
7. Mendling, J., van Dongen, B.F., van der Aalst, W.M.P.: On the degree of behavioral similarity between business process models. In: Nüttgens, M., Rump, F.J., Gadatsch, A. (eds.) *6. Workshop der Gesellschaft für Informatik e.V. (GI) und Treffen ihres Arbeitskreises “Geschäftsprozessmanagement mit Ereignisgesteuerten Prozessketten (WI-EPK)” St. Augustin, Deutschland, 29. November - 30. November 2007, CEUR Workshop Proceedings*, vol. 303, pp. 39–58. CEUR-WS.org (2007). <http://ceur-ws.org/Vol-303>
8. Momotko, M., Subieta, K.: Process query language: A way to make workflow processes more flexible. In: *East European Conference on Advances in Databases and Information Systems*, pp. 306–321. Springer (2004)
9. Polyvyanyy, A.: Business process querying. In: Sakr, S., Zomaya, A.Y. (eds.) *Encyclopedia of Big Data Technologies*. Springer (2019). [https://doi.org/10.1007/978-3-319-63962-8\\_108-1](https://doi.org/10.1007/978-3-319-63962-8_108-1)
10. Polyvyanyy, A., Weidlich, M., Conforti, R., La Rosa, M., ter Hofstede, A.H.M.: The 4C spectrum of fundamental behavioral relations for concurrent systems. In: Ciardo, G., Kindler, E. (eds.) *Application and Theory of Petri Nets and Concurrency - 35th International Conference, PETRI NETS 2014, Tunis, Tunisia, June 23–27, 2014. Proceedings. Lecture Notes in Computer Science*, vol. 8489, pp. 210–232. Springer (2014). [https://doi.org/10.1007/978-3-319-07734-5\\_12](https://doi.org/10.1007/978-3-319-07734-5_12)
11. Polyvyanyy, A., Ouyang, C., Barros, A., van der Aalst, W.M.P.: Process querying: Enabling business intelligence through query-based process analytics. *Decis. Support Syst.* **100**, 41–56 (2017). <https://doi.org/10.1016/j.dss.2017.04.011>
12. Polyvyanyy, A., Pika, A., ter Hofstede, A.H.: Scenario-based process querying for compliance, reuse, and standardization. *Information Systems*, 101563 (2020)
13. ter Hofstede, A.H., Ouyang, C., La Rosa, M., Song, L., Wang, J., Polyvyanyy, A.: Apql: A process-model query language. In: *Asia-Pacific Conference on Business Process Management*, pp. 23–38. Springer (2013)
14. van der Aalst, W.M., ter Hofstede, A.H., Kiepuszewski, B., Barros, A.P.: Workflow patterns. *Distrib. Parallel Databases* **14**(1), 5–51 (2003)
15. Weidlich, M., Mendling, J., Weske, M.: Efficient consistency measurement based on behavioral profiles of process models. *IEEE Trans. Softw. Eng.* **37**(3), 410–429 (2011). <https://doi.org/10.1109/TSE.2010.96>

# Preface

Dear Reader,

The idea of this book is due to the last decade of observations and academic discussions at scientific conferences in the areas of business process management [2, 14] and process mining [13]. These observations and discussions acknowledge the existence of a core repertoire of techniques for retrieving and manipulating process-related artifacts, for example, records of process executions, data generated by process executions, process designs, and semantic process annotations that convey domain knowledge. Such core techniques are reused in multiple contexts to support various use cases. A selection of these use cases includes process compliance, process standardization, process reuse, process redesign, process discovery and enhancement, process instance migration, and process monitoring. Moreover, the role of process querying as the mediator between the engine and the user interface in commercial process mining tools becomes increasingly important. However, process querying methods and techniques are often redefined, redeveloped, and reimplemented, with inconsistent adaptations, across scattered academic and industrial projects.

Process querying aims to identify techniques for retrieving and manipulating processes, models of processes, and related artifacts that are inherent to the various practical applications to promote centralized improvement and reuse of these techniques for the benefit of the use cases they support. It is envisaged that these core techniques will be made available for use via machine-readable instructions, called process queries, as part of domain-specific programming languages. Process model collections and process repositories, such as business process repositories, event log collections, event streams, and software code repositories, without such languages are like databases without SQL, that is, collections of tuples without effective and efficient ways to systematically derive value from them.

I became interested in the topic of process querying in 2008, shortly after starting my PhD project in Potsdam, Germany, in the group of Mathias Weske. Back then, together with Ahmed Awad and Mathias Weske, we studied ways information retrieval algorithms can improve techniques for retrieving process models [1]. However, my early interest in process querying did not go beyond a single publication.



My next acquaintance with the topic of process querying happened in 2012 when I joined the Queensland University of Technology, Brisbane, Australia, where Arthur H. M. ter Hofstede was driving research on A Process-model Query Language (APQL) [12]. While working on APQL, I got several ideas on how process querying should be done. For the next several years, I worked on formal foundations of process querying, including the work on untanglings [5, 12], behavioral profiles in general [7] and the 4C spectrum of behavioral relations in particular [3], and techniques for process model repair [8]. These works, and the ongoing thinking process that never left me, in collaboration with my colleagues, led to the definition of the problem of process querying, a concept of a process querying method, and a framework for devising such methods. Simultaneously with the above-listed activities, I was driving research on a language, called Process Query Language (PQL) [4, 6, 10, 11], for querying collections of process models based on the behaviors these models describe.

This book is intended for researchers, practitioners, lecturers, students, and tool vendors. First, all the chapters in this book are contributed by active researchers in the research disciplines of business process management, process mining, and process querying. These chapters describe state-of-the-art methods for process querying, discuss use cases of process querying, and suggest directions for future work for advancing the field. Hence, we hope the book will inspire other researchers to join the effort and develop elegant solutions to process querying problems outlined in this book. Second, by reading this book, practitioners, like business and process analysts, and data and process scientists, can broaden their repertoires of tools for analyzing large arrays of process data. Third, lecturers can use the materials from this book to present the concept of process querying and concrete methods for process querying to their students, while higher degree research students can apply process querying methods to solve engineering problems or, again, contribute to the research in process querying. Finally, several tool vendors already embed principles of process querying in their commercial tools; one of the chapters in this book comes from a vendor who develops and successfully integrates process querying ideas and methods in their toolchain. Thus, for vendors, this book depicts the existing palette of principles available in process querying to consider embedding them into their tools.

The book comprises 16 contributed chapters distributed over four parts and two auxiliary chapters. The auxiliary chapters by the editor provide an introduction to the area of process querying and give a summary of the area presented in this book as well as methods and techniques for process querying. The introductory chapter also presents a process querying framework, a system of abstract components that, when instantiated, result in a concrete process querying method. The contributed chapters present various process querying methods while also discussing how they instantiate the framework. This link to the framework makes a common theme through the book, supporting the comparison of the presented methods. The four parts of the book are due to the distinctive features of the methods they include. The first three parts are devoted to querying event logs generated by IT systems that support business processes at organizations, querying process designs captured in process models, and methods for querying both event logs and process models.

The methods in these three parts usually define a language for specifying process queries. The fourth part discusses methods that operate over inputs other than event logs and process models, for example, streams of process events, or do not develop dedicated languages for specifying queries, for example, methods for assessing process model similarity.

I am thankful to all the contributors of this book, and concretely to Alexander Artikis, Amal Elgammal, Amin Beheshti, Andreas Oberweis, Andreas Schoknecht, Antonia M. Reina Quintero, Antonio Cancela Díaz, Boualem Benatallah, Carl Corea, Chiara Di Francescomarino, Christoph Drodts, David Becher, Dennis M. Riehle, Eduardo Gonzalez Lopez de Murillas, Emiliano Reynares, Fabrizio Smith, Farhad Amouzar, Francesco Taglino, Hajo A. Reijers, Hamid Reza Motahari-Nezhad, Han van der Aa, Harald Störrle, Jerome Geyer-Klingenberg, Jessica Ambrosy, Jorge Roa, Jose Miguel Pérez Álvarez, Kazimierz Subieta, Klaus Kammerer, Luisa Parody, Manfred Reichert, María Laura Caliusco, María Teresa Gómez-López, Mariusz Momotko, Martin Klenk, Matthias Weidlich, Maurizio Proietti, Oktay Turetken, Pablo Villarreal, Paolo Tonella, Patrick Delfmann, Peter Fettke, Ralf Laue, Remco M. Dijkman, Rik Eshuis, Robert Seilbeck, Rüdiger Pryss, Samira Ghodrathnama, Steffen Höhenberger, Thomas Vogelgesang, Tom Thaler, Vlad Acretoai, and Wil van der Aalst. Thank you for your hard work, commitment, and patience. I thank Springer for publishing this book and, specifically, Ralf Gerstner for managing the communication from the publisher's side and providing timely recommendations related to the book preparation process. I thank Chun Ouyang, Alistair Barros, and Wil van der Aalst, with whom we shaped the concept of process querying and designed and validated a framework for defining process querying methods [9]. I also thank Arthur H. M. ter Hofstede for drawing my attention to the problem of process querying. Finally, I thank Jan Mendling for many years of fruitful academic collaboration and for writing the foreword to this book. Thank you All! Together, we have come a long way.

PS. For further resources on process querying and the book and information about the workshop series on the topic of process querying, please refer to our Website: [processquerying.com](http://processquerying.com).

Melbourne, VIC, Australia  
October 2021

Artem Polyvyanyy

## References

1. Awad, A., Polyvyanyy, A., Weske, M.: Semantic querying of business process models. In: EDOC, pp. 85–94. IEEE Computer Society (2008)
2. Dumas, M., Rosa, M.L., Mendling, J., Reijers, H.A.: Fundamentals of Business Process Management, 2nd edn. Springer (2018). <https://doi.org/10.1007/978-3-662-56509-4>

3. Polyvyanyy, A., Weidlich, M., Conforti, R., Rosa, M.L., ter Hofstede, A.H.M.: The 4c spectrum of fundamental behavioral relations for concurrent systems. In: *Petri Nets. Lecture Notes in Computer Science*, vol. 8489, pp. 210–232. Springer (2014)
4. Polyvyanyy, A., Rosa, M.L., ter Hofstede, A.H.M.: Indexing and efficient instance-based retrieval of process models using untanglings. In: *CAiSE. Lecture Notes in Computer Science*, vol. 8484, pp. 439–456. Springer (2014)
5. Polyvyanyy, A., Rosa, M.L., Ouyang, C., ter Hofstede, A.H.M.: Untanglings: a novel approach to analyzing concurrent systems. *Formal Aspects Comput.* **27**(5–6), 753–788 (2015)
6. Polyvyanyy, A., Corno, L., Conforti, R., Raboczi, S., Rosa, M.L., Fortino, G.: Process querying in apromore. In: *BPM (Demos). CEUR Workshop Proceedings*, vol. 1418, pp. 105–109. CEUR-WS.org (2015)
7. Polyvyanyy, A., Armas-Cervantes, A., Dumas, M., García-Bañuelos, L.: On the expressive power of behavioral profiles. *Formal Aspects Comput.* **28**(4), 597–613 (2016)
8. Polyvyanyy, A., van der Aalst, W.M.P., ter Hofstede, A.H.M., Wynn, M.T.: Impact-driven process model repair. *ACM Trans. Softw. Eng. Methodol.* **25**(4), 28:1–28:60 (2017)
9. Polyvyanyy, A., Ouyang, C., Barros, A., van der Aalst, W.M.P.: Process querying: Enabling business intelligence through query-based process analytics. *Decis. Support Syst.* **100**, 41–56 (2017). <https://doi.org/10.1016/j.dss.2017.04.011>
10. Polyvyanyy, A., ter Hofstede, A.H.M., Rosa, M.L., Ouyang, C., Pika, A.: Process query language: Design, implementation, and evaluation. *CoRR abs/1909.09543* (2019)
11. Polyvyanyy, A., Pika, A., ter Hofstede, A.H.M.: Scenario-based process querying for compliance, reuse, and standardization. *Inf. Syst.* **93**, 101,563 (2020)
12. ter Hofstede, A.H.M., Ouyang, C., Rosa, M.L., Song, L., Wang, J., Polyvyanyy, A.: APQL: A process-model query language. In: *AP-BPM. Lecture Notes in Business Information Processing*, vol. 159, pp. 23–38. Springer (2013)
13. van der Aalst, W.M.P.: *Process Mining - Data Science in Action*, 2nd edn. Springer (2016). <https://doi.org/10.1007/978-3-662-49851-4>
14. Weske, M.: *Business Process Management - Concepts, Languages, Architectures*, Third Edition. Springer (2019). <https://doi.org/10.1007/978-3-662-59432-2>

# Contents

<b>Introduction to Process Querying</b> .....	1
Artem Polyvyanyy	
<b>Part I Event Log Querying</b>	
<b>BP-SPARQL: A Query Language for Summarizing and Analyzing Big Process Data</b> .....	21
Amin Beheshti, Boualem Benatallah, Hamid Reza Motahari-Nezhad, Samira Ghodratnama, and Farhad Amouzgar	
<b>Data-Aware Process Oriented Query Language</b> .....	49
Eduardo Gonzalez Lopez de Murillas, Hajo A. Reijers, and Wil M. P. van der Aalst	
<b>Process Instance Query Language and the Process Querying Framework</b> .....	85
Jose Miguel Pérez Álvarez, Antonio Cancela Díaz, Luisa Parody, Antonia M. Reina Quintero, and María Teresa Gómez-López	
<b>Part II Process Model Querying</b>	
<b>The Diagramed Model Query Language 2.0: Design, Implementation, and Evaluation</b> .....	115
Patrick Delfmann, Dennis M. Riehle, Steffen Höhenberger, Carl Corea, and Christoph Drodtt	
<b>VM*: A Family of Visual Model Manipulation Languages</b> .....	149
Harald Störrle and Vlad Acrețoiaie	
<b>The BPMN Visual Query Language and Process Querying Framework</b> .....	181
Chiara Di Francescomarino and Paolo Tonella	

<b>Retrieving, Abstracting, and Changing Business Process Models with PQL</b> .....	219
Klaus Kammerer, Rüdiger Pryss, and Manfred Reichert	
<b>QuBPAL: Querying Business Process Knowledge</b> .....	255
Maurizio Proietti, Francesco Taglino, and Fabrizio Smith	
<b>CRL and the Design-Time Compliance Management Framework</b> .....	285
Amal Elgammal and Oktay Turetken	
<b>Process Query Language</b> .....	313
Artem Polyvyanyy	
<b>Part III Event Log and Process Model Querying</b>	
<b>Business Process Query Language</b> .....	345
Mariusz Momotko and Kazimierz Subieta	
<b>Celonis PQL: A Query Language for Process Mining</b> .....	377
Thomas Vogelgesang, Jessica Ambrosy, David Becher, Robert Seilbeck, Jerome Geyer-Klingenberg, and Martin Klenk	
<b>Part IV Other Process Querying Methods</b>	
<b>Process Querying Using Process Model Similarity</b> .....	411
Remco M. Dijkman and Rik Eshuis	
<b>Logic-Based Approaches for Process Querying</b> .....	437
Ralf Laue, Jorge Roa, Emiliano Reynares, María Laura Caliusco, and Pablo Villarreal	
<b>Process Model Similarity Techniques for Process Querying</b> .....	459
Andreas Schoknecht, Tom Thaler, Ralf Laue, Peter Fettke, and Andreas Oberweis	
<b>Complex Event Processing Methods for Process Querying</b> .....	479
Han van der Aa, Alexander Artikis, and Matthias Weidlich	
<b>Process Querying: Methods, Techniques, and Applications</b> .....	511
Artem Polyvyanyy	
<b>Index</b> .....	525

# Contributors

**Vlad Acrețoiaie** FREQUENTIS Romania SRL, Cluj-Napoca, Romania

**Farhad Amouzgar** Macquarie University, Sydney, NSW, Australia

**Alexander Artikis** Department of Maritime Studies, University of Piraeus, Piraeus, Greece  
Institute of Informatics & Telecommunications, NCSR Demokritos, Athens, Greece

**David Becher** Celonis SE, Munich, Germany

**Amin Beheshti** Macquarie University, Sydney, NSW, Australia

**Boualem Benatallah** University of New South Wales, Sydney, NSW, Australia

**María Laura Caliusco** Universidad Tecnológica Nacional – Facultad Regional Santa Fe, Santa Fe, Argentina

**Carl Corea** University of Koblenz-Landau, Koblenz, Germany

**Eduardo Gonzalez Lopez de Murillas** Eindhoven University of Technology, Eindhoven, The Netherlands

**Antonio Cancela Díaz** Dto. Lenguajes y Sistemas Informáticos, Universidad de Sevilla, Sevilla, Spain

**Patrick Delfmann** University of Koblenz-Landau, Koblenz, Germany

**Chiara Di Francescomarino** Fondazione Bruno Kessler–IRST, Trento, Italy

**Remco M. Dijkman** Eindhoven University of Technology, Eindhoven, The Netherlands

**Christoph Drodtt** University of Koblenz-Landau, Koblenz, Germany

**Amal Elgammal** Faculty of Computers and Information, Cairo University, Giza, Egypt

**Rik Eshuis** Eindhoven University of Technology, Eindhoven, The Netherlands

**Peter Fettke** German Research Center for Artificial Intelligence (DFKI) and Saarland University, Saarbrücken, Germany

**Jerome Geyer-Klingenberg** Celonis SE, Munich, Germany

**Samira Ghodrattnama** Macquarie University, Sydney, NSW, Australia

**María Teresa Gómez-López** Dto. Lenguajes y Sistemas Informáticos, Universidad de Sevilla, Sevilla, Spain

**Steffen Höhenberger** University of Münster – ERCIS, Münster, Germany

**Klaus Kammerer** Institute of Databases and Information Systems, Ulm University, Ulm, Germany

**Jessica Ambrosy** Celonis SE, Munich, Germany

**Martin Klenk** Celonis SE, Munich, Germany

**Ralf Laue** University of Applied Sciences Zwickau, Department of Computer Science, Zwickau, Germany

**Mariusz Momotko** Amazon Alexa, Amazon Development Center, Gdansk, Poland

**Hamid Reza Motahari-Nezhad** EY AI Lab, Santa Clara, CA, USA

**Andreas Oberweis** Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany

**Luisa Parody** Dto. Métodos Cuantitativos, Universidad Loyola Andalucía, Dos Hermanas, Spain

**Jose Miguel Pérez Álvarez** Naver Labs Europe, Meylan, France

**Artem Polyvyanyy** School of Computing and Information Systems, The University of Melbourne, Melbourne, VIC, Australia

**Maurizio Proietti** CNR-IASI, Rome, Italy

**Rüdiger Pryss** Institute of Clinical Epidemiology and Biometry, University of Würzburg, Würzburg, Germany

**Antonia M. Reina Quintero** Dto. Lenguajes y Sistemas Informáticos, Universidad de Sevilla, Sevilla, Spain

**Manfred Reichert** Institute of Databases and Information Systems, Ulm University, Ulm, Germany

**Hajo A. Reijers** Department of Information and Computing Sciences, Utrecht University, Utrecht, The Netherlands

**Emiliano Reynares** Universidad Tecnológica Nacional, Facultad Regional Santa Fe, Santa Fe, Argentina

**Dennis M. Riehle** University of Koblenz-Landau, Koblenz, Germany

**Jorge Roa** Universidad Tecnológica Nacional, Facultad Regional Santa Fe, Santa Fe, Argentina

**Andreas Schoknecht** Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany

**Robert Seilbeck** Celonis SE, Munich, Germany

**Fabrizio Smith** United Technologies Research Center, Rome, Italy

**Harald Störrle** QAware GmbH, München, Germany

**Kazimierz Subieta** Polish-Japanese Academy of Information Technology, Warsaw, Poland

**Francesco Taglino** CNR-IASI, Rome, Italy

**Tom Thaler** German Research Center for Artificial Intelligence (DFKI) and Saarland University, Saarbrücken, Germany

**Paolo Tonella** Università della Svizzera Italiana (USI), Lugano, Switzerland

**Oktay Turetken** Eindhoven University of Technology, Eindhoven, The Netherlands

**Han van der Aa** Data and Web Science Group, University of Mannheim, Mannheim, Germany

**Wil M. P. van der Aalst** Department of Computer Science, RWTH Aachen University, Aachen, Germany

**Pablo Villarreal** Facultad Regional Santa Fe, Universidad Tecnológica Nacional, Santa Fe, Argentina

**Thomas Vogelgesang** Celonis SE, Munich, Germany

**Matthias Weidlich** Department of Computer Science, Humboldt-Universität zu Berlin, Berlin, Germany



# Acronyms

The list of important acronyms used in this book is proposed below.

ABNF	Augmented Backus-Naur Form
ABox	Assertion component of a knowledge base (Assertion Box)
AML	Anti-Money Laundering
API	Application Programming Interface
BDO	Business Domain Ontology
BI	Business Intelligence
BNF	Backus Naur Form
BPAL	Business Process Abstract Language
BPD	Business Process Diagram
BPEL	Business Process Execution Language
BPKB	Business Process Knowledge Base
BPMNO	Business Process Model and Notation Ontology
BPMN	Business Process Model and Notation
BPMS	Business Process Management System
BPM	Business Process Management
BPS	Business Process Schema
BP	Business Process
BRO	Business Reference Ontology
CEP	Complex Event Processing
CF	Causal Footprints
CLI	Command-Line Interface
CMKB	Compliance Management Knowledge Base
CMM	Connected Meta Model
CQL	Continuous Query Language
CRL	Compliance Request Language
CRM	Customer Relationship Management
CRT	Current Reality Tree
CRUD	Create, Read, Update, Delete
CR	Compliance Requirement

CTE	Common Table Expressions
CTL	Computation Tree Logic
Celonis PQL	Celonis Process Query Language
DAPOQ-Lang	Data-Aware Process Oriented Query Language
DDL	Data Definition Language
DL	Description Logic
DML	Data Manipulation Language
DMN	Decision Model and Notation
DMQL	Diagrammed Model Query Language
DOR	Domain Ontology Relationship
DRD	Decision Requirement Diagram
DSL	Domain Specific Language
DS	Data Storage
EBNF	Extended Backus-Naur Form
EMF	Eclipse Modeling Framework
ENVS	ENVironment Stack
EPC	Event-driven Process Chain
EPL	Esper Pattern Language
ER Diagram	Entity Relationship Diagram
ERM	Entity-Relationship Model
ERP	Enterprise Resource Planning
ETL	Extract, Transform, Load
EUM	End-User Modeler
FBSE	Feature-Based Similarity Estimation
FEEL	Friendly Enough Expression Language
FIBO	Financial Industry Business Ontology
FIRO	Financial Industry Regulatory Ontology
FTL	ForSpec Temporal Logic
GA	Guarded Automaton
GMQL	Generic Model Query Language
GPL	General Purpose Language
GUI	Graphical User Interface
HDFS	Hadoop Distributed File System
IBC	Intelligent Business Cloud
IDE	Integrated Development Environment
IPO	Input Process Output
ITSM	IT Service Management
IT	Information Technology
jBPT	Business Process Technologies for Java
JIT	Just-In-Time
KPI	Key Performance Indicator
LCA	Language Concept Appropriateness
LCST	Longest Common Subsequence of Traces
LoLA	Low Level Analyzer
LP	Logic Programming

LS3	Latent Semantic Analysis-based Similarity Search
LTL	Linear Temporal Logic
MDE	Model-Driven Engineering
MTL	Model Transformation Language or Metrical Temporal Logic
NLP	Natural Language Processing
NL	Natural Language
OLAP	Online Analytical Processing
OMG	Object Management Group
OWL	Web Ontology Language
OpenSLEX	Open SQL Log Exchange Format
PADAS	Process-Aware Data Suite
PC	Personal Computer
PIM	Platform Independent Model
PIQE	Process Instance Query Expression
PIQL	Process Instance Query Language
PNML	Petri Net Markup Language
PPI	Process Performance Indicator
PQF	Process Querying Framework
PQL	Process Query Language
PSM	Platform Specific Language
QE	Query Execution
QRES	Query Result Stack
QU	Query Understanding
QoS	Quality of Service
QuBPAL	Query Language for BPAL
RDBMS	Relational Database Management System
RDFS	RDF Schema
RDF	Resource Description Framework
REST	Representational State Transfer
SAP	Systems, Applications and Products
SBA	Stack Based Approach
SESE	Single Entry Single Exit
SLA	Service Level Agreement
SPARQL	SPARQL Protocol And RDF Query Language
SQL	Structured Query Language
SSCAN	Similarity Score based on Common Activity Names
SWRL	Semantic Web Rule Language
SoD	Segregation of Duties
TBox	Terminological component of a knowledge base (Terminological Box)
TP	True Positive
UML	Unified Modeling Language
VM*	Visual Model Manipulation Language

VQL	Visual Query Language
XES	Extensible Event Stream
XMI	XML Metadata Interchange
XSLT	Extensible Stylesheet Language Transformations

# Introduction to Process Querying



Artem Polyvyanyy

**Abstract** This chapter gives a brief introduction to the research area of process querying. Concretely, it articulates the motivation and aim of process querying, gives a definition of process querying, presents the core artifacts studied in process querying, and discusses a framework for guiding the design, implementation, and evaluation of methods for process querying.

## 1 Introduction

A business process is a plan and coordination of activities and resources of an organization that aim to achieve a business objective [22]. Business Process Management (BPM) is an interdisciplinary field that studies concepts and methods that support and improve the way business processes are designed, performed, and analyzed in organizations. BPM enables organizations to systematically control operational processes with the ultimate goal of reducing their costs, execution times, and failure rates through incremental changes and radical innovations [4, 22].

Over the last two decades, many methods, techniques, and tools have been devised to support BPM practices in organizations. Use cases addressed by BPM range from regulatory process compliance, via process standardization and reuse, to variant analysis, process instance migration, and process mining techniques for automated process modeling, enhancement, and conformance checking based on event data generated by IT systems. Despite being devised for different use cases, BPM approaches and tools often rely on similar underlying algorithms, process analysis techniques, and constructed process analytics. For example, process compliance, standardization, reuse, and variant analysis methods rely on algorithms for retrieving processes that describe a case with conditions that capture a compliance violation or a process fragment suitable for standardization, variant identification, or

---

A. Polyvyanyy (✉)

School of Computing and Information Systems, Faculty of Engineering and Information Technology, The University of Melbourne, Melbourne, VIC, Australia

e-mail: [artem.polyvyanyy@unimelb.edu.au](mailto:artem.polyvyanyy@unimelb.edu.au)

© Springer Nature Switzerland AG 2022

A. Polyvyanyy (ed.), *Process Querying Methods*,

[https://doi.org/10.1007/978-3-030-92875-9\\_1](https://doi.org/10.1007/978-3-030-92875-9_1)

reuse in fresh process designs. Process instance migration and process compliance may further rely on techniques for automatically augmenting processes, such as resolving the issues associated with the non-compliance of the processes or adaptation of a long-running process instance from the old to a new process design.

Process querying aims to identify core algorithms, analysis, and analytics over processes to promote their centralized development and improvement for later reuse when solving practical problems concerned with the management of processes and process-related artifacts, like resources, information, and data. We refer to such core process-related computations as *process querying methods*. Process querying multiplies the effect of process querying methods in different use cases and suppresses reinventions of such methods in different contexts.

The remainder of this chapter is organized as follows. The next section elaborates on the aim of process querying and gives definitions of process querying and a method for process querying. Then, Sect. 3 presents a framework for devising process querying methods. The framework consists of abstract components, each with well-defined interfaces and functionality, that, when instantiated, result in a concrete method for process querying. The chapter closes with conclusions that also shed light on the directions for future work in process querying.

## 2 Process Querying

This section discusses the objective and the definition of the research area of process querying and rigorously defines the concept of a process querying method.

### 2.1 Objective

Process querying aims to support systematic improvement and reuse of methods, techniques, and tools for manipulating and managing models of processes, either already executed or designed existing or envisioned processes, and process-related resources, information, and data. The need for scoping the area of process querying has emerged from numerous observations of non-coordinated efforts for developing approaches for automated management and manipulation of process-related artifacts in the research disciplines of BPM [4, 22] and process mining [19]. To name a few, examples of research problems studied in BPM that fall in the scope of process querying include process compliance, process standardization, process reuse, process migration, process selection, process variance management, process selection, process discovery, process enhancement, and correctness checking [13]. Existing solutions to these problems often rely on techniques that share algorithmic ideas and principles. Hence, instead of conducting scattered, in silos, studies, with process querying, we propose to identify and study such central ideas and principles to improve and reuse them when solving practical process-related problems. Though

process querying emerges from the research disciplines of BPM and process mining, we envisage its application in other process-related fields, including software engineering, information systems engineering, computing, programming language theory, and process science.

## 2.2 Definition

Process querying emerges at the intersection of research areas concerned with modeling, analysis, and improvement of processes. Before giving our definition of process querying, we discuss several such areas and their relation to process querying.

**Big Data** Big data studies ways to analyze large datasets. Here, we usually speak about datasets that are too large to be analyzed using traditional techniques. Process querying also researches ways to analyze extremely large datasets but is primarily concerned with datasets that comprise event data, e.g., executions of IT systems, records of business processes, user interactions with information systems, and timestamped sensor data. In addition, process querying deals with descriptions of potentially infinite collections of processes.

**Process Modeling** A process model is a simplified representation of a real-world or an envisioned process, or collection of processes, that serves a particular purpose for a target audience. Process modeling is a technique to construct a process model. Process querying studies techniques that support instructions for automated and semi-automated process modeling. Examples of such instructions include removing or inserting parts of a process model to ensure it represents the desired collection of processes for the envisaged purpose and audience.

**Process Analysis** Process analysis studies ways to derive insights about the quality of processes, including their correctness, validity, and performance characteristics. Process querying relies on existing and studies new process analysis techniques to retrieve existing and modeling new processes with desired quality profiles. For instance, a process query can specify an intent to retrieve all processes with duration in a given range or augment process designs to ensure their correct future executions under new constraints.

**Process Analytics** Process analytics studies techniques for computational and statistical analysis of processes and the event data they induce. It is also concerned with identifying meaningful patterns in event data and communication of these patterns. Process querying relies on and extends process analytics to apply it when retrieving and manipulating processes and related artifacts. An example of such synergy between process analytics and querying is an instruction to retrieve and replace process patterns that lead to negative overall process outcomes with the patterns that were observed to result in positive outcomes.

**Process Intelligence** Process intelligence studies ways to infer insights from processes and the related resources, information, and data for their subsequent use. Examples of such insights include causes for poorly performing or unsuccessful process executions, while sample uses of the inferred insights include reporting and decision-making.

The definition of process querying is an evolving concept that is continuously refined via an iterative process of embracing and solving practical problems for retrieving and manipulating models of process and process-related artifacts. The current snapshot of this definition is given below:

Process querying combines concepts from Big data and process modeling and analysis with process analytics and process intelligence to study methods, techniques, and tools for retrieving and manipulating models of real-world and envisioned processes, and the related resources, information, and data, to organize and extract process-related insights for subsequent systematic use.

Therefore, the idea of process querying is to systematically extract insights from descriptions of processes, e.g., event logs of IT systems or process models, and the associated artifacts, e.g., resources used to support process executions, information capturing the domain knowledge, and data generated during process executions, stored in process repositories of organizations using effective instructions captured in process queries implemented using efficient techniques. Consequently, the task of process querying is to design those effective and efficient process queries over a wide range of inputs capable of delivering useful insights to the users.

### 2.3 Methods

Processes are properties of dynamic systems, where a *dynamic system* is a system that changes over time, for instance, a process-aware information system or a software system. A *process* is an ordering of events that collectively aim to achieve a goal state, where a *state* is a characteristic of a condition of the system at some point in time. In other words, a state of a process specifies all the information that is relevant to the system at a certain moment in time. An *event* is an instantaneous change of the current state of a system. An event can be distinguished from other events via its attributes and attribute values, for example, a timestamp, event identifier, process instance identifier, or activity that induced the event.

Let  $\mathcal{U}_{an}$  be the universe of *attribute names*. We distinguish three special attribute names  $time, act, rel \in \mathcal{U}_{an}$ , where *time* is the “timestamp”, *act* is the “activity”, and *rel* is the “relationship” attribute name. Let  $\mathcal{U}_{av}$  be the universe of *attribute values*.

**Event.** An *event*  $e$  is a mapping from attribute names to attribute values such that each attribute name that participates in the mapping is related to exactly one attribute value, i.e.,  $e : \mathcal{U}_{an} \rightarrow \mathcal{U}_{av}$ .

By  $\mathcal{E}$ , we denote the universe of events. Similar to attribute names, we identify three special classes of events. By  $\mathcal{E}_{time}$ , we denote the set of all events with timestamps,



i.e.,  $\mathcal{E}_{time} = \{e \in \mathcal{E} \mid time \in dom(e)\}$ . By  $\mathcal{E}_{act}$ , we denote the set of all events with activity names, i.e.,  $\mathcal{E}_{act} = \{e \in \mathcal{E} \mid act \in dom(e)\}$ . Let  $\mathcal{U}_{rel} = \mathcal{P}(\mathcal{E}) \times \mathcal{P}(\mathcal{E})$  be the set of all possible pairs of sets of events, where  $\mathcal{P}(\mathcal{E})$  is the power set of  $\mathcal{E}$ , such that  $\mathcal{U}_{rel}$  are possible attribute values, i.e.,  $\mathcal{U}_{rel} \subset \mathcal{U}_{av}$ . Then,  $\mathcal{E}_{rel}$  is the set of all events that assign a value from  $\mathcal{U}_{rel}$  to its relationship attribute, i.e.,  $\mathcal{E}_{rel} = \{e \in \mathcal{E} \mid rel \in dom(e) \wedge e(rel) \in \mathcal{U}_{rel}\}$ .

For example,  $e = \{(case, 120328), (time, 2020-03-27T03:21:05Z), (act, \text{“Close claim”})\}$  is an event with three attributes. A possible interpretation of these attributes is that event  $e$  belongs to the process with case identifier  $e(case) = 120327$ , was recorded at timestamp  $e(time) = 2019-10-22T11:37:21Z$  (ISO 8601), and was generated by the activity with name  $e(act) = \text{“Close claim”}$ .

**Process.** A process  $\pi$  is a partially ordered set  $(E, \leq)$ , where  $E \subseteq \mathcal{E}$  is a set of events and  $\leq \subseteq E \times E$  is a partial order over  $E$ , i.e.,  $\leq$  is a reflexive, antisymmetric, and transitive binary relation over  $E$ .

A process describes that certain pairs of events are ordered. Note that every pair of related, by a process, events specifies that the first event precedes the second event, while for any two unrelated events, their order is unspecified. It is a common practice to interpret two unordered events as such that may be enabled simultaneously or occur in parallel, refer to [12] for details.

A process that is a total order over a set of events is a *trace*.

**Trace.** A trace  $\tau$  is a process  $(E, <)$ , where  $<$  is a total order over  $E$ .

A *behavior* is a collection of processes in which the same process may appear several times to denote the fact that it can be, or was, observed multiple times.

**Behavior.** A behavior  $b$  is a multiset of processes.

By  $\mathcal{B}$ , we denote the universe of behaviors.

Behaviors can be described in conceptual models. According to Lindland et al. [10], a conceptual model consists of an explicit model component and an implicit model component. The *explicit component* is the set of all statements explicitly made using some modeling language, whereas the *implicit component* is the set of all statements that can be derived from the explicit component using deduction rules of the modeling language.

We refer to a conceptual model that describes behaviors as a *behavior model*. Let  $\mathcal{A} \subset \mathcal{U}_{av}$  be the universe of activities. Let  $\mathcal{U}_{ms}$  be the universe of explicit model statements. Then,  $\mathcal{M} = \mathcal{P}(\mathcal{A}) \times \mathcal{P}(\mathcal{U}_{ms})$  is the universe of activity models, a special class of explicit components of behavior models, where each activity model is a pair composed of a set of activities and a set of model statements that compose the activities into the model. By  $\odot = (\emptyset, \emptyset)$ ,  $\odot \in \mathcal{M}$ , we denote the empty model, the model without activities and statements.

We define four classes of behavior models based on their explicit and implicit components. These four classes are due to the requirements identified in [14].

**Behavior model.** A *behavior model* is a pair  $(M, B)$ , where  $M \in \mathcal{M}$  is an activity model and  $B \subseteq \mathcal{B}$  is a set of behaviors:

- An *event log* is a behavior model  $(\odot, \{b\})$ , where  $b \in \mathcal{B}$  is a finite multiset of finite traces over  $\mathcal{E}_{act}$ , i.e., the activity model is empty and only one behavior is specified.
- A *simulation model* is a behavior model  $(M, \{b\})$ , where  $M = (A, S) \in \mathcal{M}$  is a non-empty model and  $b \in \mathcal{B}$  is a finite multiset of finite processes over  $\mathcal{E}_{act}$  such that for every event  $e$  in a process in  $b$  it holds that  $e(act) \in A$ .
- A *process model* is a behavior model  $(M, B)$ , where  $M = (A, S) \in \mathcal{M}$  is a non-empty model and every  $b \in B$  is a set of processes over  $\mathcal{E}_{act} \setminus \mathcal{E}_{time}$  such that for every event  $e$  in a process in  $b \in B$  it holds that  $e(act) \in A$ .
- A *correlation model* is a behavior model  $(M, B)$ , where every  $b \in B$  is a multiset of processes over  $\mathcal{E}_{rel}$  such that if  $M = (A, S)$  is a non-empty model, then for every event  $e$  in a process in  $b \in B$ , it holds that  $act \in dom(e)$  and  $e(act) \in A$ .

Behavior models are immense information resources. A behavior model can characterize a dynamic system by describing potentially an infinite collection of processes it supports and suggesting the ways to lead the system to possible states that are not bounded by any finite collection of states [2].

We say that  $M$  and  $B$  are, respectively, the *explicit* component and the *implicit* component of the behavior model  $(M, B)$ . That is,  $M$  models behaviors  $B$ . We also say that a behavior model  $(M, \emptyset)$  is *informal*, i.e., the implicit component of an informal model is empty to indicate that no implicit model statement can be deduced from  $M$ . A behavior model  $(M, \{b\})$ , where  $b \in \mathcal{B}$ , is *formal*. An explicit component of a formal behavior model induces one behavior, i.e., all the implicit model statements are deduced from  $M$  deterministically to define one behavior. Finally, a behavior model  $(M, B)$ , where  $|B| > 1$ , is *semi-formal*, i.e., an explicit component of a semi-formal behavior model can be interpreted as one of the behaviors in  $B$  reflecting that the deduction rules of the modeling language used to construct the explicit model are nondeterministic.

An event log is a collection of traces induced by some unknown explicit component of a behavior model; hence, the empty activity model is used as the first element in the pair that defines an event log. Each trace in a log describes some observed process execution. To reflect that the same trace can be observed multiple times, the implicit component of an event log contains a multiset of traces. The multiplicity of a trace in this multiset denotes the number of times this trace was observed. One of the core problems studied in process mining is automatically discovering the explicit component of a behavior model that induced a given log [19]. To support this use case, every event in a trace is required to have the *act* attribute. A simulation model is an activity model together with a finite imitation of its operations in the real-world [3]. The implicit component of a simulation model contains a fraction of behavior that can be induced by the explicit component, which constitutes the behavior imitated during some simulation exercise. To allow traceability, each event of the implicit component has the *act* attribute that refers to

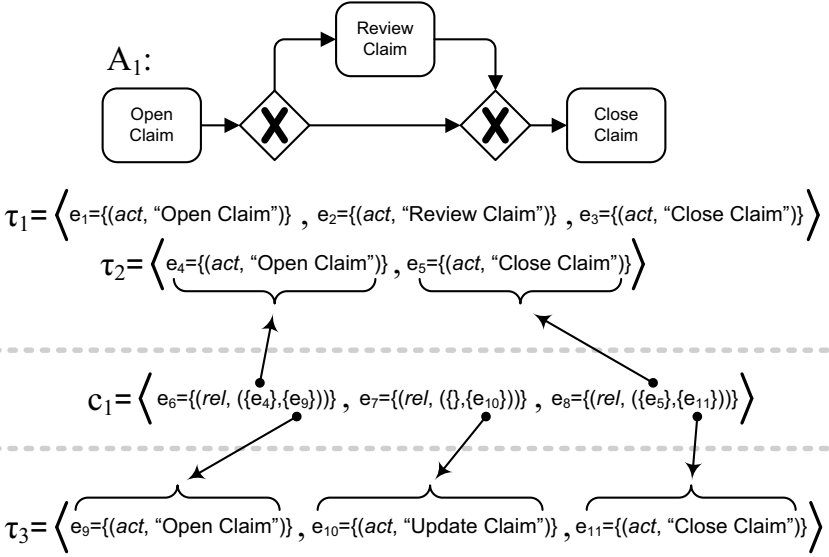


Fig. 1 Example behavior models

the activity that induced the event. A process model is an activity model together with a set of all possible behaviors that can be deduced from the statements the activity model is composed of [16, 21]. Note that a behavior induced by an explicit component of a process model can be infinite, for example, due to some cyclic process dependencies. To reflect the fact that events in the implicit components of process models are envisioned and were not observed in the real-world, they do not have timestamps. Finally, a *correlation model* is a behavior model in which every event specifies a relation between two sets of events. Correlation models, for example alignments [9, 20], describe correspondences and discrepancies between the events in two compared processes. Thus, each event in the implicit component of a correlation model uses the *rel* attribute to specify the matching events from two compared processes.

The top of Fig. 1 shows a process model with activity model  $A_1$  as the explicit component, captured in BPMN. According to BPMN semantics, the diagram describes two process instances  $\tau_1$  and  $\tau_2$ , also shown in the figure. Hence, the process model is defined by the pair  $(A_1, \{\tau_1, \tau_2\})$ . The bottom of Fig. 1 shows trace  $\tau_3$ , which is composed of three events  $e_9$ ,  $e_{10}$ , and  $e_{11}$ , that describes a process that starts with activity “Open Claim”, followed by activity “Update Claim”, and concluded by activity “Close Claim”. The pair  $L = (\odot, \{\tau_1^5, \tau_3^2\})$  specifies a sample event log. Event log  $L$  specifies that trace  $\tau_1$  was observed five times, while trace  $\tau_3$  was observed two times. Note that this log contains traces that cannot be deduced from  $A_1$ . Finally, one can use trace  $c_1$  from Fig. 1 to define a correlation model, for example,  $(\odot, \{c_1\})$ . Trace  $c_1$  relates traces  $\tau_2$  and  $\tau_3$  and specifies that events  $e_4$  and  $e_5$  in trace  $\tau_2$  relate to events  $e_9$  and  $e_{11}$  in trace  $\tau_3$ , respectively, while

event  $e_{10}$  in trace  $\tau_3$  does not correspond to any event in trace  $\tau_2$ . Hence,  $c_1$  captures minimal discrepancies between traces  $\tau_2$  and  $\tau_3$  and corresponds to the concept of optimal alignment between these two traces (assuming the use of the standard cost function) [20].

A process repository is an organized collection of behavior models. Let  $\mathcal{U}_{re}$  be the set of all *repository elements* that are not behavior models, for example, folders for organizing the models, and names and values of metadata attributes of the models.

**Process repository.** A *process repository* is a pair  $(P, R)$ , where  $P$  is a collection of behavior models and  $R \subseteq \mathcal{U}_{re}$  is a set of repository elements.

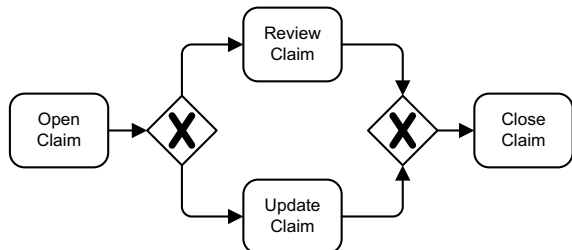
By  $\mathcal{U}_{pr}$  and  $\mathcal{U}_{pq}$ , we denote the universe of *process repositories* and *process queries*, where a process query is an instruction that requests to retrieve artifacts from a process repository or to manipulate a process repository. For example, a process query may capture an instruction to replace a process in one of the behaviors of a process model in a repository with a fresh process. Note that, to ensure consistency between the explicit component and the implicit component, a realization of this query may require updates in the explicit part of the corresponding behavior model.

Finally, a *process querying method* is a computation that given a process repository, and a process query systematically performs the query on the repository. The result of performing a query is, again, a process repository that implements the query on the input repository.

**Process querying method.** A *process querying method*  $m$  is a mapping from pairs, where each pair is composed of a process repository and a process query, to process repositories, i.e., it is a function  $m : \mathcal{U}_{pr} \times \mathcal{U}_{pq} \rightarrow \mathcal{U}_{pr}$ .

For example, a process querying method can support a process query that given a process repository that contains the process model captured in Fig. 1 and updates it to describe trace  $\tau_3$  instead of trace  $\tau_2$  to result in a process repository with a fresh model shown in Fig. 2, which represents all the traces in log  $L$  discussed above.

Fig. 2 A process model



### 3 Process Querying Framework

In [14], we proposed the *Process Querying Framework* (PQF) for devising process querying methods. Schematic visualization of the framework is shown in Fig. 3.<sup>1</sup> We present the framework in Sect. 3.1. Then, Sect. 3.2 discusses decisions that one must take when designing a new process querying method. Finally, Sect. 3.3 elaborates on the challenges associated with the design decisions and how every process querying method is a compromise of the decisions taken.

#### 3.1 Framework

The PQF is an abstract system of components that provide generic functionality and can be selectively replaced to result in a new process querying method. In Fig. 3, rectangles and ovals denote *active components* and *passive components*, respectively; note the use of an ad hoc notation. An active component represents an *action* performed by the process querying method. In turn, a passive component is a (collection of) *data objects*. Passive components serve as inputs and outputs of actions. To show that a passive component is an input to an action, an arc is drawn to point from the component to the action, while an arc that points from an action to a passive component shows that the action produces the component. The dashed lines encode the aggregation relationships. A component that is used as an input to an action contains an adjacent component. For example, a *process repository* is an aggregation of event logs, process models, correlation models, or simulation models, refer to the figure.

The framework is composed of four parts. These parts are responsible for designing process repositories and process queries (“*Model, Simulate, Record and Correlate*”), preparing process queries (“*Prepare*”), executing process queries (“*Execute*”), and interpreting results of the process querying methods (“*Interpret*”). In the figure, the parts are enclosed in areas denoted by the dotted borders. Next, we detail the role of each of these parts.

**Model, Record, Simulate, and Correlate** This part of the PQF is responsible for modeling or creating behavior models and process queries. Behavior models can be acquired in several ways. For example, they can be designed manually by an analyst or constructed automatically using process mining [19] or process querying, as a result of executing a query. Alternatively, an event log can be obtained by recording the traces of an executing IT system. Finally, a behavioral model can be a result of correlating steps of two different processes. Examples of behavior models include

---

<sup>1</sup> © 2017 Elsevier B.V, Fig. 3 is reprinted from Decis. Support Syst. 100, Artem Polyvyanyy, Chun Ouyang, Alistair Barros, Wil M. P. van der Aalst, Process querying: Enabling business intelligence through query-based process analytics, pp 41–56, with permissions from Elsevier.

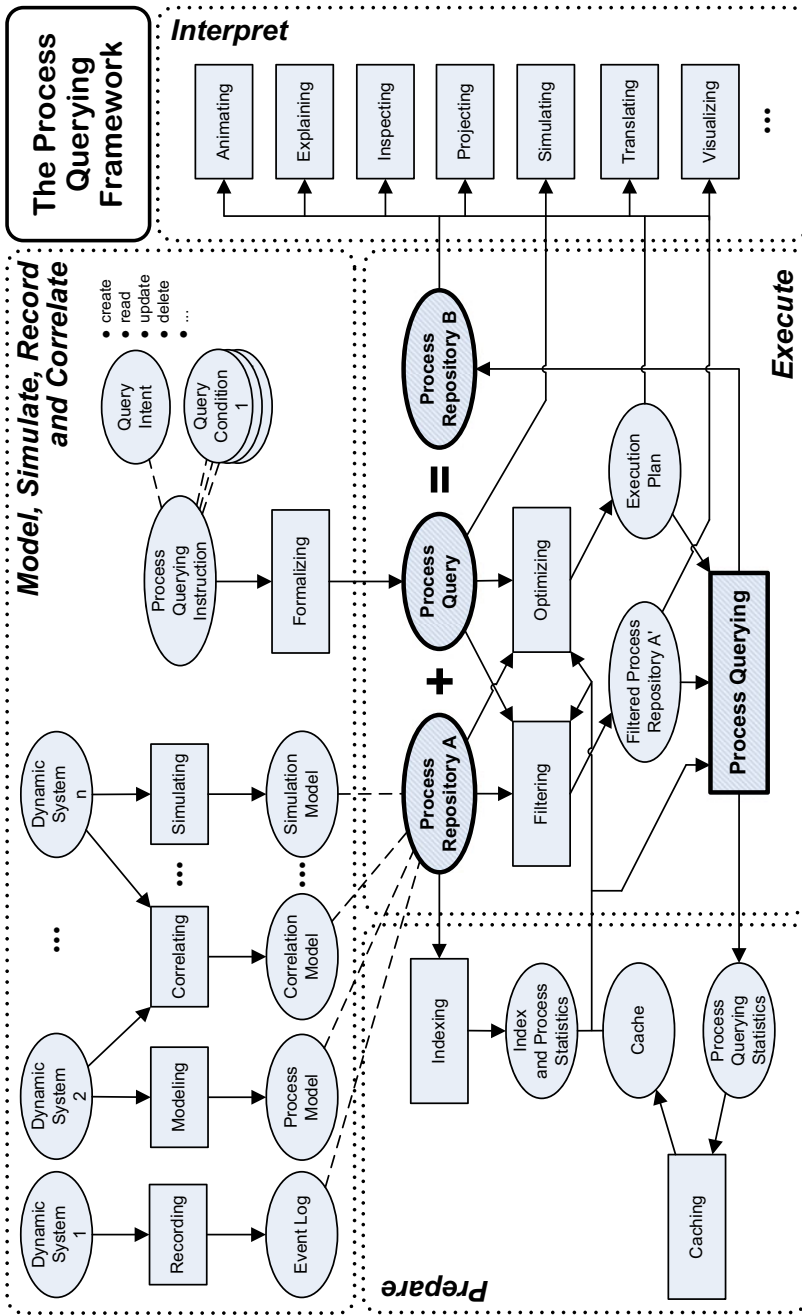


Fig. 3 A schematic view of the Process Querying Framework [14]

process models like computer programs, business process specifications captured in BPMN, YAWL, and BPEL notation, and formal automata and Petri net models, event logs of IT systems [19], and correlation models like alignments [9, 20].

A *process querying instruction* specifies which processes, behaviors, or behavior models should be added to, removed from, replaced in, or retrieved from which processes, behaviors, or behavior models in a process repository. Such an instruction is composed of a *query intent* that specifies the aim of the instruction and a list of *query conditions* that parameterize the intent. The resulting process querying instruction should unambiguously specify how to execute it over the process repository. For example, an instruction can specify to retrieve, or *read*, processes from the repository, while its conditions detail which processes should be included in the query result and which should be left out. We refer to an instruction captured in some machine-readable, i.e., formal, language as a *process query*. The *Formalizing* active component of the framework is responsible for translating process querying instructions into process queries expressed in domain-specific programming languages or some other formalisms.

**Prepare** The *Prepare* part of the framework is responsible for making process repositories ready for efficient querying. In its current version, the framework suggests two methods for preparing for querying, namely indexing and caching. In databases, *indexing* is a technique to construct a data structure, called an index, to efficiently retrieve data records based on some attributes. In computing, *caching* is a technique to store data in a cache so that future requests to that data can be served faster, where the stored data might be the result of an earlier computation.

An index is usually constructed offline and uses additional storage space to maintain an extra copy, or several copies, of the process repository. It is expected that this additional storage will be used to speed up the execution of queries. A process querying can also collect statistics over properties of the repository and its indexes, referred to as *process statistics* in the figure. Process statistics should be used to guide the execution of process queries. For example, a process querying method can proceed by first executing queries over simple models to ensure that initial results are obtained early and proposed to the user.

Caching in process querying can rely on *process querying statistics* to decide which (parts of) query results should be stored for later prompt reuse. The statistics may include aggregate information on the execution of process queries and evaluation of process query conditions, e.g., frequencies of such executions and evaluations. The results of the most frequent queries and evaluated query conditions can then be put in the cache. Next time the user requests to evaluate a query or a condition of a query stored in the cache, its result can be retrieved from the cache instead of recomputed, which is usually more efficient. Caching decisions can rely on *process querying statistics* that aims to keep track of recent frequent query executions and query condition evaluations.

One can rely on other approaches to speed up the evaluation of process queries. The standard approaches that can be explored include parallel computing, e.g., map-reduce, algorithm redesign, e.g., stochastic and dynamic optimization, and hardware

acceleration, e.g., in-memory databases and computing on graphics processing units. Note, however, that such optimization approaches are often inherent to the designs of techniques they optimize. Even though such optimizations are clearly useful, we request that future approaches impose as few restrictions as possible on the querying methods they are intended to be used with.

**Execute** The *Execute* part of the framework is responsible for executing process queries over repositories. It comprises components for filtering process repositories and optimizing and executing process queries.

Filtering is used before executing a query to tag those processes, behaviors, or models in the repository that are known to be irrelevant for the purpose of the process query. Then, the query execution routine can skip tagged artifacts to improve the efficiency of query processing. For instance, if a query requests to select all process models that describe a process with an event that refers to a given activity, it makes no sense to execute the query over models that do not contain the given activity. The filtering is performed by the eponymous active component of the framework that takes the repository and a query as input and produces a *filtered process repository* as output. A filtered repository is a projection of the original repository with some of its parts tagged as irrelevant for the purpose of the query processing. Clearly, to be considered useful, a concrete *Filtering* component must perform filtering decisions more efficiently than executing the query over the same parts of the repository.

The component responsible for query optimization, see the *Optimizing* component in the figure, takes as input a query and all the information that can help produce an efficient execution plan for the query. An *execution plan* is a list of commands that should be carried out to execute the query using the least amount of resources possible. Two types of optimizations are usually distinguished: logical and physical. A *logical* optimization entails reformulating a given query into an equivalent but easier—which usually means a faster to execute—query. A *physical* optimization, in turn, consists of determining efficient means for carrying out commands in a given execution plan.

Finally, the *Process Querying* component takes as input an execution plan of a query, a corresponding filtered repository, as well as available index, process statistics, and cache, and produces a new process repository that implements the instruction specified by the query. As a side effect of executing a query, the component updates the *process querying statistics*. The filtered repository and the execution plan are the *critical inputs* of the querying component, as without these inputs the querying cannot take place, while all the other inputs are optional or can be empty.

**Interpret** An outcome of a process query execution falls into two broad categories: successful or unsuccessful. The successful outcome signifies that the querying instruction captured in the query was successfully implemented in the repository. The latter situation may, for instance, arise when managing vast (possibly infinite) collections of processes described in a process model using scarce (finite) resources of a computer that processes the query.



When a process query fails to execute because of resource limitations, one can adopt at least two strategies to obtain a partial or the full query result. It may be possible to reformulate the original query to give up on the precision of its results. Alternatively, one may be able to optimize the querying method to allow handling the special case of the original query or the class of queries the original query falls into. The standard approaches for managing vast collections of processes include symbolic techniques, such as binary decision diagrams, and abstractions based on the structural model or behavior regularities.

It is often desirable to communicate the query results, successful or unsuccessful, to the user who issued the query. The *Interpret* part of the framework serves this purpose. All the active components of this part have a common goal: to contribute to the user's better comprehension of the querying results. The components listed in Fig. 3 are inspired by the various means for improving comprehension of conceptual models proposed by Lindland et al. [10]. As input, these components take the (filtered) process repository, the query and its execution plan, and the resulting process repository and aim to explain all the differences between the original and resulting repositories and the reasons for the differences.

One can use several techniques to foster the understanding of process querying results. First, a user can understand a concept or phenomenon by inspecting, or reading, it. One can explore various approaches to facilitate the process of reading process query results. For instance, important aspects can be emphasized, while the secondary aspects downplayed. Besides, the inspection activities can be supported by a catalog of predefined explanation notes prepared by process analysts and domain experts. Second, by presenting process querying results diagrammatically rather than in text, their comprehension can be improved. Third, the visual representations of process query results can be further animated, e.g., to demonstrate the dynamics of the processes that comprise the query result. A common approach to animating the dynamics of a process is through a token game that demonstrates the process state evolution superposed over the diagrammatic process model. Fourth, the comprehension of the process querying results can be stimulated by projecting away their parts, allowing the user to focus on a limited number of aspects at a time. Fifth, one can simulate and demonstrate to the user the processes that constitute the query result captured in a static model. Finally, process querying results can be translated into notations that the user is more familiar with. The implementation of these practices is the task of the corresponding active components of the framework.

### 3.2 *Design Decisions*

A design decision is an explicit argumentation for the reasons behind a decision made when designing a system or artifact. When instantiating the PQF to define a concrete process querying method, one needs to take several design decisions imposed by the framework. Next, we discuss three decisions one needs to take when configuring the framework, namely which behavior models, which model

semantics that induce processes, and which process queries should the process querying method support.

**Which Behavior Models to Consider?** An author of a process querying method must decide which behavior models the method will support. Note that a method that addresses querying of event logs will most likely be composed of active and passive components of the PQF that are different from a method for querying correlation models. Besides, the choice of supported formalisms for capturing behavior models restricts the class of supported processes, or languages in the terminology of the theory of computation [17], supported by the process querying method. For instance, if behavior models are restricted to deterministic finite automata, the class of processes described by the models is limited to the class of regular languages [17, 18].

**Which Processes to Consider?** A behavior model can be interpreted as such that describes several behaviors, each induced by a different model semantics criterion. The choice of a semantics criterion determines the correspondence between the model and the collection of processes associated with this model. For instance, a process model can be interpreted according to the finite, infinite, or a fair process semantics. According to the finite process semantics, a model describes processes that lead to terminal goal states. In contrast, an infinite process semantics accommodates processes that never terminate, i.e., processes in which after every event there exists some next event that gets performed. A process in which, from some state onward, an event can get enabled for execution over and over again but never gets executed is an *unfair* process. A not unfair process, as per the stated principle, is a *strongly fair* process [8]. A fair process can be finite or infinite. There are different fairness criteria for processes. Several of them, including the strong fairness criterion, are discussed in [1]. The choice of the correspondence between models and collections of processes they are associated with defines the problem space of the process querying method, as it identifies the processes to consider when executing queries.

**Which Process Queries to Consider?** An author of a process querying method must decide which queries the method will support. The design of a process query consists of two sub-tasks of choosing the *intent* of the query and, subsequently, fixing its *conditions*. The choice of supported process queries determines the expressiveness of the corresponding process querying method, i.e., it defines the ability of the method to describe and solve various problems for managing process repositories. For example, a process querying method can support queries with the intent to *read* process-related information from the repository, i.e., to *retrieve* processes for which specific conditions hold. Alternatively, one can envision a process querying method that supports queries with intents that address all the CRUD operations over models, behaviors, or processes. To specify process queries formally, one can provide formal descriptions of their abstract syntax, concrete syntax, and notation [11].

### 3.3 *Challenges and Compromise*

The design decisions taken when instantiating the PQF into a concrete process querying method inevitably lead to challenges associated with their realization. Next, in Sect. 3.3.1, we discuss three challenges associated with the design decisions discussed in Sect. 3.2. After discussing the challenges, in Sect. 3.3.2, we conclude that every process querying method is a *compromise* between specific solutions taken to address the challenges.

#### 3.3.1 Challenges

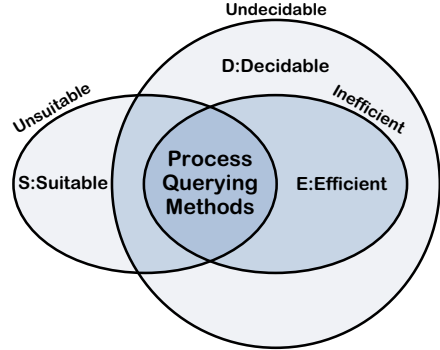
When implementing a process querying method, one inevitably faces three challenges: decidability, efficiency, and usefulness of the supported process queries.

**Decidability** Process queries must be decidable. In other words, they must be solvable by algorithms on a wide range of inputs. Indeed, a process query that cannot be computed is of no help to the user. The decidability requirement poses a significant challenge, as certain process management problems are known to be undecidable over specific classes of inputs. For example, process queries can be expressed in terms of temporal logic formulas [2, 15]. However, temporal logic formulas can be undecidable over some classes of process models [6, 7].

**Efficiency** Process querying aims to provide valuable insights into processes managed by organizations. As part of this premise, process querying should foster the learning of processes, behaviors, and behavior models contained in the repository by the novice users of the repository. In other words, it should support exploratory querying [23]. However, exploratory querying requires techniques capable of executing queries close to real time. Therefore, another challenge of process querying is to develop process queries that can be computed efficiently, that is, fast and using small memory footprints. One can measure the efficiency of the process queries using well-known techniques in computational complexity theory, which study resources, like computation time and storage space, required to solve computational problems with the goal of proposing solutions to the problems that use less resources.

**Suitability** Process querying methods should offer a great variety of concepts and principles to capture and exercise in the context of process querying. Thus, the third challenge of process querying is concerned with achieving expressiveness in terms of capturing all the suitable (appropriate for the purpose envisioned by the users) process queries that specify instructions for managing process repositories. Authors of process querying methods should strive to propose designs that support all the useful (as perceived by the users) process queries. The suitability of process querying methods can be assessed empirically or, similar to [5], by identifying common reoccurring patterns in specifications of process querying problems.

**Fig. 4** Process querying compromise



### 3.3.2 Compromise

A process querying method is identified by a collection of process queries it supports. The selection of queries to support is driven by the considerations of decidability, efficiency, and suitability of queries. As these considerations often forbid the method to support all the desired queries, we refer to the phase of selecting which queries to support and not support as the *process querying compromise*.

The process querying compromise can be formalized as follows. Let  $D$  be the set of all decidable process queries. Some decidable queries can be computed efficiently; note that, in general, the decidability of certain process queries can be unknown. Let  $E \subseteq D$  be the set of all process queries that are not only computable but are also efficiently computable. Finally, let  $S$  be the set of all process queries that the users perceive as suitable. Then, the queries in  $E \cap S$  are the queries that one should aim to support via process querying methods. Figure 4 demonstrates the relations between sets  $D$ ,  $E$ , and  $S$  visually. In an ideal situation, it should hold that  $S \subseteq E$ , i.e., all the suitable queries are computable using some efficient methods. However, in practice, it is often impossible to fulfill the requirement of  $S \subseteq E$ , or even  $S \subseteq D$ . Then, one can strive to improve the efficiency of the techniques for computing queries in  $(S \cap D) \setminus E$ , which are the decidable and practically relevant queries for which no efficient computation procedure is known.

The existence of such compromise differentiates process querying from data querying. Note that data queries usually operate over finite datasets, making it possible to implement querying using, maybe not always efficient, but certainly effective methods.

## 4 Conclusion

This chapter presents and discusses the problem of process querying. Process querying aims to coordinate the efforts invested in the design, implementation, and application of techniques, methods, and tools for retrieving and manipulating

models of processes, and the related resources, information, and data. Consequently, process querying supports centralized activities that improve process querying practices and suppresses reinventions of such practices in different contexts and variations. The chapter also presents an abstract framework for designing and implementing process querying methods. The framework consists of abstract components, each with a dedicated role and well-defined interface, which, when instantiated and integrated, result in a concrete process querying method. Finally, the chapter argues that every process querying method defines a compromise between efficiently decidable and practically relevant queries, which is unavoidably associated with challenges for designing and implementing such methods.

The concept of process querying emerged from the observations of theory and practice in the research discipline of BPM and relates to other process-centric research fields like software engineering, information systems engineering, and computing. We envisage future applications, adaptations, and improvements of process querying techniques contributed from within these fields. Future endeavors in process querying will contribute to understanding the process querying compromise, including which queries are practically relevant for the users to justify the efforts for their design and use in practice.

**Acknowledgments** Artem Polyvyanyy wants to thank Chun Ouyang, Alistair Barros, and Wil van der Aalst with whom he worked together to shape the concept of process querying and to design and validate the Process Querying Framework.

## References

1. Apt, K.R., Francez, N., Katz, S.: Appraising fairness in languages for distributed programming. *Distrib. Comput.* **2**(4), 226–241 (1988)
2. Baier, C., Katoen, J.P.: *Principles of Model Checking*. MIT Press (2008)
3. Banks, J., II, J.S.C., Nelson, B.L., Nicol, D.M.: *Discrete-Event System Simulation*, 5th edn. Pearson Education (2010)
4. Dumas, M., Rosa, M.L., Mendling, J., Reijers, H.A.: *Fundamentals of Business Process Management*, Second Edition. Springer (2018)
5. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Patterns in property specifications for finite-state verification. In: ICSE, pp. 411–420. ACM (1999)
6. Esparza, J.: Decidability and complexity of Petri net problems—an introduction. In: *Petri Nets, LNCS*, vol. 1491, pp. 374–428. Springer (1996)
7. Esparza, J., Nielsen, M.: Decidability issues for Petri nets—a survey. *Bull. EATCS* **52**, 244–262 (1994)
8. Kindler, E., van der Aalst, W.M.P.: Liveness, fairness, and recurrence in Petri nets. *Inf. Process. Lett.* **70**(6), 269–274 (1999)
9. Leemans, S.J., van der Aalst, W.M., Brockhoff, T., Polyvyanyy, A.: Stochastic process mining: Earth movers’ stochastic conformance. *Information Systems*, 101724 (2021). <https://doi.org/10.1016/j.is.2021.101724>
10. Lindland, O.I., Sindre, G., Sølvsberg, A.: Understanding quality in conceptual modeling. *IEEE Softw.* **11**(2), 42–49 (1994)
11. Meyer, B.: *Introduction to the Theory of Programming Languages*. Prentice-Hall (1990)

12. Nielsen, M., Plotkin, G.D., Winskel, G.: Petri nets, event structures and domains, Part I. *Theor. Comput. Sci.* **13**, 85–108 (1981)
13. Polyvyanyy, A.: Business process querying. In: *Encyclopedia of Big Data Technologies*. Springer (2019)
14. Polyvyanyy, A., Ouyang, C., Barros, A., van der Aalst, W.M.P.: Process querying: Enabling business intelligence through query-based process analytics. *Decis. Support Syst.* **100**, 41–56 (2017). <https://doi.org/10.1016/j.dss.2017.04.011>
15. Reichert, M., Weber, B.: Enabling Flexibility in Process-Aware Information Systems, chap. *Business Process Compliance*, pp. 297–317. Springer (2012)
16. Reisig, W.: *Understanding Petri Nets—Modeling Techniques, Analysis Methods, Case Studies*. Springer (2013)
17. Sipser, M.: *Introduction to the Theory of Computation*, 3rd edn. Cengage Learning (2012)
18. Valk, R., Vidal-Naquet, G.: Petri nets and regular languages. *J. Comput. Syst. Sci.* **23**(3), 299–325 (1981)
19. van der Aalst, W.M.P.: *Process Mining—Data Science in Action*, 2nd edn. Springer (2016)
20. van der Aalst, W.M.P., Adriansyah, A., van Dongen, B.F.: Replaying history on process models for conformance checking and performance analysis. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery* **2**(2), 182–192 (2012). <https://doi.org/10.1002/widm.1045>
21. van der Aalst, W.M.P., Stahl, C.: *Modeling Business Processes—A Petri Net-Oriented Approach*. MIT Press (2011)
22. Weske, M.: *Business Process Management - Concepts, Languages, Architectures*, 3rd edn. Springer (2019)
23. White, R.W., Roth, R.A.: *Exploratory Search: Beyond the Query-Response Paradigm*. Morgan & Claypool Publishers (2009)

# **Part I**

## **Event Log Querying**

# BP-SPARQL: A Query Language for Summarizing and Analyzing Big Process Data



Amin Beheshti, Boualem Benatallah, Hamid Reza Motahari-Nezhad, Samira Ghodratnama, and Farhad Amouzgar

**Abstract** In modern enterprises, business processes (BPs) are realized over a mix of workflows, IT systems, Web services, and direct collaborations of people. Accordingly, process data (i.e., BP execution data such as logs containing events, interaction messages, and other process artifacts) are scattered across several systems and data sources and increasingly show all typical properties of the Big Data. Understanding the execution of process data is challenging as key business insights remain hidden in the interactions among process entities: most objects are interconnected, forming complex heterogeneous but often semi-structured networks. In the context of business processes, we consider the Big data problem as a massive number of interconnected data islands from personal, shared, and business data. We present a framework to model process data as graphs, i.e., process graph, and present abstractions to summarize the process graph and to discover concept hierarchies for entities based on both data objects and their interactions in process graphs. We present a language, namely BP-SPARQL, for the explorative querying and understanding of process graphs from various user perspectives. We have implemented a scalable architecture for querying, exploration, and analysis of process graphs. We report on experiments performed on both synthetic and real-world datasets that show the viability and efficiency of the approach.

---

A. Beheshti (✉) · S. Ghodratnama · F. Amouzgar  
Macquarie University, Sydney, NSW, Australia  
e-mail: [amin.beheshti@mq.edu.au](mailto:amin.beheshti@mq.edu.au); [samira.ghodratnama@hdr.mq.edu.au](mailto:samira.ghodratnama@hdr.mq.edu.au);  
[farhad.amouzgar@hdr.mq.edu.au](mailto:farhad.amouzgar@hdr.mq.edu.au)

B. Benatallah  
University of New South Wales, Sydney, NSW, Australia  
e-mail: [boualem@cse.unsw.edu.au](mailto:boualem@cse.unsw.edu.au)

H. R. Motahari-Nezhad  
EY AI Lab, Santa Clara, CA, USA  
e-mail: [hamid.motahari@ey.com](mailto:hamid.motahari@ey.com)



## 1 Introduction

A business process is a set of coordinated tasks and activities, carried out manually or automatically, to achieve a business objective or goal [27]. In modern enterprises, business processes (BPs) are realized over a mix of workflows, IT systems, Web services, and direct collaborations of people. In such information systems, business process analysis over a wide range of systems, services, and software (which implement the actual business processes of enterprises) is required. This is challenging as in today's knowledge-, service-, and cloud-based economy the information about process execution is scattered across several systems and data sources. Consequently, process logs increasingly come to show all typical properties of the Big data [20]: wide physical distribution, diversity of formats, nonstandard data models, and independently managed and heterogeneous semantics. We use the term *process data* to refer to such large hybrid collections of heterogeneous and partially unstructured process-related data.

Understanding process data requires scalable and process-aware methods to support querying, exploration, and analysis of the process data in the enterprise because (i) with the large volume of data and their constant growth, the process data analysis and querying method should be able to scale well and (ii) the process data analysis and querying method should enable users to express their needs using process-level abstractions. Besides the need to support process-level abstractions in process data analysis scenarios, the other challenge is the need for scalable analysis techniques to support Big data analysis. Similar to Big Data processing platforms [30], such analysis and querying methods should offer automatic parallelization and distribution of large-scale computations, combined with techniques that achieve high performance on large clusters of commodity PCs, e.g., cloud-based infrastructure, and be designed to meet the challenges of process data representation that capture the relationships among data.

In this chapter, we present a summary of our previous work [5, 8, 10, 13] in organizing, querying, and analyzing business processes' data. We introduce BP-SPARQL, a query language for summarizing and analyzing process data. BP-SPARQL supports a graph-based representation of data relationships and enables exploring, analyzing, and querying process data and their relationships by superimposing process abstractions over an entity-relationship graph, formed over entities in process-related repositories. This will provide analysts with process-related entities (such as process event, business artifacts, and actors), abstractions (such as case, process instances graph, and process model), and functions (such as correlation condition discovery, regular expressions, and process discovery algorithms) as first-class concepts and operations. BP-SPARQL supports various querying needs such as entity-level (artifacts, events, and activities), summarization (OLAP Style, Group Style, and Partition Style), relationship (Regular Expression, Path Condition, and Path Node), metadata (Time and Provenance), and user-defined queries.

The remainder of this chapter is organized as follows. In Sect. 2, we present the background and contributions overview. We introduce the process graph model in

Sect. 3. Section 4 presents the abstractions used to summarize the process data. In Sect. 5, we present the query framework, and in Sect. 6 we present MapReduce techniques to scale the analysis. In Sect. 7, we describe the implementation and the evaluation experiments. Finally, we position our approach within the Process Querying Framework in Sect. 8, before concluding the chapter in Sect. 9.

## 2 Background and Contributions Overview

The problem of understanding the behavior of information systems as well as the processes and services they support has become a priority in medium and large enterprises. This is demonstrated by the proliferation of tools for the analysis of process executions, system interactions, and system dependencies and by recent research work in process data warehousing and process discovery. Indeed, the adoption of business process intelligence techniques for process improvement is the primary concern for medium and large companies. In this context, identifying business needs and determining solutions to business problems require the analysis of business process data: this enables discovering useful information, suggesting conclusions, and supporting decision-making for enterprises.

In order to understand available process data (events, business artifacts, data records in databases, etc.) in the context of process execution, we need to represent them, understand their relationships, and enable the analysis of those relationships from the process execution perspective. To achieve this, it is possible to represent process-related data as entities and any relationships among them (e.g., event relationships in process logs with artifacts, etc.) in entity-relationship graphs. In this context, business analytics can facilitate the analysis of process data in a detailed and intelligent way through describing the applications of analysis, data, and systematic reasoning [10]. Consequently, an analyst can gather more complete insights using data-driven techniques such as modeling, summarization, and filtering.

**Motivating Scenario** Modern business processes (BPs) are rarely supported by a single, centralized workflow engine. Instead, BPs are realized using a number of autonomous systems, Web services, and collaboration of people. As an example, consider the banking industry scenario. Recently, there is a movement happening in the banking industry to modernize core systems for providing solutions to account management, deposits, loans, credit cards, and the like. The goal is to provide flexibility to quickly and efficiently respond to new business requirements. In order to understand the process analysis challenges, let us consider a real-world case in the loan scenario, where Adam (a customer) plans to buy a property. He needs to find a lending bank. He can use various crowdsourcing services (e.g., Amazon Mechanical Turk<sup>1</sup>) or visit mortgage bank to find a proper bank. Then, he needs to contact the

---

<sup>1</sup> <https://www.mturk.com/>.

bank through one of many channels and start the loan pre-approval process. After that, he needs to visit various Websites or real estate services to find a property. Meanwhile, he can use social Websites (e.g., Facebook or Twitter) to socialize the problem of buying a property and ask for others' opinions to find a proper suburb. After finding a property, he needs to choose a solicitor to start the process of buying the property. Lots of other processes can be executed in between. For example, the bank may outsource the process of evaluating the property or analyzing Adam's income to other companies.

In this scenario, the data relevant to the business process of bank is scattered across multiple systems, and in many situations, stakeholders can be aware of processes, but they are not able to track or understand them: it is important to maintain the vital signs of customers by analyzing the process data. This task is challenging as (i) a massive number of interconnected data islands (from personal, shared, and business data) need to be processed, (ii) processing this data requires scalable methods, and (iii) analyzing this data depends on the perspective of the process analyst, e.g., “*Where is loan-order #756? What happened to it? Is it blocked? Give me all the documents and information related to the processing of loan-order #756? What is the typical path of loan orders? What is the process flow for it? What are the dependencies between loan applications  $A_1$  and  $A_2$ ? How much time and resources are spent in processing loan orders that are eventually rejected? Can I replace  $X$  (i.e., a service or a person) by  $Y$ ? Where data came from? How it was generated? Who was involved in processing file  $X$ ? At which stage do loan orders get rejected? How many loan orders are rejected in the time period between  $\tau_1$  and  $\tau_2$ ? What is the average time spent on processing loan orders? Where do delays occur? Etc.*”

Answering these questions in today's knowledge-, service-, and cloud-based economy is challenging as the information about process execution is scattered across several systems and data sources. Therefore, businesses need to manage unstructured, data-intensive, and knowledge-driven processes rather than well predefined business processes. Under such conditions, organizing, querying, and analyzing process data becomes of a great practical value but clearly a very challenging task as well. In the following, we provide an overview of the main contributions of this chapter:

(1) *Process-aware abstractions for querying and representing process data* [5, 8]:

We introduce a graph-based model to represent all process-related data as entities and relationships among those entities in an entity-relationship graph (ER graph). To enable analyzing process entity-relationship graph directly, still in a process-aware context, we define two main abstractions for summarizing the process data (modeled as an ER graph): *folder* nodes (a container for the results of a query that returns a collection of related process entities) and *path* nodes (a container for the results of a query that returns a collection of related paths found in the entity-relationship graph). We present other extensions of folder and path nodes including *timed* folder nodes and also process metadata queries to support process analysis needs in contexts such as process entity

provenance [7] and artifact versioning. These summarization abstractions and functions offer a comprehensive set that covers most prominent needs of querying process-related data from different systems and services.

- (2) *Summarizing process data* [10]: We introduce a framework and a set of methods to support scalable graph-based OLAP (online analytical processing) queries over process execution data. The goal is to facilitate the analytics over the ER graph through summarizing the process graph and providing multiple views at different granularities. To achieve this goal, we present a model for process OLAP (P-OLAP) and define OLAP specific abstractions in process context such as process cubes, dimensions, and cells. We present a MapReduce-based graph processing engine, to support Big data analytics over process data. We identify useful machine learning algorithms [2] and provide an external algorithm controller to enable summarizing the process data (modeled as an ER graph), by extracting complex data structures such as time series, hierarchies, patterns, and subgraphs. We define a set of domain-specific abstractions and functions such as process event, process instance, events correlation condition, process discovery algorithm, correlation condition discovery algorithm, and regular expression to summarize the process data and to enable the querying and analysis of relationships among process-related entities.
- (3) *Scalable process data analysis and querying methods* [13]. In order to support the scalable exploration and analysis of process data, we present a domain-specific language, namely BP-SPARQL, that supports the querying of process data (modeled as an ER graph) using the abovementioned process-level abstractions. BP-SPARQL translates process-level queries into graph-level abstractions and queries. To support the scalable and efficient analysis over process data, BP-SPARQL is implemented over the MapReduce<sup>2</sup> framework by providing a data mapping layer for the automatic translation of queries into MapReduce operations. For this purpose, we designed and implemented a translation from BP-SPARQL to Hadoop PigLatin [22], where the resulting PigLatin program is translated into a sequence of MapReduce operations and executed in parallel on a Hadoop cluster. The proposed translation offers an easy and efficient way to take advantage of the performance and scalability of Hadoop for the distributed and parallelized execution of BP-SPARQL queries on large graph datasets. BP-SPARQL supports the following query types: (i) entity-level queries: for querying process-related entities, e.g., business artifacts, actors, and activities, (ii) relationship queries: for discovering relationships and patterns among process entities using regular expressions, (iii) summarization queries: these queries allow for analyzing case-based processes to find potential process instances by supporting OLAP Style, Group Style, and Partition Style queries, (iv) metadata queries: for analyzing the evolution of business artifacts and their provenance over time, and (v) user-defined queries.

---

<sup>2</sup> The popular MapReduce [16] scalable data processing framework and its open-source realization Hadoop [28] offer a scalable dataflow programming model.

### 3 Process Abstractions

In this section, we provide a summary of our previous work [5, 6, 8] on modeling process data as an ER graph. To organize the process data, we introduce a graph-based data model. The data model includes entities (e.g., events, artifacts, and actors), their relationships, and abstractions which act as higher level entities to store and browse the results of queries for follow-on analysis. The data model is based on the RDF [24] data representation.

**Definition 3.1 (Entity)** An entity  $N$  is a data object that exists separately and has a unique identity.

Entities can be structured (e.g., customer, bank, and branch) or unstructured (body of an e-mail message). Structured entities are instances of entity types. This entity model offers flexibility when types are unknown and takes advantage of structure when types are known. Specific types of entities include:

- (Entity: Business Artifact) is a digital representation of something, i.e., data object, that exists separately as a single and complete unit and has a unique identity. A business artifact is a *mutable* object, i.e., its attributes (and their values) are able and are likely to change over periods of time. An artifact  $A$  is represented by a set of attributes  $\{a_1, a_2, \dots, a_k\}$ , where  $k$  represents the number of attributes. An artifact may appear in many versions. A *version*  $v$  of a business artifact is an *immutable* copy of the artifact at a certain point in time. A business artifact  $A$  can be represented by a set of versions  $\{v_1, v_2, \dots, v_n\}$ , where  $n$  represents the number of versions. A business artifact can capture its current state as a version and can restore its state by loading it. Each version is represented as a data object that exists separately and has a unique identity. Each version  $v_i$  consists of a snapshot, a list of its parent versions, and metadata, such as commit message, author, owner, or time of creation. In order to represent the history of a business artifact, it is important to create archives containing all previous states of an artifact. The archive allows us to easily answer certain temporal queries such as retrieval of any specific version and finding the history of an artifact.
- (Entity: Actor). An actor  $R$  is an entity acting as a catalyst of an activity, e.g., a person or a piece of software that acts for a user or other programs. A process may have more than one actor enabling, facilitating, controlling, and affecting its execution.
- (Entity: Event). An event is an object representing an activity performed by an actor. An event  $E$  can be presented as the set  $\{R, \tau, D\}$ , where  $R$  is an actor (i.e., a person or device) executing or initiating an activity,  $\tau$  is a timestamp of the event, and  $D$  is a set of data elements recorded with the event (e.g., the size of an order). We assume that each distinct event does not have a temporal duration. For instance, an event may indicate an arrival of a loan request (as an XML document) from a bank branch, an arrival of a credit card purchase order from a business partner, or a completion of a transmission or a transaction.

**Definition 3.2 (Relationship)** A *relationship* is a directed link between a pair of entities, which is associated with a predicate defined on the attributes of entities that

characterizes the relationship. A relationship can be *explicit*, such as “was triggered by” in  $event_1 \xrightarrow{\text{(wasTriggeredBy)}} event_2$  in business processes (BPs) execution log, or *implicit*, such as a relationship between an entity and a larger (composite) entity that can be inferred from the nodes. An entity is related to other entities by time (time-based), content (content-based), or activity (activity-based):

- (Time-Based Relationships). Time is the relationship that orders events, for example,  $event_A$  happened before  $event_B$ . A timestamp is attached to an event where every timestamp value is unique and accurately represents an instant in time. Considering activities  $A_{\tau_1}$  and  $A_{\tau_2}$ , where  $\tau$  is a timestamp of an event,  $A_{\tau_1}$  happened before  $A_{\tau_2}$  if and only if  $\tau_1 < \tau_2$ .
- (Content-Based Relationships). When talking about content, we refer to entity attributes. In process context, we consider content-based relationships as correlation condition-based relationships, where a *correlation condition* [21] is a binary predicate defined over attributes of two entities  $E_x$  and  $E_y$  and denoted by  $\psi(E_x, E_y)$ . This predicate is true when  $E_x$  and  $E_y$  are correlated and false otherwise. A correlation condition  $\psi$  allows to partition an entity-relationship graph into sets of related entities.
- (Activity-Based Relationships). This is a type of relationship between two entities that is established as the result of performing an activity. In this context, an activity can be described by a set of attributes such as (i) What (types of activity), (ii) How (actions such as creation, transformation, or use), (iii) When (the timestamp in which the activity has occurred), (iv) Who (an actor that enables, facilitates, controls, or affects the activity execution), (v) Where (the organization/department where the activity happened), (vi) Which (the system which hosts the activity), and (vii) Why (the goal behind the activity, e.g., fulfillment of a specific phase).

Other types of relationships are discussed in [4]. Figure 1 illustrates a sample process data modeled as an ER graph, illustrating possible relationships among entities in the motivating (banking) scenario. Next, we define RDF triples as a representation of relationships.

**Definition 3.3 (RDF Triple)** The RDF terminology  $T$  is defined as the union of three pairwise disjoint infinite sets of terms: the set  $U$  of URI references, the set  $L$  of literals, and the set  $B$  of blanks. The set  $U \cup L$  of names is called the vocabulary. An RDF triple (subject, predicate, object) is an element of  $(s, p, o) \in (U \cup B) \times U \times T$ , where  $s$  is a subject,  $p$  is a predicate, and  $o$  is an object.

An RDF graph is a finite set of RDF triples. An RDF triple can be viewed as a relationship (an arc) from subject  $s$  to object  $o$ , where predicate  $p$  is used to label the relationship. This is represented as  $s \xrightarrow{(p)} o$ . Next, we define an ER graph as a graph capable of representing process data as entities and relationships among those entities. An ER graph may contain all possible relationships (from Definition 3.2) among its nodes.

**Definition 3.4** An entity-relationship (ER) graph  $G = (V, E)$  is a directed graph with no directed cycles, where  $V$  is a set of nodes and  $E \subseteq (V \times V)$  is a set of

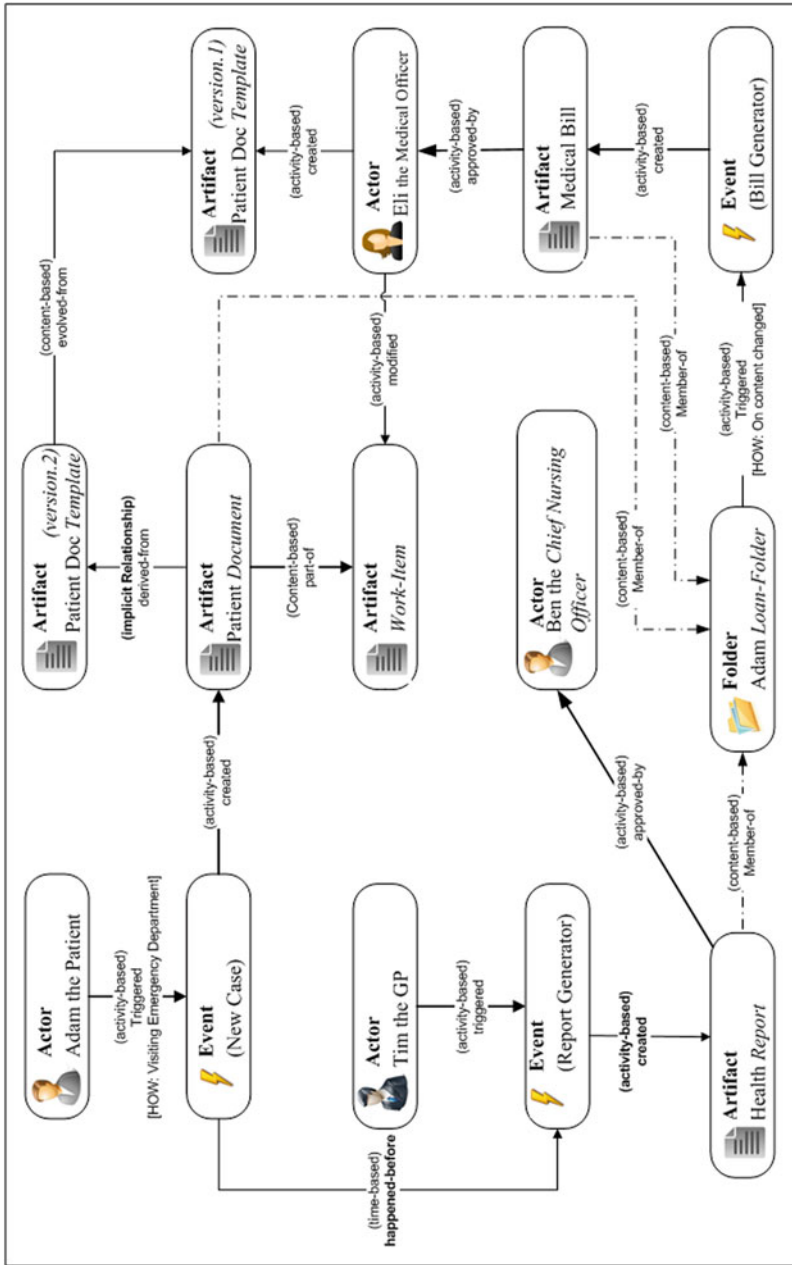


Fig. 1 A sample graph of relationships among entities in the motivating banking scenario presented in Sect. 2

ordered pairs called edges. An ER graph  $G$  is modeled using the RDF data model to make statements about resources (in expressions of the form subject–predicate–object, known as RDF triples), where a resource in an ER graph is defined as follows: (i) the sets  $V_G$  and  $E_G$  are resources and (ii) the set of ER graphs is closed under intersection, union, and set difference: let  $G_1$  and  $G_2$  be two ER graphs, then  $G_1 \cup G_2$ ,  $G_1 \cap G_2$ , and  $G_1 - G_2$  are ER graphs.

## 4 Summarizing Big Process Data

In this section, we provide a summary of our previous work [6, 10] on summarizing the process data. We introduce a framework and a set of methods to support scalable graph-based OLAP (online analytical processing) analytics over process execution data.

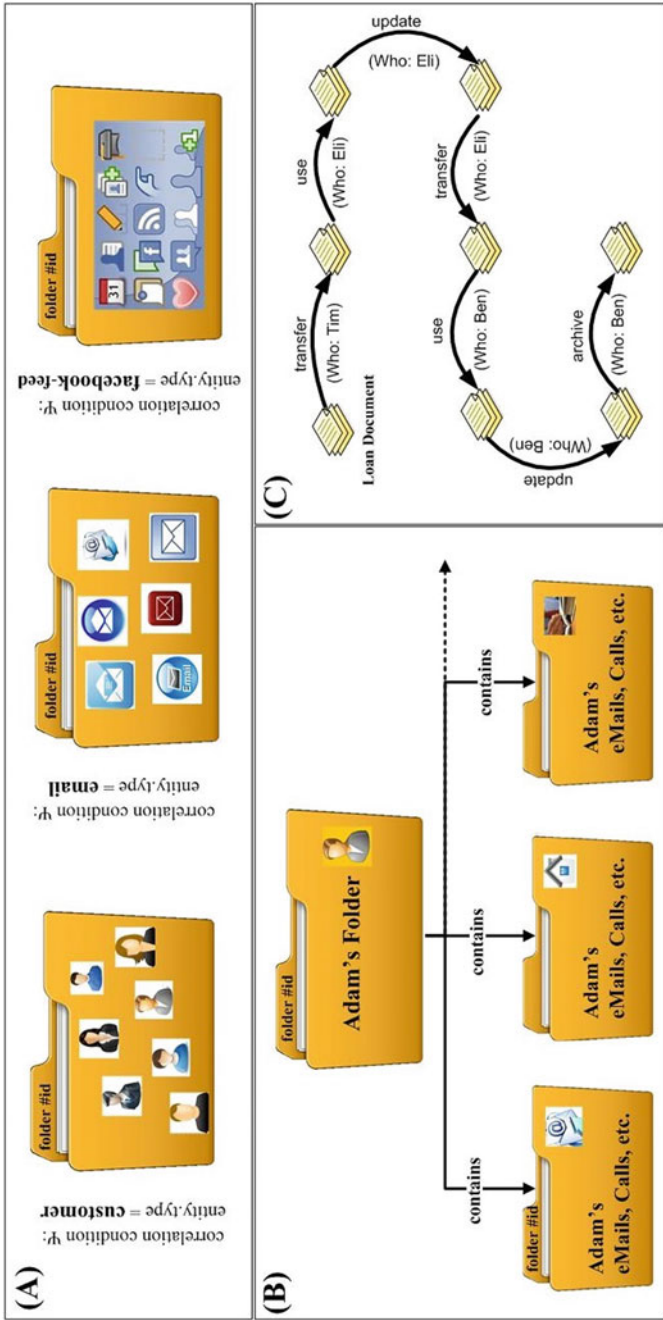
We present two types of queries to summarize the process data (modeled as ER graphs), namely “Correlation Condition” and “Regular Expression”:

- **Correlation Condition** is a binary predicate defined on the attributes of events that allows to identify whether two or more events are potentially related to the same execution instance of a process [5]. In particular, a correlation condition takes ER graph and a predicate (as input), applies an algorithm (e.g., the one introduced in our previous work [21]) to find the set of correlated events for that condition, and returns a set of ER subgraphs  $\{G'_1, G'_2, \dots, G'_k\}$  representing correlated events. Formally,  $G' = (V', E')$  is an ER subgraph of an ER graph  $G=(V, E)$  iff  $V' \subseteq V$  and  $E' \subseteq E$  and  $((v_1, v_2) \in E' \rightarrow \{v_1, v_2\} \subseteq V')$ .
- **Regular Expression** is a function which represents a relation between ER graph  $G$  and a set of paths  $\{P_1, P_2, \dots, P_m\}$ , where a *path*  $P$  is a transitive relationship between entities capturing sequences of edges from the start entity to the end entity. A path can be represented as a sequence of RDF triples, where the object of each triple in the sequence coincides with the subject of its immediate successor. We developed a regular expression processor which supports optional elements, loops, alternation, and grouping [9].

An example path is illustrated in Fig. 2c. This path depicts that the loan document was transferred by Tim, used and transferred by Eli to Ben, who updated the document and archived it. Paths (defined by regular expressions) are written by domain experts, and BP-SPARQL takes care of the processing and optimization that is needed for efficient crawling, analyzing, and querying of the ER graph. In particular, a regular expression  $RE$  is an operator that specifies a search pattern and can result in a set of paths. Related paths can be stored in a path node [5], i.e., a container for a collection of related paths found in the entity-relationship graph.

**Definition 4.1 (Path Node [5])** A path node is a place holder for a set of related paths: these paths are the result of a given query that requires grouping graph patterns in a certain way. We define a path node as a triple of  $(V_{start}, V_{end}, RE)$  in which  $V_{start}$  is the starting node,  $V_{end}$  is the ending node, and  $RE$  is a regular





**Fig. 2** An example of a basic folder (a), a high-level folder (b), and a path (c)

expression. We use existing reachability approaches to verify whether an entity is reachable from another entity in the graph. Path nodes can be timed. A *timed-path* node [8] is defined as a timed container for a set of related entities which are connected through *transitive* relationships, e.g., it is able to trace the evolution of patterns among entities over time as new entities and relationships are added over time.

Besides applying queries to the relationships in ER graphs (where the result will be a set of related paths and stored in path nodes), queries can be applied to the content of the entities in the ER graph (without considering their relationships), i.e., the query may define criteria beyond existing relationships among entities to allow for discovering other relationships. In this case, the result will be a set of correlated entities and possible relationships among them stored in *folder nodes*.

**Definition 4.2 (Folder Node)** A folder node [5] contains a set of correlated entities that are the result of applying a function (e.g., Correlation Condition) on entity attributes. The folder concept is akin to that of a database view, defined on a graph. However, a folder is part of the graph and creates a higher level node that other queries could be executed on. *Basic folders* can be used to aggregate the same entity types, e.g., related e-mails, Facebook feeds, or activities. Figure 2a illustrates a set of basic folders. *High-level folders*, illustrated in Fig. 2b, can be used to create and query a set of related folders with relationships at higher levels of abstraction. A folder may have a set of attributes that describes it. Folder nodes can be timed. Timed folders [8] document the evolution of folder node by adapting a monitoring code snippet. New members can be added to or removed from a timed folder over time.

Now, to support scalable graph-based OLAP analytics over process data, we define a mapping from process data into a graph model.

**Definition 4.3 (Process Instance)** A Process Instance can be defined as a path  $P$  in which the nodes in  $P$  are of type “event” and are in chronological order, and all the relationships in the path are of type activity-based relationships.

A collection of (disconnected) ER graphs, each representing a process instance, can be considered as a process instance graph and defined as follows:

**Definition 4.4 (Process Instances Graph)** A Process Instances Graph is a set of related process instances. A Process Instances Graph is the result of a Correlation Condition query (stored in a folder node) or a Regular Expression query (stored in a path node).

In process analysis context, another common function applied on ER graphs is *process discovery*. In particular, process mining techniques and tools offer a wide range of algorithms for discovering knowledge from process execution data. In BP-SPARQL, we have identified many useful machine learning algorithms [2] and provide an external algorithm controller to enable summarizing large graphs, by extracting complex data structures such as time series, hierarchies, patterns, and subgraphs. Figure 3 illustrates a taxonomy of machine learning algorithms

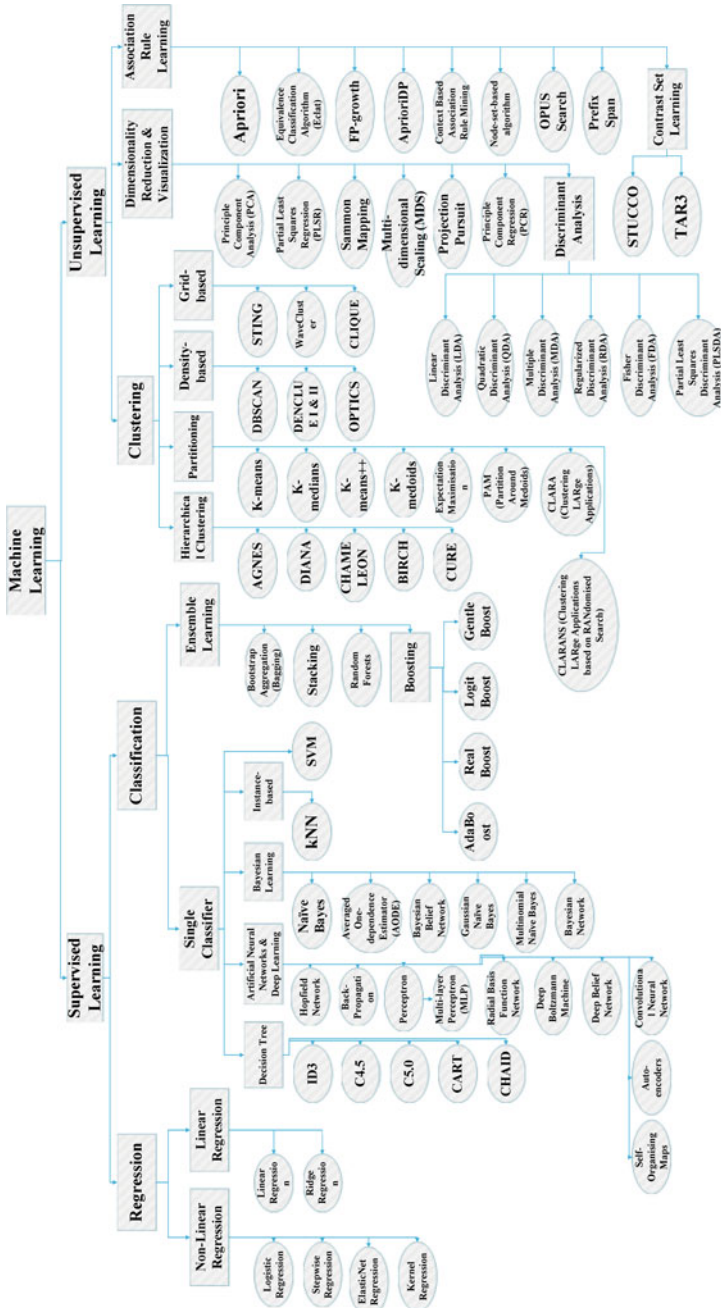


Fig. 3 A taxonomy of machine learning algorithms for summarizing the large graphs [2]

for summarizing large ER graphs. We provide an extensible interface to support external graph reachability algorithms (such as [5, 29]: Transitive Closure, GRIPP, Tree Cover, Chain Cover, Path-Tree Cover, and Shortest Path) to discover a set of related paths and store them in path nodes.

**Definition 4.5 (Process Model)** A process model  $PM = (PG, F_{pd})$  is a summarized representation of a process instances graph  $PG$  obtained by applying a process discovery algorithm  $F_{pd}$ .

To discover process models, a query can be applied on a previously constructed process instances graph (stored in a folder node). For example, a process instances graph, as a result of a correlation condition, may partition a subset of the events in the graph into instances of a process.

## 5 Querying Big Process Data

In this section, we present a summary of our previous work [5, 8, 10, 13] on querying process data. We present the BP-SPARQL (Business Process SPARQL), an extension of SPARQL<sup>3</sup> for analyzing business process execution data. BP-SPARQL enables the modeling, storage, and querying of process data using Hadoop MapReduce framework. In BP-SPARQL, we use folder and path nodes as first-class entities that can be defined at several levels of abstraction and queried. BP-SPARQL provides a data mapping layer for the automatic translation of SPARQL queries into MapReduce operations. To additionally account for the subjectiveness of process data analysis, as well as to enable process analysts to analyze business data in an easy way, we support various querying needs over process data. Next, we present various types of queries supported by BP-SPARQL including entity-level (artifacts, events, and activities), summarization (OLAP Style, Group Style, and Partition Style), relationship (Regular Expression, Path Condition, and Path Node), metadata (Time and Provenance), and user-defined queries.

### 5.1 Entity-Level Queries

We support the use of SPARQL to query entities and their attributes. We introduce the *entity* statement which enables process analysts to extract information about process-related entities such as business artifacts, people, and activities in an easy way. This statement has the following syntax:

```
entity [entity-type] \[attribute-name] =/!=/>/>=/</<= [value]
```

<sup>3</sup> Among languages for querying graphs, SPARQL [24] is an official W3C standard based on a powerful graph matching mechanism.

In this statement, “entity” is a reserved word, “entity-type” is the type of process-related entities, such as artifact and people, and the “value” represents the value of the entity. The value should be quoted. The “\” character represents the filter to be applied to entity attributes. The entity statement supports the conditional AND and OR operators. Parenthesis can be used in complex filters. The result of this query is an entity or a set of entities satisfying the condition. The entity statement automatically translates to a SPARQL query. The details about this translation process can be found in [5].

*Example 5.1* Considering the motivating scenario, Tim (a process analyst) is interested in finding home-loan documents submitted to “Sydney” branches.

```
entity artifact \category='home-loan' AND \submission-branch='Sydney'
```

In this example, “artifact” is the type of the entity to be filtered, “\category='home-loan'” filters the category of the output entities to home-loan documents, and “\submission-branch='Sydney'” filters the output to entities whose “submission branch” attribute is set to “Sydney”. More complex filters can be applied in this query. For example, if Tim is interested to find artifacts submitted in December 2017, he needs to add “(\submission-date ≥ '01-12-2017' AND \submission-date ≤ '31-12-2017')” condition to the above query.

## 5.2 Summarization Queries

BP-SPARQL supports three types of summarization queries: OLAP Style, Group Style, and Partition Style queries.

*OLAP Style Queries* BP-SPARQL supports scalable graph-based OLAP analytics over process execution data [6, 10]. The goal is to facilitate the analytics over the ER graph through summarizing the process graph and providing multiple views at different granularities. To achieve this goal, we present a process OLAP (P-OLAP) model and define OLAP specific abstractions in process context such as process cubes, dimensions, and cells. For example, analytics queries can be used to partition the ER graph in the example scenario into sets of related actors collaborating on (specific) loan applications. To achieve this, a set of dimensions [10] coming from the attributes of customer, loan documents, actors, and the relationship among them should be analyzed.

*Group Style Queries* To summarize the process data, BP-SPARQL extensively supports multiple information needs with one data structure (ER graph) and one function (machine learning algorithms presented in Fig. 3). This capability enables analysts summarizing the large process graph, by extracting complex data structures such as time series, hierarchies, patterns, and subgraphs.

*Partition Style Queries* A correlation condition query can be used to partition (a subset of) the entities in the graph into related instances. Such partition style queries enable the analyst to divide the process data into partitions that can be stored in folder nodes and accessed separately. For example, it can be used to partition the events in the process graph into a set of process instances. This statement has the following syntax:

```
correlation [Correlation-Condition]
```

In this statement, “correlation” is a reserved word and “Correlation-Condition” is the condition to be defined. The result of this query is a collection of related entities satisfying the condition. The correlation statement is automatically translated to a BP-SPARQL query. The details about this translation process can be found in [5].

*Example 5.2* Tim is interested in partitioning the graph in the example scenario into a set of related entities having the same type (e.g., customer, actors, and document). The correlation condition  $\psi(node_x, node_y) : node_x.type = node_y.type$  can be defined over the attribute *type* of two node entities  $node_x$  and  $node_y$ . This predicate is true when  $node_x$  and  $node_y$  have the same type and false otherwise. Related node entities will be stored in folders, where each folder conforms to an entity-type described by a set of attributes.

A correlation condition can be assigned to a folder node to store the result of the query. Also, timed folders [8] can be used to document the evolution of the folder over time. A monitoring code snippet can be assigned to a folder, e.g., to execute the correlation condition query over time or execute the query in case of triggers. In this case, new entities can be added to timed folders over time.

### 5.3 Regular Expression Queries

Regular Expression queries can be used to discover transitive relationships between two entities in the ER graph. In order to discover transitive relationships among entities, BP-SPARQL supports regular language reachability algorithms [5, 29] over ER graphs. The result of such a query is stored in a path node. In particular, BP-SPARQL is designed to be customizable by process analysts who can codify their knowledge into *regular expressions* that describe paths through the nodes and edges in the ER graph.

A path through the graph recognized by a regular expression would be useful if, by computing the path and its end point nodes, it answers a *question* posed by a process analyst. In this context, regular expressions are written by domain experts and will be executed over the ER graph. BP-SPARQL takes care of the processing and optimization that is needed for efficient crawling, analyzing, and querying of the graph. We introduce the *relationship* statement which enables process analysts

to discover useful patterns among process-related entities. This statement has the following syntax:

```
relationship [Regular-Expression]
```

In this statement, “relationship” is a reserved word and “Regular-Expression” is a parameter. This statement will be automatically translated into a path query in BP-SPARQL. The details about this translation process can be found in [5]. The following examples illustrate how a domain expert can use regular expressions to discover transitive relationships between business artifacts, people, and activities:

*Example 5.3* Find bank staff who is working on Adam’s home-loan document.

**Regular Expression:**

Adam (edge node)\* assigned-to Staff

**Example discovered path:**

Adam  $\xrightarrow{\text{(submitted)}}$  document  $\xrightarrow{\text{(part-of)}}$  work-item  $\xrightarrow{\text{(assigned-to)}}$  Staff

*Example 5.4* Find manager who approved Adam’s loan report.

**Regular Expression:**

Adam (edge node)+ approved-by Manager

**Example discovered path:**

Adam  $\xrightarrow{\text{(submitted)}}$  document  $\xrightarrow{\text{(part-of)}}$  work-item  $\xrightarrow{\text{(assigned-to)}}$  staff  $\xrightarrow{\text{(created)}}$  report  $\xrightarrow{\text{(approved-by)}}$  Manager

*Example 5.5* Find artifacts related to Adam.

**Regular Expression :**

Adam (edge node)\* edge Artifact

**Example discovered paths:**

Adam  $\xrightarrow{\text{(submitted)}}$  Home-Loan-Document  
 Adam  $\xrightarrow{\text{(submitted)}}$  document  $\xrightarrow{\text{(part-of)}}$  work-item  $\xrightarrow{\text{(assigned-to)}}$  Tim  $\xrightarrow{\text{(created)}}$  Home-Loan-Report

In these examples, regular expressions are used to discover sets of paths in the process graph.

### 5.3.1 Path Condition Queries

Path condition queries are similar to relationship queries; they are able to store the query results in folder nodes. In particular, a *path condition* [10] can be used to group related entities in an ER graph based on a set of dimensions coming from the attributes of network structures: we need to apply conditions not only on graph entities but also on the relationships between them. A path condition  $\phi$  is defined as a binary predicate on the attributes of a path that allows to identify whether two or more entities (in a given ER graph) are potentially related through that path.

For example, Tim is interested in finding a set of related actors (e.g., home-loan employees) working on Adam’s home-loan application. Therefore, Tim is interested in creating a folder node for a set of related actors and then adding related actors

to this folder if there exists a specific path between the customer and an actor. The path condition  $\phi(node_{start}, node_{end}, RE)$  can be defined on the existence of the path codified by the regular expression  $RE:[Adam(edge\ node)^* assigned-to\ STAFF]$  between starting node,  $node_{start}$ , and ending node,  $node_{end}$ . This predicate is *true* if the path exists and *false* otherwise. A path condition can be assigned to a folder node to store the result of the query (a set of entities). The details about this type of queries can be found in [10].

### 5.3.2 Path Node Queries

There are situations where process analysts are interested in storing the discovered paths as a result of a relationship query. This will enable to store a set of related patterns in a path node and use them as an input for further analytics tasks. Notice that results of relationship queries may be different over time, as new nodes and relationships can be added over time. The details about this type of queries can be found in [10].

## 5.4 Metadata Queries

In a process execution path, a huge amount of process-related metadata, such as versioning (what are the various versions of an artifact and how are they related), provenance (what manipulations were performed on the artifact to get it to this version), security (who has access to the artifact over time), and privacy (what actions were performed to protect or release artifact information) can be recorded. These metadata can be used to imbue the process data with additional semantics.

In [8], we formalized metadata to be collected, while an activity is taking place, including:

- *When*, to indicate the timestamp in which the activity has occurred
- *Who*, to indicate an actor that enables, facilitates, controls, or affects the activity execution
- *Where*, to indicated the organization/department the activity happened
- *Which*, to indicate the system which hosts the activity
- *Why*, to indicate the purpose of the activity, e.g., fulfillment of a specific phase or experiment

We highlighted that discovering paths through ER graphs forms the basis of many metadata queries. Next, we present simple queries to discover evolution (how the business artifact evolved over time?), derivation (what are the ancestors of the business artifact?), and time series (what are the snapshots of the business



artifact over time?) of business-related artifacts. The query template statement has the following syntax:

```
metadata evolutionOf/derivationOf/timeseriesOf [artifact-name]
filter [who, where, which, when, ...]
```

This statement can be used for discovering evolution of artifacts (using `evolutionOf` construct), derivation of artifacts (using `derivationOf` construct), and time series of artifacts/actors (using `timeseriesOf` construct). The “filter” statement restricts the result to those activities for which the filter expression evaluates to true. Variables, such as artifact, type (e.g., lifecycle or archiving), action (e.g., creation, use, or storage), actor, and location (e.g., organization), are defined as filters.

*Example 5.6* For querying the evolution of an entity *En*, all activity paths on top of *En* ancestors should be discovered. For example, considering the motivating scenario, Tim is interested to see how version  $v_2$  of Adam’s loan document evolved from version  $v_1$ .

The following is a sample query template for this example:

```
evolutionOf Adam_loan_document_v2
```

Figure 4a illustrates the result of this query. Figure 4b illustrates how a set of paths between the two versions can be stored in a sample path node. In particular, three paths are recognized, assigned unique identifiers (e.g., `path#1`), and stored under a path node name. Further user-defined queries can be applied to the path nodes for subsequent analysis. Additional filters can also be added to the above query. For example, if Tim is interested to see the activities that involved creating a new artifact, he can use the following query template:

```
evolutionOf Adam_loan_document_v2 \what='lifecycle' \how='create'
```

As a result, Tim will only see paths `path#1` and `path#2`, in Fig. 4b. More examples can be found in [8].

## 5.5 User-Defined Queries

Besides the abovementioned query templates, a process analyst may apply user-defined queries. In this case, the analyst should be familiar with SPARQL syntax. In particular, a basic SPARQL query has the following form:

```
select ?variable1 ?variable2 ...
where {pattern1. pattern2. #Other-Patterns}
```

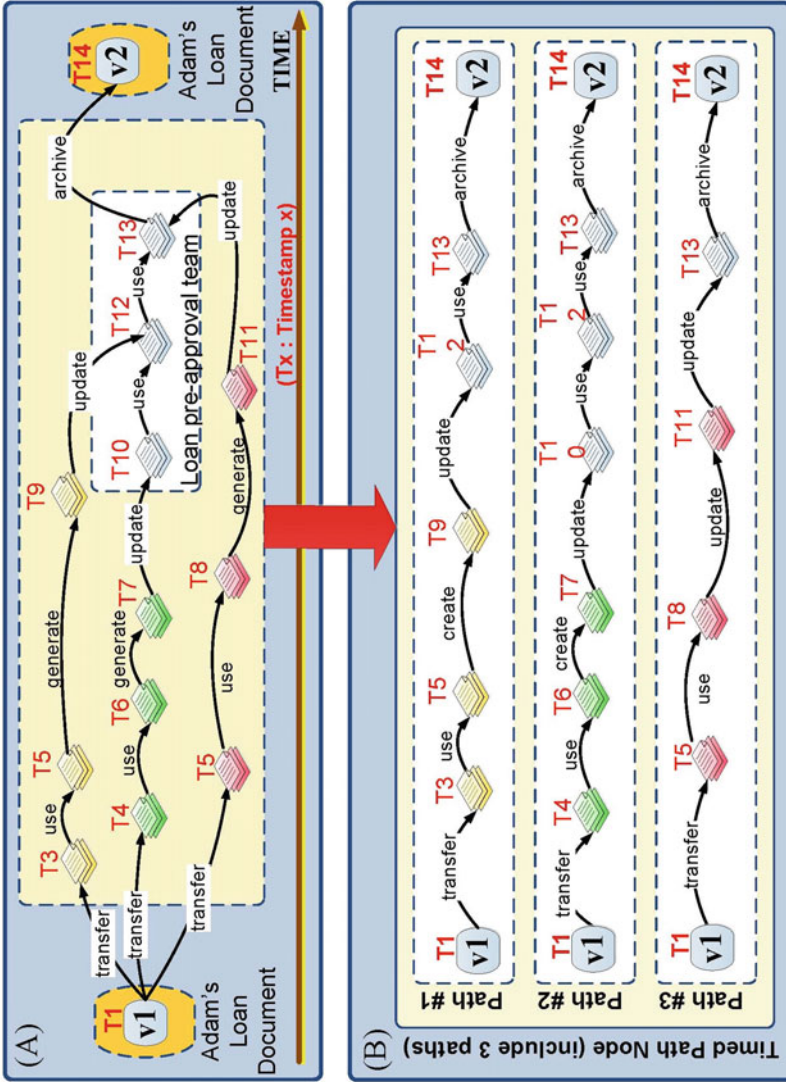


Fig. 4 Example business process executions

Each pattern consists of *subject*, *predicate*, and *object*, and each of these can be either a variable or a literal. The query specifies the known literals and leaves the unknown as variables. To answer a query, one needs to find all possible variable bindings that satisfy the given patterns. It is possible to use the “@” symbol for representing attribute edges and distinguishing them from the relationship edges between graph nodes. As an example, considering the motivating scenario, Tim may be interested in monitoring the conversations (e.g., messages) between the loan approval team. For example, Tim may be interested in retrieving a list of messages that have the same value on “requestsize” and “responsesize” attributes and the values for their timestamps fall between  $t_1$  and  $t_2$ . The following is the SPARQL query for this example:

```

1: select ?m
2: where{
3:   ?m @type message.
4:   ?m @requestsize ?x.
5:   ?m @responsesize ?y.
6:   ?m @timestamp ?t.
7:   FILTER (?x=?y && ?t > t1 && ?t < t2). }

```

In this query, variable  $?m$  represents a message in the graph. Variables  $?x$ ,  $?y$ , and  $?t$  represent the value of the attribute “requestsize” (line 4), “responsesize” (line 5), and “timestamp” (line 6), respectively. The *FILTER* statement restricts the result to those messages for which the filter expression evaluates to *true*.

## 6 Scalable Analysis Using MapReduce

The popular MapReduce [16] scalable data processing framework and its open-source realization Hadoop [28] offer a scalable dataflow programming model that appeals to many users. In MapReduce frameworks, computations are specified via two user-defined functions: a mapper that takes key–value pairs as input and produces key–value pairs as output and a reducer that consumes those key–value pairs (generated in the mapper phase) and aggregates data based on individual keys. In practice, the extreme simplicity of the MapReduce programming model leads to several problems. For example, it does not directly support complex N-step data flows which often arise in practice. To address this problem, Apache Pig [22] system offers compassable high-level data manipulation constructs in the spirit of SQL, while at the same time retaining the properties of MapReduce systems makes them attractive for certain users, data types, and workloads.

Pig’s language layer consists of a textual language called PigLatin which supports ease of programming, optimization opportunities, and extensibility. It is possible to write a single script in PigLatin that is automatically parallelized and distributed across a Hadoop cluster. A script in Pig often follows the *input – process – output* (IPO) model: (i) Input: data is read from the Hadoop Distributed File System (HDFS), (ii) Process: a number of operations (e.g., LOAD, SPLIT, JOIN, FILTER, GROUP, and STORE) are performed on the data, and (iii) Output: the resulting relation is written back to the file system.

Retrieving related graphs containing a graph query from a large RDBMS graph database is a key performance issue, where a primary challenge in computing the answers of such graph queries is that pairwise comparisons of graphs are usually hard problems. We use Hadoop [28] data processing platform to store and retrieve graphs in Hadoop file system and to support cost-effective and scalable processing of graphs. We use Apache Pig, a high-level procedural language on top of MapReduce, for querying large graphs stored in Hadoop file system. We use the algebra proposed in [25] for mapping SPARQL queries to PigLatin programs and consequently generating MapReduce operations. To capture the storage model in Pig, an input graph needs to be “split” into property-based partitions using PigLatin’s SPLIT command. Then, the star-structured joins are achieved using the m-way JOIN operator, and chain joins are executed using the binary JOIN operator.

PigLatin queries are compiled into a sequence of MapReduce operations that run over Hadoop. The Hadoop scheduling supports partition parallelism such that in every stage, one operator is running on a different partition of data. In particular, the logical plan for Pig queries can be described as follows: (i) load the input dataset using the LOAD operator, (ii) create vertical partitioned relations using the SPLIT operator, (iii) join partitioned relations based on join conditions: In SPARQL, join conditions are implied by repeated occurrences of variables in different triple patterns. Consequently, for each star join, the join will be computed in a single MapReduce cycle, and (iv) store the result on disk using the STORE operator.

For example, consider the following query in the context of the motivating scenario: “give the name of Australian banks which offer loan products (e.g., home loan, business loan, variable/fix rates, etc.) within 15 days, along with the review details for these products.” Figure 5 illustrates the corresponding BP-SPARQL query and MapReduce execution flow. As illustrated, the query can be factorized into three main sections: (three star-join structures (S1, S2, S3) describing resources of type Vendor, Offer, and Review, respectively, two chain-join pattern (J1, J2) combining the star patterns, and the filter processing. In particular, such queries can be considered equivalent to the select–project–join construct in SQL, where each MapReduce cycle may involve communication and I/O costs due to the data transfer between the mapper and the reducer.

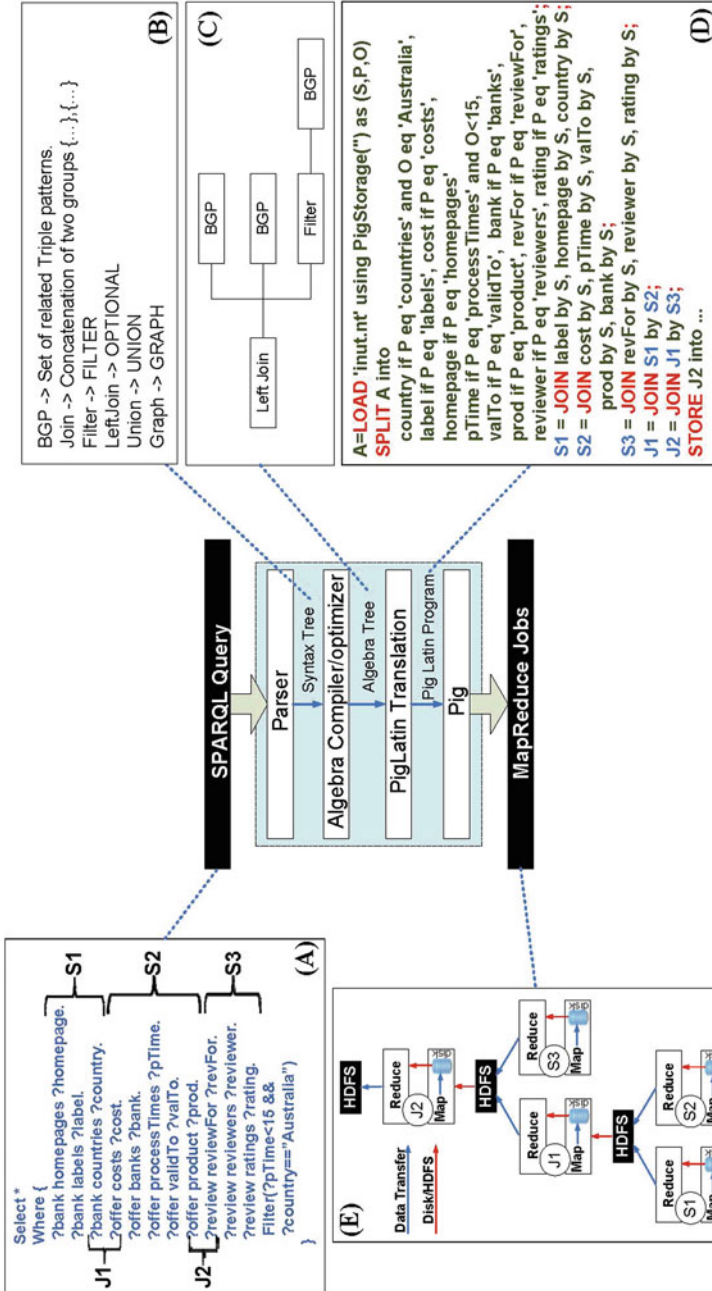


Fig. 5 An example BP-SPARQL query [10, 13] (a) the SPARQL syntax tree (b); and Algebra tree (c), translated PigLatin (d), and its MapReduce execution flow (e)

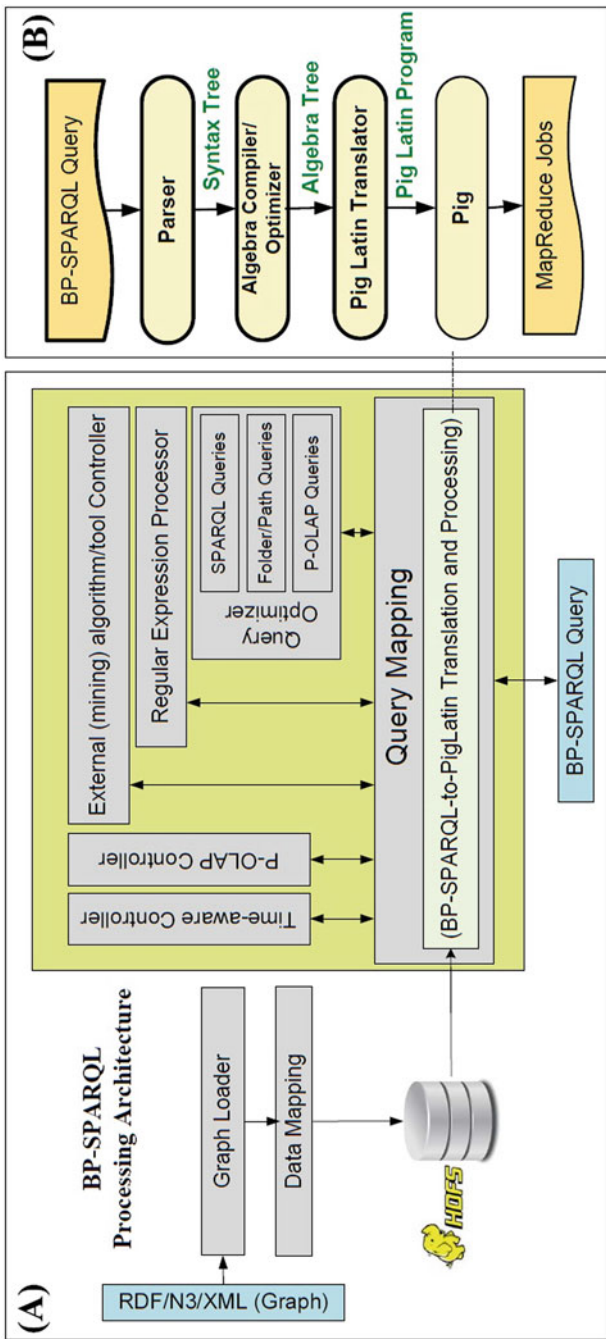
## 7 Implementation

We have implemented a research prototype of BP-SPARQL platform for organizing, indexing, and querying process data. The prototype supports two types of storage backend: Relational Database System and Hadoop File System. We use Apache Pig (pig.apache.org), a high-level procedural programming language on top of Hadoop for querying large graphs. The main components of the query engine include (i) *Data Mapping Layer*: this layer is responsible for creating data element mappings between semantic Web technology and physical storage layer, i.e., relational database schema and Hadoop File System, (ii) *Time-Aware Controller*: RDF databases are not static and changes may apply to graph entities (i.e., nodes, edges, and folder/path nodes) over time. Time-aware controller is responsible for data changes, data manipulation, and incremental graph loading, (iii) *Query Mapping Layer*: this layer is responsible for BP-SPARQL queries translation and processing, (iv) *Regular Expression Processor*: we developed a regular expression processor which supports optional elements, loops, alternation, and groupings, (v) *External Algorithm/Tool Controller*: this component is responsible for supporting external graph reachability/mining algorithms, and (vi) *GOLAP Controller*: This component is responsible for partitioning graphs and evaluation of OLAP operations independently for each partition, providing a natural parallelization of execution.

The details on the implementation and evaluation can be found in our previous work [5, 8, 10, 13]. Figure 6a illustrates BP-SPARQL graph processing architecture.

## 8 Process Querying Framework

Polyvyany et al. [23] proposed the framework for developing process querying methods. Complimentary to the active components proposed in this framework, BP-SPARQL focused on facilitating the querying and analysis of data-driven processes and knowledge-intensive processes. Such processes include a set of coordinated tasks and activities, controlled by knowledge workers to achieve a business objective or goal. Examples include police investigation processes and government processes, such as in immigration, health, and education departments. Facilitating the querying and analysis of such processes is important, as the continuous improvement in connectivity, storage, and data processing capabilities allows access to a data deluge from sensors, social media, news, user-generated, government, and private data sources. Accordingly, business processes become inseparable from data. Examples of process data include data from the execution of business processes, documentation and description of processes, process models, process variants, artifacts related to business processes, and data generated or exchanged during process execution. In the following, we discuss some of the related works in organizing, querying, and analyzing process data.



**Fig. 6** BP-SPARQL graph processing architecture (a) and the modular translation process for mapping SPARQL to PigLatin (b) [10]

**From Data Lakes to Knowledge Lakes** With data science continuing to emerge as a powerful differentiator across industries, almost every organization is now focused on understanding their business and transforming data into actionable insights. The notion of a Data Lake [12] has been coined to address this challenge and to convey the concept of a centralized repository containing limitless amounts of raw (or minimally curated) data stored in various data islands. The rationale behind a Data Lake is to store raw data and let the data analyst decide how to cook/curate them later. While Data Lakes do a great job in organizing Big data and providing answers on known questions, the main challenges are to understand the potentially interconnected data stored in various data islands and to prepare them for analytics.

The notion of Knowledge Lake [14], i.e., a contextualized Data Lake, is introduced to automatically transform the raw (process) data into contextualized data and knowledge. The term Knowledge here refers to a set of facts, information, and insights extracted from the raw data using data curation techniques, such as extraction, linking, summarization, annotation, enrichment, classification, and more. In particular, a Knowledge Lake is a centralized repository containing a virtually inexhaustible amount of data and contextualized data that is readily made available anytime to anyone authorized to perform analytical activities. Knowledge Lakes provide the foundation for Big data analytics by automatically curating the raw data in Data Lakes and preparing them for deriving insights.

**Process Graph Modeling** Graphs are essential modeling and analytical objects for representing information networks. Several graph querying techniques [1] such as pattern match query, reachability query, shortest path query, and subgraph search have been proposed for querying and analyzing graphs. These methods rely on constructing some indices to prune the search space of each vertex to reduce the whole search space. In [1], the authors discuss a number of data models and query languages for graph data. Many of these models use RDF [24] (Resource Description Framework), an official W3C recommendation for semantic Web data models, to model graphs and use SPARQL, an official W3C recommendation for querying RDF graphs [24]. SPARQL queries are pattern matching queries on triples that constitute an RDF data graph, where RDF is a data model for schema-free structured information. Several research efforts have been proposed to address efficient and scalable management of RDF data. SPARQL [24] is a declarative query language, a W3C standard, for querying and extracting information from directed labeled RDF graphs. SPARQL supports queries consisting of triple patterns, conjunctions, disjunctions, and other optional patterns. However, there is no support for querying grouped entities. Paths are not first-class objects in SPARQL [24]. PPARQL [3] extends SPARQL with regular expression patterns allowing path queries. SPARQLeR [18] is an extension of SPARQL designed for finding semantic associations (and path patterns) in RDF graphs. In BP-SPARQL, we support folder and path nodes as first-class entities that can be defined at several levels of abstractions and queried.

**Knowledge-Intensive Processes** Case-managed processes are primarily referred to as semistructured processes, since they often require the ongoing intervention of



skilled and knowledgeable workers. Such knowledge-intensive processes involve operations that rely on professional knowledge. For these reasons, it is considered that human knowledge workers are responsible to drive the process, which cannot otherwise be automated as in workflow systems [13]. Knowledge-intensive processes often involve the collection and presentation of a diverse set of artifacts and human activities around artifacts. This emphasizes the artifact-centric nature of such processes. Many approaches [15, 17, 26] use business artifacts that combine data and processes in a holistic manner and as the basic building block. The work by Gerede et al. [17] used a variant of finite-state machines to specify lifecycles. The theoretical work by Bhattacharya et al. [15] explored declarative approaches to specifying the artifact lifecycles following an event-oriented style. Another line of work in this category focused on querying artifact-centric processes [19].

**Process Data Analytics** In our recent book [11], we provided an overview of the state-of-the-art in the area of business process management in general and process data analytics in particular. This book provides defrayals on (i) technologies, applications, and practices used to provide process analytics from querying to analyzing process data, (ii) a wide spectrum of business process paradigms that have been presented in the literature from structured to unstructured processes, (iii) the state-of-the-art technologies and the concepts, abstractions, and methods in structured and unstructured BPM including activity-based, rule-based, artifact-based, and case-based processes, and (iv) the emerging trend in the business process management area such as process spaces, Big data for processes, crowdsourcing, social BPM, and process management in the cloud. BPM in the cloud solutions offer visibility and management of business processes, low start-up costs, and fast return on investment. Crowdsourcing can help organizations to increase productivity by discovering and exploiting informal knowledge and relationships to improve activity execution. Crowdsourcing can also enable socialBPM to assign an activity to a broader set of performers or to find appropriate contributors for its execution. Social BPMs inevitably require advanced crowd management capabilities in future social computing platforms.

## 9 Conclusion

The continuous demand for the business process improvement and excellence has prompted the need for business process analysis. Recently, business world is getting increasingly dynamic as various technologies such as Internet and e-mail have made dynamic processes more prevalent. In this chapter, we focused on the problem of explorative querying and understanding of business processes data. Our study shows that only part of interactions related to the process executions are covered by process-aware systems as business processes are realized over a mix of workflows, IT systems, Web services, and direct collaborations of people. In order to fulfill the requirements, we proposed a framework for organizing,

indexing, and querying ad hoc process data. In this framework, we proposed novel abstractions for summarizing process data and a language, BP-SPARQL, for the explorative querying and understanding of BP execution from various user perspectives. In future work, we will employ interactive graph exploration and visualization techniques (e.g., storytelling systems) to facilitate the use of BP-SPARQL through visual query interface.

## References

1. Aggarwal, C., Wang, H.: *Managing and Mining Graph Data*. Springer Publishing Company (2010)
2. Amouzgar, F., Beheshti, A., Ghodrathnama, S., Benatallah, B., Yang, J., Sheng, Q.Z.: iSheets: A spreadsheet-based machine learning development platform for data-driven process analytics. In: *Service-Oriented Computing - ICSOC 2018 Workshops - ADMS, ASOCA, ISYyCC, CloTS, DDBS, and NLS4IoT*, Hangzhou, China, November 12–15, 2018, Revised Selected Papers, pp. 453–457 (2018)
3. Anyanwu, K., Maduko, A., Sheth, A.P.: SPARQ2L: towards support for subgraph extraction queries in RDF databases. In: *Proceedings of the 16th International Conference on World Wide Web, WWW 2007, Banff, Alberta, Canada, May 8–12, 2007*, pp. 797–806 (2007)
4. Barros, A.P., Decker, G., Dumas, M., Weber, F.: Correlation patterns in service-oriented architectures. In: *FASE*, pp. 245–259 (2007)
5. Beheshti, S., Benatallah, B., Nezhad, H.R.M., Sakr, S.: A query language for analyzing business processes execution. In: *Business Process Management - 9th International Conference, BPM 2011, Clermont-Ferrand, France, August 30 - September 2, 2011. Proceedings*, pp. 281–297 (2011)
6. Beheshti, S., Benatallah, B., Nezhad, H.R.M., Allahbakhsh, M.: A framework and a language for on-line analytical processing on graphs. In: *Web Information Systems Engineering - WISE 2012 - 13th International Conference, Paphos, Cyprus, November 28–30, 2012. Proceedings*, pp. 213–227 (2012)
7. Beheshti, S., Nezhad, H.R.M., Benatallah, B.: Temporal provenance model (TPM): model and query language. *CoRR* **abs/1211.5009** (2012)
8. Beheshti, S., Benatallah, B., Nezhad, H.R.M.: Enabling the analysis of cross-cutting aspects in ad-hoc processes. In: *Advanced Information Systems Engineering - 25th International Conference, CAiSE 2013, Valencia, Spain, June 17–21, 2013. Proceedings*, pp. 51–67 (2013)
9. Beheshti, S., Benatallah, B., Motahari-Nezhad, H.R.: Galaxy: A platform for explorative analysis of open data sources. In: *Proceedings of the 19th International Conference on Extending Database Technology, EDBT 2016, Bordeaux, France, March 15–16, 2016, Bordeaux, France, March 15–16, 2016.*, pp. 640–643 (2016)
10. Beheshti, S., Benatallah, B., Motahari-Nezhad, H.R.: Scalable graph-based OLAP analytics over process execution data. *Distrib. Parallel Databases* **34**(3), 379–423 (2016)
11. Beheshti, S., Benatallah, B., Sakr, S., Grigori, D., Motahari-Nezhad, H.R., Barukh, M.C., Gater, A., Ryu, S.H.: *Process Analytics - Concepts and Techniques for Querying and Analyzing Process Data*. Springer (2016)
12. Beheshti, A., Benatallah, B., Nouri, R., Chhieng, V.M., Xiong, H., Zhao, X.: CoreDB: a data lake service. In: *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management, CIKM 2017, Singapore, November 06–10, 2017*, pp. 2451–2454 (2017)
13. Beheshti, A., Benatallah, B., Motahari-Nezhad, H.R.: Processatlas: A scalable and extensible platform for business process analytics. *Softw. Pract. Exper.* **48**(4), 842–866 (2018)
14. Beheshti, A., Benatallah, B., Nouri, R., Tabebordbar, A.: CoreKG: A knowledge lake service. *PVLDB* **11**(12), 1942–1945 (2018)

15. Bhattacharya, K., Gerede, C.E., Hull, R., Liu, R., Su, J.: Towards formal analysis of artifact-centric business process models. In: *BPM*, pp. 288–304 (2007)
16. Dean, J., Ghemawat, S.: MapReduce: simplified data processing on large clusters. *Commun. ACM* **51**(1), 107–113 (2008)
17. Gerede, C., Su, J.: Specification and verification of artifact behaviors in business process models. In: *ICSOC*, pp. 181–192 (2007)
18. Kochut, K., Janik, M.: SPARQLer: Extended SPARQL for semantic association discovery. In: *The Semantic Web: Research and Applications, 4th European Semantic Web Conference, ESWC 2007, Innsbruck, Austria, June 3–7, 2007, Proceedings*, pp. 145–159 (2007)
19. Kuo, J.: A document-driven agent-based approach for business processes management. *Inf. Softw. Technol.* **46**(6), 373–382 (2004)
20. McAfee, A., Brynjolfsson, E., Davenport, T.H., Patil, D., Barton, D.: Big data: the management revolution. *Harv. Bus. Rev.* **90**(10), 60–68 (2012)
21. Motahari-Nezhad, H., Saint-Paul, R., Casati, F., Benatallah, B.: Event correlation for process discovery from Web service interaction logs. *VLDB J.* **20**(3), 417–444 (2011)
22. Olston, C., Reed, B., Srivastava, U., Kumar, R., Tomkins, A.: Pig Latin: a not-so-foreign language for data processing. In: *SIGMOD*, pp. 1099–1110 (2008)
23. Polyvyanyy, A., Ouyang, C., Barros, A., van der Aalst, W.M.P.: Process querying: Enabling business intelligence through query-based process analytics. *Decis. Support Syst.* **100**, 41–56 (2017)
24. Prud’hommeaux, E., Seaborne, A.: SPARQL query language for RDF (working draft). Tech. rep., W3C (2007)
25. Schätzle, A., Przyjaciel-Zablocki, M., Lausen, G.: PigSPARQL: mapping SPARQL to Pig Latin. In: *Proceedings of the International Workshop on Semantic Web Information Management* (2011)
26. Sun, Y., Su, J., Yang, J.: Universal artifacts: A new approach to business process management (BPM) systems. *ACM Trans. Manag. Inf. Syst.* **7**(1), 3:1–3:26 (2016)
27. van der Aalst, W., ter Hofstede, A.H.M., Weske, M.: Business process management: A survey. In: *BPM* (2003)
28. White, T.: *Hadoop: The Definitive Guide*, original edn. O’Reilly Media (2009)
29. Yu, J.X., Cheng, J.: Graph reachability queries: A survey. In: *Managing and Mining Graph Data*, pp. 181–215. Springer (2010)
30. Zikopoulos, P., Eaton, C., et al.: *Understanding Big data: Analytics for enterprise class Hadoop and streaming data*. McGraw-Hill Osborne Media (2011)

# Data-Aware Process Oriented Query Language



Eduardo Gonzalez Lopez de Murillas, Hajo A. Reijers,  
and Wil M. P. van der Aalst

**Abstract** The size of execution data available for process mining analysis grows several orders of magnitude every couple of years. Extracting and selecting the relevant data to enable process mining remains a challenging and time-consuming task. In fact, it is the biggest handicap when applying process mining and other forms of process-centric analysis. This work presents a new query language, DAPOQ-Lang, which overcomes some of the limitations identified in the field of process querying and fits within the Process Querying Framework. The language is based on the OpenSLEX meta model, which combines both data and process perspectives. It provides simple constructs to intuitively formulate questions. The syntax and semantics have been formalized and an implementation of the language is provided, along with examples of queries to be applied to different aspects of the process analysis.

## 1 Introduction

One of the main goals of process mining techniques is to obtain insights into the behavior of systems, companies, business processes, or any kind of workflow under study. Obviously, it is important to perform the analysis on the right data. Data extraction and preparation are among the first steps to take and, in many cases, up to 80% of the time and effort, and 50% of the cost is spent during these phases [12].

---

E. G. L. de Murillas (✉)

Department of Mathematics and Computer Science, Eindhoven University of Technology,  
Eindhoven, The Netherlands

H. A. Reijers

Department of Information and Computing Sciences, Utrecht University, Utrecht, The  
Netherlands

e-mail: [h.a.reijers@uu.nl](mailto:h.a.reijers@uu.nl)

W. M. P. van der Aalst

Department of Computer Science, RWTH Aachen University, Aachen, Germany

e-mail: [wvdaalst@pads.rwth-aachen.de](mailto:wvdaalst@pads.rwth-aachen.de)

© Springer Nature Switzerland AG 2022

A. Polyvyanyy (ed.), *Process Querying Methods*,

[https://doi.org/10.1007/978-3-030-92875-9\\_3](https://doi.org/10.1007/978-3-030-92875-9_3)

Being able to extract and query some specific subset of the data becomes crucial when dealing with complex and heterogeneous datasets. In addition, the use of querying tools allows one to find specific cases or exceptional behavior. Whatever the goal, analysts often find themselves in the situation in which they need to develop ad hoc software to deal with specific datasets, given that existing tools might be difficult to use, too general, or just not suitable for process analysis.

Different approaches exist to support the *querying of process data*. Some of them belong to the field of business process management (BPM). In this field, events are the main source of information. They represent transactions or activities that were executed at a certain moment in time in the environment under study. Querying this kind of data allows us to obtain valuable information about the behavior and execution of processes. There are other approaches originating from the field of data provenance, which are mainly concerned with recording and observing the origins of data. This field is closely related to scientific workflows in which the traceability of the origin of experimental results becomes crucial to guarantee correctness and reproducibility. In the literature, we find many languages to query process data. However, none of these approaches succeeds at combining process and data aspects in an integrated way. An additional challenge to overcome is the development of a query mechanism that allows to exploit this combination, while being intuitive and easy to use.

In order to make the querying of process event data easier and more efficient, we propose the Data-Aware Process Oriented Query Language (DAPOQ-Lang). This query language, first introduced in [3], exploits both process and data perspectives. The aim of DAPOQ-Lang is not to theoretically enable new types of computations, but to ease the task of writing queries in the specific domain of process mining. Therefore, our focus is on the ease of use. We propose the following example to show the ease of use of DAPOQ-Lang. Let us consider a generic question that could be asked by an analyst when carrying out a process mining project:

**GQ:** In which cases, there was (a) an event that happened between time T1 and T2, (b) that performed a modification in a version of class C, (c) in which the value of field F changed from X to Y?

This query involves several types of elements: cases, events, object versions, and attributes. We instantiate this query with some specific values for T1 = “1986/09/17 00:00”, T2 = “2016/11/30 19:44”, C = “CUSTOMER”, F = “ADDRESS”, X = “Fifth Avenue”, and Y = “Sunset Boulevard”. Query 1 presents the corresponding DAPOQ-Lang query. This example shows how compact a DAPOQ-Lang query can be. The specifics of this query will be explained in the coming sections.

The rest of this chapter is organized as follows. Section 2 introduces some background information, which is needed to understand the specifics of our query language. Section 3 presents the query language, focusing on the syntax and semantics. Section 4 provides information about the implementation and its evaluation. Section 5 presents possible use cases. Section 6 positions DAPOQ-Lang in the Process Querying Framework [10]. Section 7 concludes the chapter.

**Query 1** DAPOQ-Lang query to retrieve cases with an event happening between two dates that changed the address of a customer from “Fifth Avenue” to “Sunset Boulevard”.

```

1  def P1 = createPeriod("1986/09/17_00:00", "2016/11/30_19:44", "yyyy/MM/dd_HH:mm
    ↪ ")
2
3  casesOf(
4    eventsOf(
5      versionsOf(
6        allClasses().where {name == "CUSTOMER"}
7      ).where {
8        changed([at: "ADDRESS", from:"Fifth_Avenue", to:"Sunset_Boulevard"]) }
9    ).where
10   {
11     def P2 = createPeriod(it.timestamp)
12     during(P2, P1)
13   }
14 )

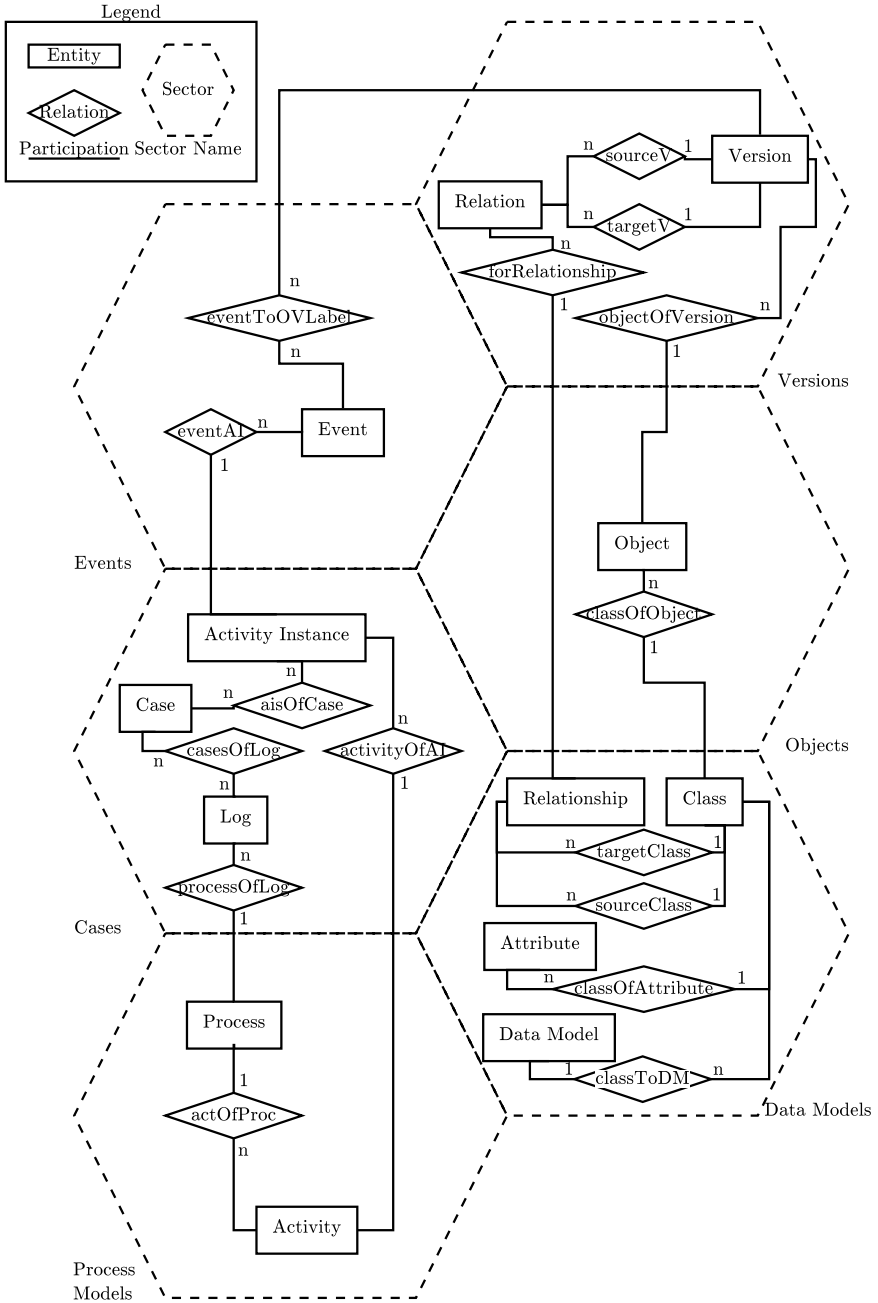
```

## 2 Preliminaries

To enable our approach to data querying, we need to have access to data storage, and the information should comply with a certain structure. An appropriate structure has been previously defined as a meta model [4] and implemented in a *queryable* file format called OpenSLEX. The meta model captures all the necessary aspects to enable data querying with our language. This section describes the structure of OpenSLEX and provides the necessary background to understand the details of DAPOQ-Lang.

Standards of reference, like XES [5], are focused on the process view (events, traces, and logs) of systems. OpenSLEX supports all concepts present in XES but, in addition, considers the data elements (data model, objects, and versions) as an integral part of its structure. This makes it more suitable for database environments where only a small part of the information is process-oriented (i.e., events) with respect to the rest of data objects of different classes that may be seen as an augmented view on the process information. The OpenSLEX format is supported by a meta model (Fig. 1) that considers *data models* and *processes* as the entities at the highest abstraction level. These entities define the structure of more granular elements like *logs*, *cases*, and *activity instances* with respect to processes and *objects* with respect to classes in the data model. Each of these elements at the intermediate level of abstraction can be broken apart into more granular pieces. This way, *cases* are formed by *events*, and *objects* can be related to several *object versions*. Both *events* and *object versions* represent different states of a higher level abstraction (*cases* or *objects*) at different points in time. A detailed ER diagram of the OpenSLEX format can be found online.<sup>1</sup> The format makes use of an SQL

<sup>1</sup> <https://github.com/edugonza/OpenSLEX/blob/master/doc/meta-model.png>.



**Fig. 1** ER diagram of the OpenSLEX meta model. The entities have been grouped into sectors, delimited by the dashed lines

schema to store all the information, and a Java API<sup>2</sup> is available for its integration in other tools. The use of OpenSLEX in several environments, e.g., database redo logs and ERP databases, is evaluated in [4], focusing on the data extraction and transformation phase. To provide the necessary background for the understanding of this work, a simplified version of the meta model is formally presented below. Every database system contains information structured with respect to a data model. Definition 1 provides a formalization of a data model in the current context.

**Definition 1 (Data Model)** A data model is a tuple  $DM = (CL, AT, classOfAttribute, RS, sourceClass, targetClass)$ , such that

- $CL$  is a set of class names.
- $AT$  is a set of attribute names.
- $classOfAttribute \in AT \rightarrow CL$  is a function that maps each attribute to a class.
- $RS$  is a set of relationship names.
- $sourceClass \in RS \rightarrow CL$  is a function mapping each relationship to its source class.
- $targetClass \in RS \rightarrow CL$  is a function mapping each relationship to its target class.

Data models contain classes (i.e., tables), which contain attribute names (i.e., columns). Classes are related by means of relationships (i.e., foreign keys). Definition 2 formalizes each of the entities of the OpenSLEX meta model, as can be observed in Fig. 1, and shows connections between them.

**Definition 2 (Connected Meta Model)** Let  $V$  be some universe of values and  $TS$  a universe of timestamps. A connected meta model is defined as a tuple  $CMM = (DM, OC, classOfObject, OVC, objectOfVersion, EC, eventToOVLabel, IC, eventAI, PMC, activityOfAI, processOfLog)$  such that

- $DM = (CL, AT, classOfAttribute, RS, sourceClass, targetClass)$  is a data model.
- $OC$  is a collection of objects.
- $classOfObject \in OC \rightarrow CL$  is a function that maps each object to its corresponding class.
- $OVC = (OV, attValue, startTimestamp, endTimestamp, REL)$  is a version collection, where  $OV$  is a set of object versions,  $attValue \in (AT \times OV) \dashrightarrow V$  is a mapping of pairs of object version and attribute to a value,  $startTimestamp \in OV \rightarrow TS$  is a mapping between object versions and start timestamps,  $endTimestamp \in OV \rightarrow TS$  is a mapping between object versions and end timestamps, and  $REL \subseteq (RS \times OV \times OV)$  is a set of triples relating pairs of object versions through a specific relationship.
- $objectOfVersion \in OV \rightarrow OC$  is a function that maps each object version to an object.
- $EC = (EV, EVAT, eventTimestamp, eventLifecycle, eventResource, eventAttributeValue)$  is an event collection, where  $EV$  is a set of events,  $EVAT$

<sup>2</sup> <https://github.com/edugonza/OpenSLEX>.



is a set of event attribute names,  $eventTimestamp \in EV \rightarrow TS$  maps events to timestamps,  $eventLifecycle \in EV \rightarrow \{start, complete, \dots\}$  maps events to life cycle attributes,  $eventResource \in EV \rightarrow V$  maps events to resource attributes, and  $eventAttributeValue \in (EV \times EVAT) \not\rightarrow V$  maps pairs of event and attribute name to values.

- $eventToOVLLabel \in (EV \times OV) \not\rightarrow V$  is a function that maps pairs of an event and an object version to a label. The existence of a label associated with an event and an object version, i.e.,  $(ev, ov) \in dom(eventToOVLLabel)$ , means that both event and object version are linked. The label defines the nature of the link, e.g., “insert”, “update”, “delete”, etc.
- $IC = (AI, CS, LG, aisOfCase, casesOfLog, CSAT, caseAttributeValue, LGAT, logAttributeValue)$  is an instance collection, where  $AI$  is a set of activity instances,  $CS$  is a set of cases,  $LG$  is a set of logs,  $aisOfCase \in CS \rightarrow \mathcal{P}(AI)$  is a mapping between cases and sets of activity instances,<sup>3</sup>  $casesOfLog \in LG \rightarrow \mathcal{P}(CS)$  is a mapping between logs and sets of cases,  $CSAT$  is a set of case attribute names,  $caseAttributeValue \in (CS \times CSAT) \not\rightarrow V$  maps pairs of case and attribute name to values,  $LGAT$  is a set of log attribute names, and  $logAttributeValue \in (LG \times LGAT) \not\rightarrow V$  maps pairs of log and attribute name to values.
- $eventAI \in EV \rightarrow AI$  is a function that maps each event to an activity instance.
- $PMC = (PM, AC, actOfProc)$  is a process model collection, where  $PM$  is a set of processes,  $AC$  is a set of activities, and  $actOfProc \in PM \rightarrow \mathcal{P}(AC)$  is a mapping between processes and sets of activities.
- $activityOfAI \in AI \rightarrow AC$  is a function that maps each activity instance to an activity.
- $processOfLog \in LG \rightarrow PM$  is a function that maps each log to a process.

A connected meta model (CMM) provides the functions that make it possible to connect all the entities in the meta model. This is important in order to correlate elements, e.g., events that modified the same object. However, some constraints must be fulfilled for a meta model to be considered a valid connected meta model (e.g., versions of the same object do not overlap in time). The details about such constraints are out of the scope of this work, but their description can be found in [4]. DAPOQ-Lang queries are executed on data structures that fulfill the constraints set on the definition of a connected meta model. According to our meta model description, *events* can be linked to *object versions*, which are related to each other by means of *relations*. These *relations* are instances of data model *relationships*. In database environments, this would be the equivalent to using foreign keys to relate table rows and knowing which events relate to each row. For the purpose of this work, we assume that pairwise correlations between events, by means of related object versions, are readily available in the input connected meta model.

---

<sup>3</sup>  $\mathcal{P}(X)$  is the powerset of  $X$ , i.e.,  $Y \in \mathcal{P}(X)$  if  $Y \subseteq X$ .

### 3 DAPOQ-Lang

DAPOQ-Lang is a Data-Aware Process Oriented Query Language that allows the user to query data and process information stored in a structure compatible with the OpenSLEX meta model [4]. As described in Sect. 2, OpenSLEX combines database elements (data models, objects, and object versions) with common process mining data (events, logs, and processes), considering them as first-class citizens. DAPOQ-Lang considers the same first-class citizens as OpenSLEX, which makes it possible to write queries in the process mining domain enriched with data aspects with lower complexity than in other general purpose query languages like SQL.

Intuitively, we could think that considering all the elements of the OpenSLEX meta model as first-class citizens would introduce a lot of complexity in our language. However, these elements have been organized in a type hierarchy as subtypes of higher level superclasses (Fig. 2). It can be seen that *MMElement* (Meta Model Element) is an abstract class at the highest level *superclass*. Next, we distinguish two subtypes of elements: (1) stored elements (*StoredElement*), i.e., elements that can be found directly stored in an OpenSLEX structure, such as activities, events, objects, and logs and (2) computed elements (*ComputedElement*), i.e., elements that are computed based on the rest, temporal periods of cases and temporal periods of events. We will exploit this hierarchy to design a simple language, providing many basic functions that can operate on any *MMElement*, and some specific functions that focus on specific subtypes. Given a connected meta model *CMM* (Definition 2), we define the concept of *MMElement* in DAPOQ-Lang as the union of all its terminal subtypes:  $MMElement = AC \cup LG \cup EV \cup REL \cup OC \cup AT \cup CL \cup PER \cup PM \cup CS \cup AI \cup OV \cup RS \cup DM$ . Some of the functions that we define operate on sets of elements ( $\mathcal{P}(MMElement)$ ). However, as a constraint of our query language, we require that the elements of those sets belong to the same subtype (i.e., a set of *Class* elements, or a set of *Version* elements, or a set of *Event* elements, etc.). Therefore, we define the set *MMSets* as the set of all possible subsets of each element type in a

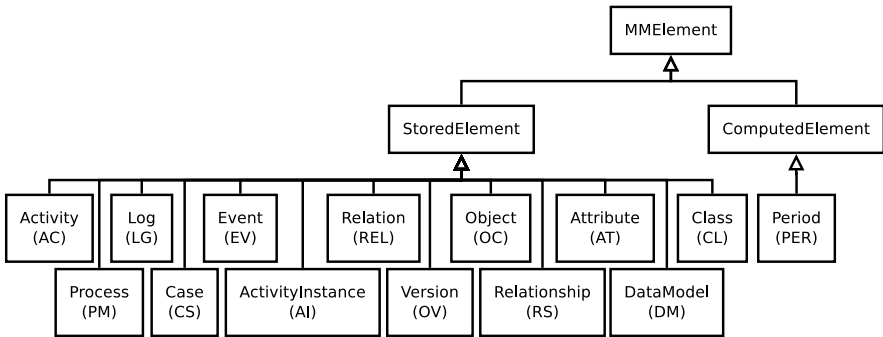


Fig. 2 DAPOQ-Lang type hierarchy. Arrows indicate subtype relations

meta model  $MM$ :

$$\begin{aligned}
 MM\text{Sets} = \mathcal{P}(AC) \cup \mathcal{P}(LG) \cup \mathcal{P}(EV) \cup \mathcal{P}(REL) \cup \mathcal{P}(OC) \cup \mathcal{P}(AT) \cup \mathcal{P}(CL) \cup \\
 \mathcal{P}(PER) \cup \mathcal{P}(PM) \cup \mathcal{P}(CS) \cup \mathcal{P}(AI) \cup \mathcal{P}(OV) \cup \mathcal{P}(RS) \cup \mathcal{P}(DM)
 \end{aligned}
 \tag{1}$$

The output of any query will be an element set  $es \in MM\text{Sets}$ , i.e., a set of elements of the same type. The following subsections describe the syntax and semantics of DAPOQ-Lang in detail.

### 3.1 Syntax

The language has been designed with the aim to find a balance between simplicity and expressive power. To do so, we exploited the specifics of the underlying meta model defining a total of 57 basic functions, as organized in 5 well-defined blocks, that can be applied in the context of a given meta model  $MM$ . The functions proposed in Sects. 3.1.1–3.1.5 will be used to define syntax and semantics of DAPOQ-Lang in Sects. 3.1 and 3.2.

#### 3.1.1 Terminal Meta Model Elements

We define a set of 13 basic terminal functions. Each of them maps to the set of all elements of the corresponding type (Fig. 2) in the OpenSLEX meta model structure (Definition 2). Given a connected meta model, we define the following:

1. *allDatamodels*: the set of all data models, i.e.,  $DM$ .
2. *allClasses*: the set of all classes, i.e.,  $CL$ .
3. *allAttributes*: the set of all class attributes, i.e.,  $AT$ .
4. *allRelationships*: the set of all class relationships, i.e.,  $RS$ .
5. *allObjects*: the set of all objects, i.e.,  $OC$ .
6. *allVersions*: the set of all object versions, i.e.,  $OV$ .
7. *allRelations*: the set of all relations, i.e.,  $REL$ .
8. *allEvents*: the set of all events, i.e.,  $EV$ .
9. *allActivityInstances*: the set of all activity instances, i.e.,  $AI$ .
10. *allCases*: the set of all cases, i.e.,  $CS$ .
11. *allLogs*: the set of all logs, i.e.,  $LG$ .
12. *allActivities*: the set of all activities, i.e.,  $AC$ .
13. *allProcesses*: the set of all processes, i.e.,  $PM$ .

### 3.1.2 Elements Related to Elements



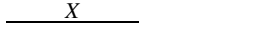

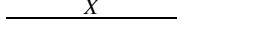
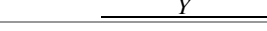
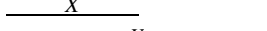
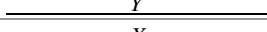
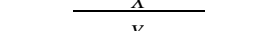
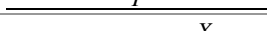
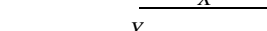
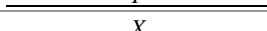
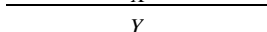
The following 14 functions take as an input a set of elements of the same type and return a set of elements related to them of the type corresponding to the return type of the function, e.g., a call to *eventsOf(es)*, being  $es \in \mathcal{P}(LG)$  will return the set of events that are related to the logs in the set  $es$ . Thanks to the subtype hierarchy, in most of the cases, we can reuse the same function call for input sets of any type, which leads to a more compact syntax. In the cases when an input of any type would not make sense, we can still restrict the input type to a particular kind, as is the case with the function *versionsRelatedTo*, which only accepts sets of object versions as input.

14. *datamodelsOf*  $\in MMSets \rightarrow \mathcal{P}(DM)$ : returns the set of data models related to the input.
15. *classesOf*  $\in MMSets \rightarrow \mathcal{P}(CL)$ : returns the set of classes related to the input.
16. *attributesOf*  $\in MMSets \rightarrow \mathcal{P}(A)$ : returns the set of attributes related to the input.
17. *relationshipsOf*  $\in MMSets \rightarrow \mathcal{P}(RS)$ : returns the set of relationships related to the input.
18. *objectsOf*  $\in MMSets \rightarrow \mathcal{P}(O)$ : returns the set of objects related to the input.
19. *versionsOf*  $\in MMSets \rightarrow \mathcal{P}(OV)$ : returns the set of object versions related to the input.
20. *relationsOf*  $\in MMSets \rightarrow \mathcal{P}(REL)$ : returns the set of relations related to the input.
21. *eventsOf*  $\in MMSets \rightarrow \mathcal{P}(E)$ : returns the set of events related to the input.
22. *activityInstancesOf*  $\in MMSets \rightarrow \mathcal{P}(AI)$ : returns the set of activity instances related to the input.
23. *activitiesOf*  $\in MMSets \rightarrow \mathcal{P}(AC)$ : returns the set of activities related to the input.
24. *casesOf*  $\in MMSets \rightarrow \mathcal{P}(CS)$ : returns the set of cases related to the input.
25. *logsOf*  $\in MMSets \rightarrow \mathcal{P}(LG)$ : returns the set of logs related to the input.
26. *processesOf*  $\in MMSets \rightarrow \mathcal{P}(PM)$ : returns the set of processes related to the input.
27. *versionsRelatedTo*  $\in \mathcal{P}(OV) \rightarrow \mathcal{P}(OV)$ : returns the set of object versions directly related (distance 1) to the input object versions through any kind of relationship.

### 3.1.3 Computation of Temporal Values

Some elements in our meta model contain temporal properties (e.g., events have timestamps, object versions have life spans, etc.) which allows making temporal computations with them. To do so, we provide the following 8 functions to compute time periods (with a start and an end), as well as durations. Durations (*DUR*) are special in the sense that they can be considered as a scalar and are not part of the

**Table 1** Relations in Allen's interval algebra

Relation	Name	Illustration	Interpretation
$X < Y$	before		X takes place before Y
$Y > X$	after		
$X \text{ m } Y$	meets		X meets Y ( <i>i</i> stands for <i>inverse</i> )
$Y \text{ mi } X$	meetsInv		
$X \text{ o } Y$	overlaps		X overlaps with Y
$Y \text{ oi } X$	overlapsInv		
$X \text{ s } Y$	starts		X starts Y
$Y \text{ si } X$	startsInv		
$X \text{ d } Y$	during		X during Y
$Y \text{ di } X$	duringInv		
$X \text{ f } Y$	finishes		X finishes Y
$Y \text{ fi } X$	finishesInv		
$X = Y$	matches		X is equal to Y

*MMElement* subtype hierarchy. Durations can only be used to be compared with other durations.

28.  $periodsOf \in MMSets \rightarrow \mathcal{P}(PER)$ : returns the computed periods for each of the elements of the input set.
29.  $globalPeriodOf \in MMSets \rightarrow PER$ : returns a global period for all the elements in the input set, i.e., the period from the earliest to the latest timestamp.
30.  $createPeriod \in TS \times TS \rightarrow PER$ : returns a period for the specified start and end timestamps.
31.  $getDuration \in PER \rightarrow DUR$ : returns the duration of a period in milliseconds.
32.  $Duration.ofSeconds \in \mathbb{N} \rightarrow DUR$ : returns the duration of the specified seconds.<sup>4</sup>
33.  $Dduration.ofMinutes \in \mathbb{N} \rightarrow DUR$ : returns the duration of the specified minutes.
34.  $Duration.ofHours \in \mathbb{N} \rightarrow DUR$ : returns the duration of the specified hours.
35.  $Duration.ofDays \in \mathbb{N} \rightarrow DUR$ : returns the duration of the specified days.

### 3.1.4 Temporal Interval Algebra

Allen's interval algebra, described in [1], introduces a calculus for temporal reasoning that defines possible relations between time intervals. It provides the tools to reason about the temporal descriptions of events in the broadest sense. Table 1 shows the 13 base relations between two intervals. These temporal relations

<sup>4</sup>  $\mathbb{N}$  is the set of natural numbers.

are used in our approach to reason about data elements for which we can compute a temporal interval.

We have introduced the functions to compute and create periods of time. The following 13 functions cover all the interval operators, described by Allen's interval algebra, that we can use to compare periods. Take  $(a, b)$  to be a pair of periods for which:

36.  $before \in PER \times PER \rightarrow \mathbb{B}$ :  $before(a, b)$  iff  $a$  takes place before  $b$ .<sup>5</sup>
37.  $after \in PER \times PER \rightarrow \mathbb{B}$ :  $after(a, b)$  iff  $a$  takes place after  $b$ .
38.  $meets \in PER \times PER \rightarrow \mathbb{B}$ :  $meets(a, b)$  iff the end of  $a$  is equal to the start of  $b$ .
39.  $meetsInv \in PER \times PER \rightarrow \mathbb{B}$ :  $meetsInv(a, b)$  iff the start  $a$  is equal to the end of  $b$ .
40.  $overlaps \in PER \times PER \rightarrow \mathbb{B}$ :  $overlaps(a, b)$  iff the end of  $a$  happens during  $b$ .
41.  $overlapsInv \in PER \times PER \rightarrow \mathbb{B}$ :  $overlapsInv(a, b)$  iff the start of  $a$  happens during  $b$ .
42.  $starts \in PER \times PER \rightarrow \mathbb{B}$ :  $starts(a, b)$  iff both start at the same time, but  $a$  is shorter.
43.  $startsInv \in PER \times PER \rightarrow \mathbb{B}$ :  $startsInv(a, b)$  iff both start at the same time, but  $a$  is longer.
44.  $during \in PER \times PER \rightarrow \mathbb{B}$ :  $during(a, b)$  iff  $a$  starts after  $b$  started and ends before  $b$  ends.
45.  $duringInv \in PER \times PER \rightarrow \mathbb{B}$ :  $duringInv(a, b)$  iff  $a$  starts before  $b$  starts and ends after  $b$  ends.
46.  $finishes \in PER \times PER \rightarrow \mathbb{B}$ :  $finishes(a, b)$  iff both end at the same time, but  $a$  is shorter.
47.  $finishesInv \in PER \times PER \rightarrow \mathbb{B}$ :  $finishesInv(a, b)$  iff both end at the same time, but  $a$  is longer.
48.  $matches \in PER \times PER \rightarrow \mathbb{B}$ :  $matches(a, b)$  iff both have the same start and end.

### 3.1.5 Operators on Attributes of Elements

Events, object versions, cases, and logs of the OpenSLEX meta model can be enriched with attribute values. The following functions allow the user to obtain their values:

49.  $eventHasAttribute \in EVAT \times EV \rightarrow \mathbb{B}$ :  $true$  iff the event contains a value for a certain attribute name.
50.  $versionHasAttribute \in AT \times OV \rightarrow \mathbb{B}$ :  $true$  iff the object version contains a value for a certain attribute name.
51.  $caseHasAttribute \in CSAT \times CS \rightarrow \mathbb{B}$ :  $true$  iff the case contains a value for a certain attribute name.

---

<sup>5</sup>  $\mathbb{B}$  is the set of Boolean values.

52.  $\text{logHasAttribute} \in \text{LGAT} \times \text{LG} \rightarrow \mathbb{B}$ : *true* iff the log contains a value for a certain attribute name.
53.  $\text{getAttributeEvent} \in \text{EVAT} \times \text{EV} \not\rightarrow V$ : returns the value for an attribute of an event.
54.  $\text{getAttributeVersion} \in \text{AT} \times \text{OV} \not\rightarrow V$ : returns the value for an attribute of an object version.
55.  $\text{getAttributeCase} \in \text{CSAT} \times \text{CS} \not\rightarrow V$ : returns the value for an attribute of a case.
56.  $\text{getAttributeLog} \in \text{LGAT} \times \text{LG} \not\rightarrow V$ : returns the value for an attribute of a log.
57.  $\text{versionChange} \in \text{AT} \times V \times V \times \text{OV} \rightarrow \mathbb{B}$ : *true* iff the value for an attribute linked to an object version changed from a certain value (in the previous object version) to another (in the provided object version).

By definition,  $\text{getAttribute}^*$  functions (items 53 to 56) are defined for combinations of elements and attributes for which the corresponding  $^*\text{HasAttribute}$  function (items 49 to 52) evaluates to *true*.

### 3.1.6 Abstract Syntax

The abstract syntax of DAPOQ-Lang is defined using the notation proposed in [8]. In DAPOQ-Lang, a query is a sequence of *Assignments* combined with an *ElementSet*:

$$\begin{aligned} \text{Query} &\triangleq s : \text{Assignments}; es : \text{ElementSet} \\ \text{Assignments} &\triangleq \text{Assignment}^* \end{aligned}$$

The result of a query is an *ElementSet*, i.e., the set of elements (of the same type) from the queried OpenSLEX dataset that satisfies certain criteria. An *Assignment* assigns an *ElementSet* to a variable. Then, any reference to such variable, via its identifier, will be replaced by the corresponding *ElementSet*.

$$\begin{aligned} \text{Assignment} &\triangleq v : \text{Varname}; es : \text{ElementSet} \\ \text{Varname} &\triangleq \text{identifier} \end{aligned}$$

An *ElementSet* can be defined over other *ElementSets* by *Construction* or *Application*. It can also be defined by means of a variable identifier, i.e., an *ElementSetVar*, by a call to a terminal element function with an *ElementSetTerminal*

(Sect. 3.1.1), by computation or creation of *Periods*, or by filtering elements of the previous options with a *FilteredElementSet*.

$$\begin{aligned} \textit{ElementSet} &\triangleq \textit{Construction} \mid \textit{Application} \mid \textit{Period} \mid \textit{ElementSetVar} \mid \\ &\quad \textit{ElementSetTerminal} \mid \textit{FilteredElementSet} \\ \textit{ElementSetVar} &\triangleq \textit{identifier} \end{aligned}$$

An *ElementSetTerminal* is the *ElementSet* resulting from a call to the corresponding terminal element function (e.g., *allEvents*).

$$\begin{aligned} \textit{ElementSetTerminal} &\triangleq \textit{AllDatamodels} \mid \textit{AllClasses} \mid \textit{AllAttributes} \mid \\ &\quad \textit{AllRelationships} \mid \textit{AllObjects} \mid \textit{AllVersions} \mid \\ &\quad \textit{AllRelations} \mid \textit{AllActivityInstances} \mid \textit{AllEvents} \mid \\ &\quad \textit{AllCases} \mid \textit{AllLogs} \mid \textit{AllActivities} \mid \textit{AllProcesses} \end{aligned}$$

An *ElementSet* can be composed from other *ElementSets* by applying set operations such as *union*, *exclusion*, and *intersection*.

$$\begin{aligned} \textit{Construction} &\triangleq es_1, es_2 : \textit{ElementSet}; o : \textit{Set\_Op} \\ \textit{Set\_Op} &\triangleq \textit{Union} \mid \textit{Excluding} \mid \textit{Intersection} \end{aligned}$$

Also, an *ElementSet* can be constructed by means of a call to one of the *ElementOf\_Op* functions, which includes the functions described in Sect. 3.1.2 that return sets of elements related to other elements, and the *periodsOf* function described in Sect. 3.1.3 that computes the periods of elements.

$$\begin{aligned} \textit{Application} &\triangleq es : \textit{ElementSet}; o : \textit{ElementOf\_Op} \\ \textit{ElementOf\_Op} &\triangleq \textit{datamodelsOf} \mid \textit{classesOf} \mid \textit{attributesOf} \mid \textit{relationshipsOf} \mid \\ &\quad \textit{objectsOf} \mid \textit{versionsOf} \mid \textit{relationsOf} \mid \textit{eventsOf} \mid \\ &\quad \textit{activityInstancesOf} \mid \textit{casesOf} \mid \textit{activitiesOf} \mid \textit{logsOf} \mid \\ &\quad \textit{processesOf} \mid \textit{periodsOf} \mid \textit{versionsRelatedTo} \end{aligned}$$

An *ElementSet* can be built by means of filtering, discarding elements of another *ElementSet* according to certain criteria. These criteria are expressed as a *PredicateBlock*, which will be evaluated for each member of the input *ElementSet*. Depending on the result of evaluating the *PredicateBlock*, each element will be filtered out or included in the new *ElementSet*.

$$\textit{FilteredElementSet} \triangleq es : \textit{ElementSet}; pb : \textit{PredicateBlock}$$



A *PredicateBlock* is a sequence of *Assignments* combined with a *Predicate*. Such *Predicate* can be recursively defined as a binary (*and*, *or*) or unary (*not*) combination of other *Predicates*.

$$\begin{aligned}
 \text{PredicateBlock} &\triangleq s : \text{Assignments}; p : \text{Predicate} \\
 \text{Predicate} &\triangleq \text{AttributePredicate} \mid \text{Un\_Predicate} \mid \text{Bin\_Predicate} \mid \\
 &\quad \text{TemporalPredicate} \\
 \text{Bin\_Predicate} &\triangleq p_1, p_2 : \text{Predicate}; o : \text{BinLogical\_Op} \\
 \text{Un\_Predicate} &\triangleq p : \text{Predicate}; o : \text{UnLogical\_Op} \\
 \text{BinLogical\_Op} &\triangleq \text{And} \mid \text{Or} \\
 \text{UnLogical\_Op} &\triangleq \text{Not}
 \end{aligned}$$

Also, a *Predicate* can be defined as an *AttributePredicate*, which either refers to *AttributeExists* function that checks the existence of an attribute for a certain element, an operation on attribute values (e.g., to compare attributes to substrings, constants, or other attributes), or an *AttributeChange* predicate, making use of the functions specified in Sect. 3.1.5.

$$\begin{aligned}
 \text{AttributePredicate} &\triangleq \text{AttributeExists} \mid \text{AttributeValuePred} \mid \text{AttributeChange} \\
 \text{AttributeExists} &\triangleq at : \text{AttributeName} \\
 \text{AttributeValuePred} &\triangleq at_1, at_2 : \text{Attribute}; o : \text{Value\_Op} \\
 \text{AttributeChange} &\triangleq at : \text{AttributeName}; from, to : \text{Value} \\
 \text{AttributeName} &\triangleq \text{identifier} \\
 \text{Value\_Op} &\triangleq == \mid >= \mid <= \mid > \mid < \mid \text{startsWith} \mid \text{endsWith} \mid \text{contains} \\
 \text{Attribute} &\triangleq \text{AttributeName} \mid \text{Value} \\
 \text{Value} &\triangleq \text{literal}
 \end{aligned}$$

Finally, a *Predicate* can be defined as a *TemporalPredicate*, i.e., a Boolean operation comparing periods or durations. *Period* comparisons based on Allen's Interval Algebra are supported by the functions defined in Sect. 3.1.4. *Duration* comparisons are done on simple scalars (e.g., ==, >, <, ≥, and ≤). A *Period* can be either created from some provided timestamps with the function *createPeriod* or computed as the global period of an element or a set of elements with the function *globalPeriodOf*. Also, a *Period* can be constructed referring to a variable containing another period by means of an identifier. *Durations* can be obtained from existing

periods (with the function *durationOf*) or created from specific durations in seconds, minutes, hours, or days with the functions defined in Sect. 3.1.3.

$$\begin{aligned}
\textit{TemporalPredicate} &\triangleq \textit{per}_1, \textit{per}_2 : \textit{Period}; \textit{o} : \textit{Period\_Op} \mid \\
&\quad \textit{dur}_1, \textit{dur}_2 : \textit{Duration}; \textit{o} : \textit{Numerical\_Comp\_Op} \\
\textit{Period} &\triangleq \textit{PeriodCreation} \mid \textit{PeriodVar} \\
\textit{PeriodCreation} &\triangleq \textit{ts}_1, \textit{ts}_2 : \textit{Timestamp}; \textit{o} : \textit{createPeriod} \mid \\
&\quad \textit{es} : \textit{ElementSet}; \textit{o} : \textit{globalPeriodOf} \\
\textit{PeriodVar} &\triangleq \textit{identifier} \\
\textit{Period\_Op} &\triangleq \textit{before} \mid \textit{after} \mid \textit{meets} \mid \textit{meetsInv} \mid \textit{overlaps} \mid \\
&\quad \textit{overlapsInv} \mid \textit{starts} \mid \textit{startsInv} \mid \textit{during} \mid \\
&\quad \textit{duringInv} \mid \textit{finishes} \mid \textit{finishesInv} \mid \textit{matches} \\
\textit{Duration} &\triangleq \textit{p} : \textit{Period}; \textit{o} : \textit{getDuration} \mid \textit{v} : \textit{Value}; \textit{o} : \textit{DurationOf} \\
\textit{DurationOf} &\triangleq \textit{Duration.ofSeconds} \mid \textit{Duration.ofMinutes} \mid \\
&\quad \textit{Duration.ofHours} \mid \textit{Duration.ofDays} \\
\textit{Numerical\_Comp\_Op} &\triangleq == \mid >= \mid <= \mid > \mid <
\end{aligned}$$

Figure 3 shows the syntax tree of Query 1 according to the presented abstract syntax. It also contains elements of the proposed concrete syntax, presented in detail in Sect. 5, to demonstrate the mapping to real DAPOQ-Lang queries.

## 3.2 Semantics

In this section, we make use of denotational semantics, as proposed in [8], to describe the meaning of DAPOQ-Lang queries. We define function  $M_T$  to describe the meaning of the nonterminal term  $T$  (e.g.,  $M_{Query}$  describes the meaning of the nonterminal  $Query$ ). First, we introduce a notation of overriding union that will be used in further discussions.

**Definition 3 (Overriding Union)** The overriding union of  $f : X \rightarrow Y$  by  $g : X \rightarrow Y$  is denoted as  $f \oplus g : X \rightarrow Y$  such that  $dom(f \oplus g) = dom(f) \cup dom(g)$  and

$$f \oplus g(x) = \begin{cases} g(x) & \text{if } x \in dom(g) \\ f(x) & \text{if } x \in dom(f) \setminus dom(g). \end{cases}$$

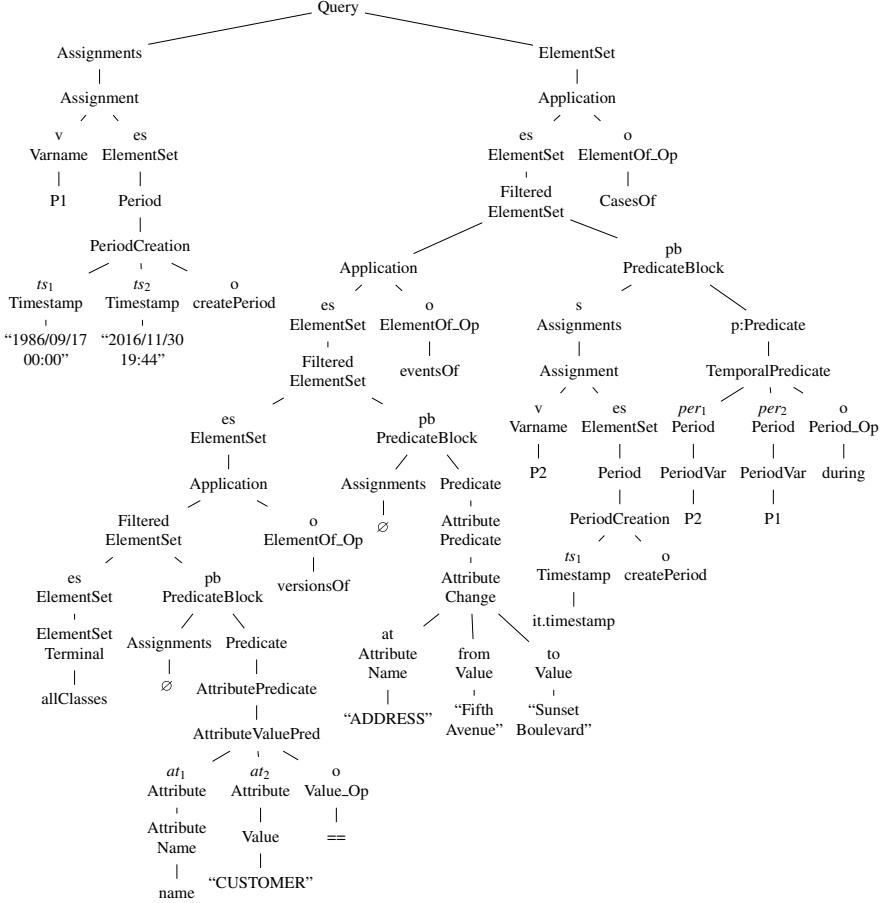


Fig. 3 Syntax tree of DAPOQ-Lang Query 1

In the previous section, we have introduced the use of variables in the language. These variables must be translated into a value in *MMSets* (Eq. 1) during the execution of our queries. A *Binding* assigns a set of elements to a variable name:

$$Binding \triangleq Varname \rightarrow MMSets$$

Queries are computed based on a dataset complying with the structure of the OpenSLEX meta model. Such a meta model can be seen as a tuple of sets of elements of each of the basic types:

$$MetaModel \triangleq (AC, LG, EV, REL, OC, AT, CL, PER, PM, CS, AI, OV, RS, DM)$$

The meaning function of a query takes a query and a meta model dataset as an input and returns a set of elements that satisfy the query:

$$M_{Query} : Query \times MetaModel \rightarrow MMSets$$

This function is defined as

$$M_{Query}[q : Query, MM : MetaModel] \triangleq M_{ElementSet}(q.es, MM, M_{Assignments}(q.s, MM, \emptyset))$$

The evaluation of the query meaning function  $M_{Query}$  depends on the evaluation of the assignments and the element set involved. Evaluating the assignments means resolving their corresponding element sets and remembering the variables to which they were assigned.

$$M_{Assignments} : Assignments \times MetaModel \times Binding \rightarrow Binding$$

A sequence of assignments resolves to a binding, which links sets of elements to variable names. Assignments that happen later in the order of declaration take precedence over earlier ones when they share the variable name.

$$M_{Assignments}[s : Assignments, MM : MetaModel, B : Binding] \triangleq$$

**if**  $\neg(s.TAIL).EMPTY$  **then**

$$M_{Assignments}(s.TAIL, MM, B \oplus M_{Assignment}(s.FIRST, MM, B))$$

**else**  $B$

The result of an assignment is a binding, linking a set of elements to a variable name.

$$M_{Assignment} : Assignment \times MetaModel \times Binding \rightarrow Binding$$

$$M_{Assignment}[a : Assignment, MM : MetaModel, B : Binding] \triangleq$$

$$\{(a.v, M_{ElementSet}(a.es, MM, B))\}$$

An *ElementSet* within the context of a meta model and a binding returns a set of elements of the same type that satisfy the restrictions imposed by the *ElementSet*.

$$M_{ElementSet} : ElementSet \times MetaModel \times Binding \rightarrow MMSets$$

An *ElementSet* can be resolved as a *Construction* of other *ElementSets* with the well-known set operations of union, exclusion, and intersection. It can be the result of evaluating an *Application* function, returning elements related to other elements, the result of the creation of *Periods*, or the value of a variable previously declared

(*ElementSetVar*). Also, it can be the result of a terminal *ElementSet*, e.g., the set of all the events (*allEvents*). Finally, an *ElementSet* can be the result of filtering another *ElementSet* according to *PredicateBlock*, which is a *Predicate* preceded by a sequence of *Assignments*. These *Assignments* are only valid within the scope of the *PredicateBlock* and are not propagated outside of it (i.e., if a variable is reassigned, it will maintain its original value outside of the *PredicateBlock*). The resulting *FilteredElementSet* will contain only the elements of the input *ElementSet* for which the evaluation of the provided *Predicate* is *true*.

$M_{ElementSet}[es : ElementSet, MM : MetaModel, B : Binding] \triangleq$

**case** *es* **of**

*Construction*  $\Rightarrow$

**case** *es.o* **of**

*Union*  $\Rightarrow M_{ElementSet}(es.es_1, MM, B) \cup M_{ElementSet}(es.es_2, MM, B)$

*Excluding*  $\Rightarrow M_{ElementSet}(es.es_1, MM, B) \setminus M_{ElementSet}(es.es_2, MM, B)$

*Intersection*  $\Rightarrow M_{ElementSet}(es.es_1, MM, B) \cap M_{ElementSet}(es.es_2, MM, B)$

**end**

*Application*  $\Rightarrow es.o(M_{ElementSet}(es.es, MM, B))$

*Period*  $\Rightarrow M_{Period}(es, MM, B)$

*ElementSetVar*  $\Rightarrow \begin{cases} B(es) & \text{if } es \in dom(B) \\ \emptyset & \text{otherwise} \end{cases}$

*ElementSetTerminal*  $\Rightarrow es^{MM}$

*FilteredElementSet*  $\Rightarrow \{e \in M_{ElementSet}(es.es, MM, B) \mid$

$M_{Predicate}(es.pb.p, MM, M_{Assignments}(es.pb.s, MM, B \oplus (it, e)))\}$

**end**

A *Predicate* is evaluated as a Boolean, with respect to a *MetaModel* and a *Binding*:

$M_{Predicate} : Predicate \times MetaModel \times Binding \rightarrow \mathbb{B}$

The meaning function of *Predicate* evaluates to a Boolean value, which can be recursively constructed combining binary (*and*, *or*) or unary (*not*) predicates. Also, a *Predicate* can be defined as an *AttributePredicate* that evaluates the existence of attributes, comparisons of attribute values, or attribute value changes. Finally, a *Predicate* can be defined as a *TemporalPredicate*, which can compare durations or periods by means of Allen's Interval Algebra operators.

$$M_{\text{Predicate}}[p : \text{Predicate}, MM : \text{MetaModel}, B : \text{Binding}] \triangleq$$

**case**  $p$  **of**

*AttributePredicate*  $\Rightarrow$

**case**  $p$  **of**

*AttributeExists*  $\Rightarrow$  **if**  $B(it) \in EV : \text{eventHasAttribute}(p.at, B(it))$

**elif**  $B(it) \in OV : \text{versionHasAttribute}(p.at, B(it))$

**elif**  $B(it) \in CS : \text{caseHasAttribute}(p.at, B(it))$

**elif**  $B(it) \in LG : \text{logHasAttribute}(p.at, B(it))$

**else** :  $\emptyset$

*AttributeValuePred*  $\Rightarrow$

$p.o(M_{\text{Attribute}}(p.at_1, B(it), MM), M_{\text{Attribute}}(p.at_2, B(it), MM))$

*AttributeChange*  $\Rightarrow$

**if**  $B(it) \in MM.OV$  **then** :  $\text{versionChange}(p.at, p.from, p.to, B(it))$  **else** :  $\emptyset$

**end**

*Un\_Predicate*  $\Rightarrow \neg M_{\text{Predicate}}(p.p, MM, B)$

*Bin\_Predicate*  $\Rightarrow$

**case**  $p.o$  **of**

*And*  $\Rightarrow M_{\text{predicate}}(p.p_1, MM, B) \wedge M_{\text{predicate}}(p.p_2, MM, B)$

*Or*  $\Rightarrow M_{\text{predicate}}(p.p_1, MM, B) \vee M_{\text{predicate}}(p.p_2, MM, B)$

**end**

*TemporalPredicate*  $\Rightarrow$

**case**  $p.o$  **of**

*Period\_Op*  $\Rightarrow p.o(M_{\text{Period}}(p.per_1, MM, B), M_{\text{Period}}(p.per_2, MM, B))$

*Duration\_Op*  $\Rightarrow p.o(M_{\text{Duration}}(p.dur_1, MM, B), M_{\text{Duration}}(p.dur_2, MM, B))$

**end**

**end**

A *Period* for a given meta model dataset and a binding returns an instance of *PER*, i.e., a single period element:

$$M_{\text{Period}} : \text{Period} \times \text{MetaModel} \times \text{Binding} \rightarrow \text{PER}$$

The meaning function of *Period* will return a period element that can be created (*PeriodCreation*) or assigned from a variable name containing a period (*PeriodVar*). In the case of a *PeriodCreation*, a period can be created for the specified start and end timestamps using the *createPeriod* function or it can be computed as the global period of another set of periods (*globalPeriodOf*).

$$M_{Period}[p : Period, MM : MetaModel, B : Binding] \triangleq$$

**case**  $p$  **of**

*PeriodCreation*  $\Rightarrow$

**case**  $p.o$  **of**

*createPeriod*  $\Rightarrow p.o(p.ts_1, p.ts_2)$

*globalPeriodOf*  $\Rightarrow p.o^{MM}(M_{ElementSet}(p.es, MM, B))$

**end**

*PeriodVar*  $\Rightarrow \begin{cases} B(p) & \text{if } p \in dom(B) \\ \emptyset & \text{otherwise} \end{cases}$

**end**

A *Duration* is a value representing the length of a period, and it is computed within the context of a meta model dataset and a binding:

$$M_{Duration} : Duration \times MetaModel \times Binding \rightarrow DUR$$

A *Duration* can be evaluated based on the duration of a period (*getDuration*) or a duration specified in scalar units (*DurationOf*).

$$M_{Duration}[d : Duration, MM : MetaModel, B : Binding] \triangleq$$

**case**  $d.o$  **of**

*DurationOf*  $\Rightarrow d.o(d.v)$

*getDuration*  $\Rightarrow d.o(M_{Period}(d.p, MM, B))$

**end**

Finally, an *Attribute* is a value assigned to an element in the context of a meta model:

$$M_{Attribute} : Attribute \times Element \times MetaModel \rightarrow Value$$

In order to evaluate the value of an *Attribute*, we can refer to the *AttributeName*, in which case the value will be obtained in different ways depending on the type

of element (event, object version, case, or log). Also, an *Attribute* can be explicitly defined by its *Value*.

$$M_{Attribute}[at : Attribute, e : Element, MM : MetaModel] \triangleq$$

**case** *at* **of**

*AttributeName*  $\Rightarrow$

**case** *e* **of**

*Event*  $\Rightarrow$  **if** *eventHasAttribute(at, e)* **then** : *getAttributeEvent(at, e)* **else** :  $\emptyset$

*Version*  $\Rightarrow$  **if** *versionHasAttribute(at, e)* **then** : *getAttributeVersion(at, e)*

**else** :  $\emptyset$

*Case*  $\Rightarrow$  **if** *caseHasAttribute(at, e)* **then** : *getAttributeCase(at, e)* **else** :  $\emptyset$

*Log*  $\Rightarrow$  **if** *logHasAttribute(at, e)* **then** : *getAttributeLog(at, e)* **else** :  $\emptyset$

**end**

*Value*  $\Rightarrow$  *at*

**end**

This concludes the formal definition of DAPOQ-Lang in terms of syntax and semantics at an abstract level. The coming sections provide some details about the concrete syntax, implementation, and its performance.

## 4 Implementation and Evaluation

DAPOQ-Lang<sup>6</sup> has been implemented as a Domain Specific Language (DSL) on top of Groovy<sup>7</sup>, a dynamic language for the Java platform. This means that, on top of all the functions and operators provided by DAPOQ-Lang, *any syntax allowed by Groovy or Java can be used within DAPOQ-Lang queries*. DAPOQ-Lang heavily relies on a Java implementation of the OpenSLEX<sup>8</sup> meta model using SQLite<sup>9</sup> as a storage and querying engine. However, DAPOQ-Lang abstracts from the specific storage choice, which allows it to run on any SQL database and not just SQLite. The platform PADAS<sup>10</sup> (Process Aware Data Suite) integrates DAPOQ-Lang and

<sup>6</sup> <https://github.com/edugonza/DAPOQ-Lang/>.

<sup>7</sup> <http://groovy-lang.org/>.

<sup>8</sup> <https://github.com/edugonza/OpenSLEX/>.

<sup>9</sup> <https://www.sqlite.org>.

<sup>10</sup> <https://github.com/edugonza/PADAS/>.



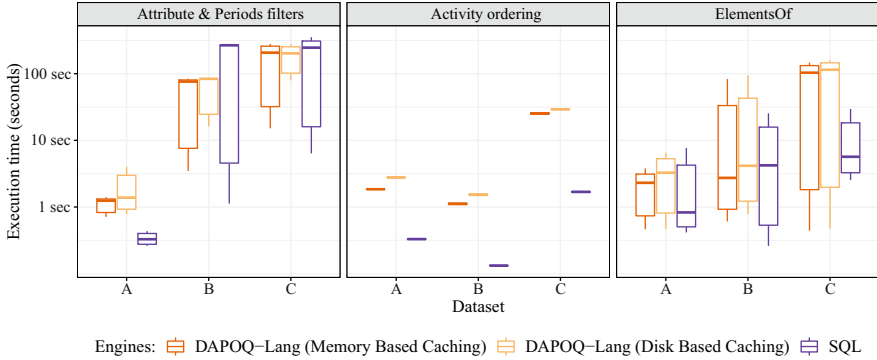
**Table 2** Characteristics of the three datasets employed in the evaluation

Dataset	# Objects	# Versions	# Events	# Cases	# Logs	# Activities
A	6740	8424	8512	108, 751	34	14
B	7, 339, 985	7, 340, 650	26, 106	82, 113	10, 622	172
C	162, 287	277, 094	277, 094	569, 026	29	62

OpenSLEX in a user-friendly environment to process the data and run queries. The current implementation relies on the SQLite library to store the data and execute certain subqueries. Therefore, it is to be expected that DAPOQ-Lang introduces certain overhead, given that data retrieval and object creation on the client side consume extra time and memory compared to an equivalent SQL query. In order to assess the impact of DAPOQ-Lang on query performance, we run a benchmark of pairs of equivalent queries, as expressed in DAPOQ-Lang and SQL, on the same database. The queries are organized in 3 categories and run on the 3 datasets described in [4]: A (event data obtained from the redo logs of a simulated ticket selling platform), B (event records from a financial organization), and C (ERP event data from a sample SAP system) (Table 2).

The DAPOQ-Lang queries of each pair were run with two different configurations: memory-based and disk-based caching. *Memory-based caching* uses the heap to store all the elements retrieved from the database during the execution of the query. This is good for speed when dealing with small or medium size datasets but represents a big limitation to deal with big datasets given the impact on memory use and garbage collection overhead. *Disk-based caching* makes use of MapDB,<sup>11</sup> a disk-based implementation of Java hash maps, to serialize and store on disk all the elements retrieved from the database. This significantly reduces the memory consumption and allows handling much larger datasets, which comes at the cost of speed given the overhead introduced by serialization and disk I/O operations. Figure 4 shows the results of the benchmark, with one plot per query type, one box per query engine (SQL, DAPOQ-Lang, and DAPOQ-Lang with disk caching), for the three datasets. As expected, we observe that the performance of DAPOQ-Lang queries is, in general, poorer than that of the equivalent SQL queries, especially when it comes to queries regarding the order of activities. This is due to the overhead on transmission and processing of data and the fact that many filtering operations are performed on the client instead of the server side. Obviously, there is a trade-off between ease of use and performance. Nevertheless, performance was never the main motivation for the development of DAPOQ-Lang, but ease of use and speed of query writing. In future versions, further efforts will be made to improve performance and to provide more comprehensive benchmarks.

<sup>11</sup> <http://www.mapdb.org>.



**Fig. 4** Benchmark of queries run with DAPOQ-Lang, DAPOQ-Lang with disk-based caching, and SQL on an SQLite backend. Note that the vertical axis is logarithmic

## 5 Application and Use Cases

The purpose of this section is to demonstrate the applicability of our approach and tools. First, we explore the professional profiles, in the context of process mining, to which this language is directed to, and we identify the most common data aspects to query given each profile. Then, we provide some use cases of DAPOQ-Lang with examples of relevant queries for each data aspect. Finally, we compare DAPOQ-Lang to SQL by means of an example. The example highlights the expressiveness and compactness of our query language.

### 5.1 Business Questions in Process Mining

Process mining is a broad field, with many techniques available tailored toward a variety of analysis questions. “Process miners” (analysts or users carrying out a process mining project) are often interested in discovering process models from event data. Sometimes these models are provided beforehand and the focus is on conformance between the models and events. It can be the case that assessing the performance of specific activities is critical. Also, finding bottlenecks in the process can be of interest for the analysts. In some contexts, where existing regulations and guidelines impose restrictions on what is allowed and what is not in the execution of a process, compliance checking becomes a priority. In the literature, we can find examples of frequently posed questions for specific domains, like healthcare [7], in which root cause analysis becomes relevant in order to trace back data related to a problematic case. All these perspectives pose different challenges to process miners, who need to “dig” into the data to find answers to relevant questions.

Previous works [6, 9] identified professional roles and profiles in the area of business process management by analyzing job advertisements and creating a

classification based on the competencies. We make use of this classification to point out the corresponding data aspects relevant for each profile. Table 3 presents, in the two leftmost columns, the classification of the roles, as proposed by the authors of studies [6, 9]. In the column *Main Focus*, we propose, based on the role description, the sub-disciplines of process mining and data engineering that become relevant for each job profile (i.e., discovery, compliance checking, conformance checking, performance analysis, root cause analysis, integration, and data integrity). The rest of the columns indicate whether certain event data aspects become particularly interesting to be queried for each professional role, considering the role description and the main focus. We have grouped these event data aspects into two big categories that reflect the expected output of the queries: (a) *specialized sublogs* are event logs that contain only event data that reflects certain desired properties (e.g., temporal constraints, activity occurrence constraints), and (b) *metrics, artifacts, and provenance* are the resulting values of the computation of certain event data properties (e.g., performance metrics).

We see that there is a clear distinction between roles interested in performance and root cause analysis, in contrast to those mainly interested in compliance. The former will need to obtain *performance metrics* from the data, e.g., average case duration, or most time-consuming tasks. Also, they are interested in finding data related to problematic cases, e.g., obtaining all the products purchased in an unpaid order (*dependency relations*), or finding out providers of a defective batch of products (*data lineage*). However, those with a focus on compliance typically need to answer questions related to *temporal* constraints (e.g., if cases of a particular type of client are resolved within the agreed SLAs), *activity occurrence* constraints (e.g., whether a purchase was always paid), and *order of actions* (e.g., if an invoice is created before a delivery is dispatched).

As the roles get more concerned with the technical aspects of IT systems, more focus is put on performance and data properties. Especially, for technical architects, data integrity is crucial, since they are the ones in charge of integrating both applications and data storage systems. Being able to filter information based on *data properties* and find irregular *data changes* is important to verify a correct integration of different infrastructures.

Now that we have identified data aspects of interest, in what follows we present a set of example DAPOQ-Lang queries. The aim of these examples is twofold: to serve as a template to write queries and to demonstrate that the features of DAPOQ-Lang indeed cover all the aspects described in Table 3.

## 5.2 Exporting Logs

One of the main purposes when querying process execution data is to export it as a compatible event log format. DAPOQ-Lang provides utilities to export logs, cases, and events as XES event logs, which can be further analyzed using process mining

**Table 3** Types of BPM professionals, according to [9], and relation to querying in process mining

Role [9]	Description [9]	Main focus	Specialized sublogs					Metrics, artifacts, and provenance		
			Temporal constraints	Activity occurrence	Order of actions	Data properties	Data changes	Data lineage	Dependency relations	Performance metrics
Business Process Analyst	Elicits, analyzes, documents, and communicates user requirements and designs according to business processes and IT systems; acts as a liaison between business and IT	Discovery, Compliance, and Conformance	✓	✓	✓	✓	✓			
Business Process Compliance Manager	Analyses regulatory requirements and ensures compliance of business processes and IT systems	Compliance and Conformance	✓	✓		✓				
Business Process Manager, Sales and Marketing	Designs sales processes and analyses requirements for related IT systems; supports and executes sales and marketing processes	Compliance	✓	✓	✓	✓				
Business Process Improvement Manager	Analyses, measures, and continuously improves business process, e.g., through application of Lean or Six Sigma management techniques	Performance, Conformance and Root Cause Analysis	✓	✓	✓	✓	✓		✓	✓
ERP Solution Architect	Implements business processes in ERP systems	Performance, Conformance and Root Cause Analysis	✓	✓	✓	✓	✓	✓	✓	✓
IT-Business Strategy Manager	Aligns business and IT strategies; monitors technological innovations and identifies business opportunities	Performance and Conformance	✓	✓		✓				✓
Technical Architect	Develops and integrates hardware and software infrastructures	Integration and Data Integrity				✓		✓	✓	✓

platforms such as ProM<sup>12</sup> or RapidProM.<sup>13</sup> The following queries show the way to export XES logs for different types of data. These functions can be applied to all the query types defined under the *Specialized Sublogs* category (Table 3) in order to extract the corresponding XES event log. When a set of logs is retrieved, an independent XES log is generated for each of them.

**Query 2** Export all the logs with a specific name. The result can be one or many logs being exported according to the XES format.

```
1 exportXLogsOf(allLogs().where{ name == "log01" })
```

When the input of *exportXLogsOf* is a set of cases, one single XES log is exported.

**Query 3** Export in a single XES log all the cases of different logs.

```
1 exportXLogsOf(casesOf(allLogs()).where { name.contains("1" )})
```

In the case of a set of events, a single XES log with a single trace is exported (Query 4).

**Query 4** Export in a single XES log all the events of different logs.

```
1 exportXLogsOf(eventsOf(allLogs()).where{ name.contains("1" )})
```

A special situation is when we want to export a set of logs or cases while filtering out events that do not comply with some criteria. In that case, we call *exportXLogsOf* with a second argument representing the set of events that can be exported (Query 5). Any event belonging to the log to be exported not contained in this set of events will be excluded from the final XES log.

**Query 5** Export one or many XES logs excluding all the events that do not belong to a specific subset.

```
1 exportXLogsOf(allLogs(), eventsOf(allClasses()).where { name == "BOOKING" })
```

### 5.3 Specialized Sublogs

So far, we have seen how to export logs as they are stored in the dataset under analysis. However, it is very common to focus on specific aspects of the data depending on the questions to answer. This means that we need to create specialized sublogs according to certain criteria. This section presents examples of queries to

<sup>12</sup> <http://www.promtools.org>.

<sup>13</sup> <http://www.rapidprom.org/>.

create specialized sublogs that comply with certain constraints in terms of *temporal properties*, *activity occurrence*, *order of action*, *data properties*, or *data changes*.

**Temporal Constraints** A way to create specialized sublogs is to filter event data based on temporal constraints. The creation and computation of periods makes it possible to select only data relevant during a certain time span. Query 6 returns events that happened during period  $p$  and that belong to the log “log01”.

**Query 6** Temporal constraints. Retrieve all the events of “log01” that happened during a certain period of time.

```

1 def evLog01 = eventsOf(allLogs().where{ name == "log01" })
2 def p = createPeriod("2014/11/27_15:57", "2014/11/27_16:00", "yyyy/MM/dd_HH:mm
  ↪ ")
3
4 eventsOf(p).intersection(evLog01)

```

Query 7 focuses on the duration of cases rather than on the specific time when they happened. Only cases of log “log01” with a duration longer than 11 minutes will be returned.

**Query 7** Temporal constraints. Retrieve cases of “log01” with a duration longer than 11 minutes. The variable “it” is used to iterate over all values of “c” within the “where” closure

```

1 def c = casesOf(allLogs().where{ name == "log01" })
2
3 c.where { globalPeriodOf(it).getDuration() > Duration.ofMinutes(11) }

```

**Activity Occurrence** Another way to select data is based on activity occurrence. The following query shows an example of how to retrieve cases in which two specific activities were performed regardless of the order. First, cases that include the first activity are retrieved (*casesA*). Then, cases that include the second activity are retrieved (*casesB*). Finally, the intersection of both sets of cases is returned.

**Query 8** Activity occurrence. Retrieve cases where activities that contain the words “INSERT” or “UPDATE” and “CUSTOMER” happened in the same case.

```

1 def actA = allActivities().where {
2   name.contains("INSERT") && name.contains("CUSTOMER") }
3
4 def actB = allActivities().where {
5   name.contains("UPDATE") && name.contains("CUSTOMER") }
6
7 def casesA = casesOf(actA)
8 def casesB = casesOf(actB)
9
10 casesA.intersection(casesB)

```

**Order of Actions** This time we are interested in cases in which the relevant activities happened in a specific order. The following query, an extended version of Query 8, selects the cases that include both activities. Yet, before storing the intersection of cases containing events of the activities in the set *actA* with cases

containing events of the activities in the set *actB* in a variable (line 13), the query performs a filter based on the order of these two activities. To do so, for each case, the set of events is retrieved (line 15). Next, the events of the first and second activities are selected (lines 16 and 17). Finally, the periods of both events are compared (line 18), evaluating the condition to the value *true* for each case in which all the events of activity *A* happened *before* the events of activity *B*. Only the cases for which the condition block (lines 14 to 18) evaluated to *true* are stored in the variable *casesAB* and returned.

**Query 9** Order of actions. Retrieve cases where activities that contain the words “INSERT” and “CUSTOMER” happen before activities that contain the words “UPDATE” and “CUSTOMER”.

```

1  def actA = allActivities().where {
2    name.contains("INSERT") && name.contains("CUSTOMER") }
3
4  def actB = allActivities().where {
5    name.contains("UPDATE") && name.contains("CUSTOMER") }
6
7  def casesA = casesOf(actA)
8  def casesB = casesOf(actB)
9
10 def eventsA = eventsOf(actA)
11 def eventsB = eventsOf(actB)
12
13 def casesAB = casesA.intersection(casesB)
14 .where {
15   def ev = eventsOf(it)
16   def evA = ev.intersection(eventsA)
17   def evB = ev.intersection(eventsB)
18   before(globalPeriodOf(evA), globalPeriodOf(evB))
19 }

```

**Data Properties** Some elements in our OpenSLEX dataset contain attributes that can be queried. These elements are object versions, events, cases, and logs. The following query shows how to filter events based on their attributes. First, the query compares the value of the attribute *resource* to a constant. Also, it checks if the attribute *ADDRESS* contains a certain substring. Finally, it verifies that the event contains the attribute *CONCERT\_DATE*. Only events that satisfy the first and either the second or the third will be returned as a result of the query.

**Query 10** Data properties. Retrieve events of resource “SAMPLE” that either have an attribute *ADDRESS* which value contains “35” or have a *CONCERT\_DATE* attribute.

```

1  allEvents().where {
2    resource == "SAMPLEDB" && (at.ADDRE.contains("35") || has(at.CONCERT_DATE))}

```

**Data Changes** An important feature of our query language is the function named *changed*. This function determines if the value of an attribute for a certain object version changed. The function has the attribute name as a required parameter (*at:*) and two optional parameters (*from:*, and *to:*). Query 11 returns all the events related to object versions for which the value of the attribute “BOOKING\_ID” changed. No restrictions are set on the specific values. Therefore, the call to *changed* will be

evaluated to *true* for an object version only if the value of the attribute in preceding version was different from the value in the current one.

**Query 11** Data changes. Retrieve events that affected versions where the value of “BOOKING\_ID” changed.

```
1 eventsOf(allVersions()).where { changed([at: "BOOKING_ID"]) }
```

Query 12 shows a similar example. This time we want to obtain the events related to object versions for which the attribute “SCHEDULED\_DATE” changed from “11-JUN-82” to a different one.

**Query 12** Data changes. Retrieve events that affected versions where the value of “SCHEDULED\_DATE” changed from “11-JUN-82” to a different value.

```
1 eventsOf(allVersions()).where { changed([at: "SCHEDULED_DATE", from: "11-JUN
  ↪ -82"]) }
```

Query 13 instead retrieves the events related to object versions for which the attribute “SCHEDULED\_DATE” changed to “22-MAY-73” from a different one.

**Query 13** Data changes. Retrieve events that affected versions where the value of “SCHEDULED\_DATE” changed to “22-MAY-73” from a different value.

```
1 eventsOf(allVersions()).where { changed([at: "SCHEDULED_DATE", to: "22-MAY-73"
  ↪ ]) }
```

Finally, Query 14 imposes a stricter restriction, retrieving only the events related to object versions for which the attribute “SCHEDULED\_DATE” changed from “24-MAR-98” to “22-MAY-73”.

**Query 14** Data changes. Retrieve events that affected versions where the value of “SCHEDULED\_DATE” changed from “24-MAR-98” to “22-MAY-73”.

```
1 eventsOf(allVersions()).where {
2   changed([at: "SCHEDULED_DATE", from: "24-MAR-98", to: "22-MAY-73"]) }
```

## 5.4 Metrics, Artifacts, and Provenance

In the previous section, we have seen examples of how to obtain specialized sublogs given certain criteria. However, we do not always want to obtain events, cases, or logs as the result of our queries. In certain situations, the interest is in data objects, and their relations to other elements of the dataset, e.g., objects of a certain type, artifacts that coexisted during a given period, or data linked to other elements. Also, one can be interested in obtaining performance metrics based on existing execution data. All these elements cannot be exported as an event log, since they do not always represent event data. However, they can be linked to related events or traces. This



section shows example queries that exploit these relations and provide results that cannot be obtained as plain event logs.

**Data Lineage** Data lineage focuses on the lifecycle of data, its origins, and where it is used over time. DAPOQ-Lang supports data lineage mainly with the *ElementsOf* functions listed in Sect. 3.1.2. These functions return elements of a certain type linked or related to input elements of another type. As an example, we may have an interest in obtaining all the products in the database affected by a catalog update process during a certain period in which prices were wrongly set. The following query finds the cases in log “log01” whose life span overlaps with a certain period and returns the object versions related to them.

**Query 15** Data lineage. Retrieves versions of objects affected by any case in “log01” whose life span overlapped with a certain period of time. The date format is specified.

```

1  def P1 = createPeriod("2014/11/27_15:56", "2014/11/27_16:30", "yyyy/MM/dd_HH:mm
2      ↪ ")
3
4  versionsOf (
5    casesOf(allLogs().where{name=="log01"})
6    .where {
7      overlaps(globalPeriodOf(it), P1)
8    }
9  )

```

**Dependency Relations** An important feature of the language is the ability to query existing relations between elements of different types, as well as within object versions of different classes. Query 15 showed an example of relations between elements of different types (logs to cases, cases to versions). The following query shows an example of a query on object versions related to other object versions. First, two different classes of data objects are obtained (lines 1 and 2). Then, the versions of the class “TICKET” are retrieved (line 3). Finally, the object versions related to object versions belonging to class “BOOKING” are obtained (lines 5 and 6), and only the ones belonging to class “TICKET” are selected (line 7).

**Query 16** Dependency relations. Retrieve versions of ticket objects that are related to versions of booking objects.

```

1  def ticketClass = allClasses().where{ name == "TICKET" }
2  def bookingClass = allClasses().where{ name == "BOOKING" }
3  def ticketVersions = versionsOf(ticketClass)
4
5  versionsRelatedTo (
6    versionsOf(bookingClass)
7  ).intersection(ticketVersions)

```

**Performance Metrics** As has been previously discussed, measuring performance and obtaining metrics for specific cases or activities are very common and relevant questions for many professional roles. DAPOQ-Lang supports this aspect by computing periods and durations to measure performance. The resulting periods can be used to compute performance statistics such as average execution time or

maximum waiting time. The following query shows how to compute periods for a subset of the events in the dataset.

**Query 17** Performance metrics. Retrieve periods of events belonging to activities that contain the words “UPDATE” and “CONCERT” in their name.

```

1 def actUpdateConcert = allActivities().where {
2   name.contains("UPDATE") && name.contains("CONCERT")
3 }
4
5 periodsOf(eventsOf(actUpdateConcert))

```

Query 18 demonstrates how to filter out periods based on their duration. Cases with events executed by a certain resource are selected and their periods are computed. Next, only periods with a duration longer than 11 min are returned.

**Query 18** Performance metrics. Retrieve periods of a duration longer than 11 minutes computed on cases which had at least one event executed by the resource “SAMPLEDB”.

```

1 def c = casesOf(allEvents().where { resource == "SAMPLEDB" })
2
3 periodsOf(c).where { it.getDuration() > Duration.ofMinutes(11) }

```

## 5.5 DAPOQ-Lang vs. SQL

So far, we have seen several examples of “toy” queries to demonstrate the use of the functions and operators provided by DAPOQ-Lang. Obviously, any DAPOQ-Lang query can be computed with other Turing-complete languages. When it comes to data querying on databases, SQL is the undisputed reference. It is the common language to interact with most of the relational database implementations available today. It is a widespread language, known by many professionals from different fields. Even without considering scripting languages like PL/SQL and just with CTEs (Common Table Expressions) and Windowing, SQL has been proven to be Turing-complete [2]. Therefore, the aim of DAPOQ-Lang is not to enable new types of computations, but to ease the task of writing queries in the specific domain of process mining.

Let us consider again the generic question (GQ) presented in Sect. 1:

**GQ:** In which cases, there was (a) an event that happened between time T1 and T2, (b) that performed a modification in a version of class C, (c) in which the value of field F changed from X to Y?

This question involves several types of elements: cases, events, object versions, and attributes. We instantiate this query with some specific values for T1 = “1986/09/17 00:00”, T2 = “2016/11/30 19:44”, C = “CUSTOMER”, F = “ADDRESS”, X = “Fifth Avenue”, and Y = “Sunset Boulevard”. Assuming that our database already complies with the structure proposed by the OpenSLEX meta model, we can write the following SQL query to answer the question:

**Query 19** Standard SQL query executed on the OpenSLEX dataset in [4] and equivalent to the DAPOQ-Lang Query 1

```

1  SELECT distinct C.id AS "id", CAT.name, CATV.value, CATV.type
2  FROM
3  "case" AS C
4  JOIN activity_instance_to_case AS AITC ON AITC.case_id = C.id
5  JOIN activity_instance AS AI ON AI.id = AITC.activity_instance_id
6  JOIN event AS E ON E.activity_instance_id = AI.id
7  JOIN event_to_object_version AS ETOV ON ETOV.event_id = E.id
8  JOIN object_version AS OV ON ETOV.object_version_id = OV.id
9  JOIN object AS O ON OV.object_id = O.id
10 JOIN class AS CL ON O.class_id = CL.id AND CL.name = "CUSTOMER"
11 JOIN attribute_name AS AT ON AT.name = "ADDRESS"
12 JOIN attribute_value AS AV ON AV.attribute_name_id = AT.id AND
13     AV.object_version_id = OV.id
14 LEFT JOIN case_attribute_value AS CATV ON CATV.case_id = C.id
15 LEFT JOIN case_attribute_name AS CAT ON CAT.id = CATV.case_attribute_name_id
16 WHERE
17     E.timestamp > "527292000000" AND
18     E.timestamp < "1480531444303" AND
19     AV.value LIKE "Sunset_Boulevard" AND
20     EXISTS
21     (
22     SELECT OVP.id
23     FROM
24     object_version AS OVP,
25     attribute_value AS AVP
26     WHERE
27     AVP.attribute_name_id = AT.id AND
28     AVP.object_version_id = OVP.id AND
29     OVP.object_id = OV.object_id AND
30     AVP.value LIKE "Fifth_Avenue" AND
31     OVP.id IN
32     (
33     SELECT OVPP.id
34     FROM object_version AS OVPP
35     WHERE
36     OVPP.end_timestamp <= OV.start_timestamp AND
37     OVPP.end_timestamp >= 0 AND
38     OVPP.object_id = OV.object_id AND
39     OVPP.id != OV.id
40     ORDER BY OVPP.end_timestamp DESC LIMIT 1
41     )
42     )

```

The logic is the following. Two subqueries are nested in order to retrieve (a) object versions preceding another object version (lines 33–40) and object versions that contain the attribute that changed (lines 22–41). Parts of the query focus on checking the value of the attributes (lines 27–30), the timestamp of the events (lines 17–18), and the class of the object versions (line 10). The rest of the query is concerned with joining rows of different tables by means of foreign keys.

The equivalent DAPOQ-Lang query, previously presented in Query 1, removes most of the clutter and boilerplate code in order to join tables together and lets the user focus on the definition of the constraints. The query is built up with an assignment and several nested queries. First, a period of time is defined (line 1). Then, object versions of a certain class are retrieved (lines 5–6) and filtered based on the changes of one of the attributes (line 7). Next, the events related to such object versions are obtained (lines 4–8) and filtered based on the time when they occurred (lines 8–12). Finally, the cases of these events are returned (lines 3–13).

**Table 4** Event log obtained from the execution of Query 1

#	Case	Activity name	Timestamp	Class	Address
1	1	Insert Customer	2014-11-27 15:57:13	CUSTOMER	Fifth Avenue
2	1	Update Customer	2014-11-27 16:05:01	CUSTOMER	Sunset Boulevard
3	2	Insert Customer	2014-11-27 15:58:14	CUSTOMER	Fifth Avenue
4	2	Update Customer	2014-11-27 16:05:37	CUSTOMER	Sunset Boulevard
5	3	Insert Customer	2014-11-27 15:59:16	CUSTOMER	Fifth Avenue
6	3	Update Customer	2014-11-27 16:05:54	CUSTOMER	Sunset Boulevard
7	4	Insert Customer	2014-11-27 16:01:03	CUSTOMER	Fifth Avenue
8	4	Update Customer	2014-11-27 16:07:02	CUSTOMER	Sunset Boulevard

Table 4 shows the event log obtained from the execution of this query, where we can observe that insertions of new customers are followed by updates that modify the address attribute.

In essence, the advantage of DAPOQ-Lang over SQL is on the ease of use in the domain of process mining. The fact that we can assume how logs, cases, events, and objects are linked allows us to focus on the important parts of the query. Also, providing functions that implement the most frequent operations on data (such as period and duration computation) makes writing queries faster and less prone to errors.

## 6 DAPOQ-Lang and the Process Querying Framework

The Process Querying Framework (PQF) provides a comprehensive overview of the aspects involved in the process querying cycle. This framework partly originates from a collection of functional and non-functional requirements for process querying in the process management field. The requirements, based on CRUD operations (Create, Read, Update, and Delete), focus on the relevant BPM use cases presented in [11]. As has been shown in Sect. 5, our query language fulfills or supports, to some extent, the tasks involved in the requirements regarding “Check conformance using event data” and “Analyze performance using event data.” In this section, we instantiate DAPOQ-Lang in the PQF.

The first part of PQF is named “**Model, Simulate, Record, and Correlate.**” DAPOQ-Lang does not aim to support any of the aspects covered by this part of the framework. However, the OpenSLEX meta model, and more specifically its implementation, enables the *recording* and *correlation* of behavioral and historical data in a structured way, i.e., create and update operations. This feature enables the construction of a dataset ready to be queried (i.e., read operations) by DAPOQ-Lang’s query engine. Therefore, DAPOQ-Lang addresses *event log* and *correlated data* querying with the intent of retrieving information previously recorded in an OpenSLEX-compliant storage.

With respect to the “**Prepare**” part of the PQF, DAPOQ-Lang’s support is twofold: (1) It proposes the OpenSLEX meta model to structure behavioral and object data in a format that enables *indexing* and makes the querying process easier to carry out from the usability point of view. Also, it speeds up query execution providing the most frequently requested information in a preprocessed format. (2) The underlying OpenSLEX implementation makes use of *caching* to speed up response time and make efficient use of memory. Two strategies are supported: (a) in-memory caching, which benefits speed but suffers when dealing with large datasets, and (b) disk-based caching, which makes it possible to handle larger datasets that would not fit in memory but introduces an overhead due to the serialization and disk-writing steps.

As DAPOQ-Lang is a query language with an existing implementation, it covers the “**Execute**” part of the PQF. The nested nature of the expressions in DAPOQ-Lang enables the *filtering* of the data, executing parts of the query only on the relevant elements. The implementation includes several *optimizations*, like pre-fetching of attributes as a way to save time in the filtering step of the query execution. The execution of a query in DAPOQ-Lang yields results in the form of a set of elements, obtained from the original dataset, representing the subset of the original data that satisfies the expressed constraints.

Finally, the “**Interpret**” part of the framework is supported by DAPOQ-Lang in two ways: (1) enabling the *inspection* of data using explorative queries and (2) exporting the result of queries to XLog, which makes it possible to apply any existing process mining technique that requires an event log as input, while benefiting from the capabilities of DAPOQ-Lang to build relevant sublogs for the query at hand.

## 7 Conclusion

In the field of process mining, the need for better querying mechanisms has been identified. This work proposes a method to combine both process and data perspectives in the scope of process querying, helping with the task of obtaining insights about processes. To do so, DAPOQ-Lang, a Data-Aware Process Oriented Query Language, has been developed, which allows the analyst to select relevant parts of the data in a simple way to, among other things, generate specialized event logs to answer meaningful business questions. We have formally described the syntax and semantics of the language. We presented its application by means of simple use cases and query examples in order to show its usefulness and simplicity. In addition, we provide an efficient implementation that enables not only the execution but also the fast development of queries. This work shows that it is feasible to develop a query language that satisfies the needs of process analysts, while balancing these with demands for simplicity and ease of use. Finally, we positioned DAPOQ-Lang within the Process Querying Framework [10]. DAPOQ-Lang presents certain limitations in terms of performance, expressiveness, and ease

of use. As future work, efforts will be made on (a) expanding the language with new functionalities and constructs relevant in the process mining context, (b) improving the query planning and execution steps in order to achieve better performance, and (c) carrying out empirical evaluations with users in order to objectively assess the suitability of the language within the process mining domain.

## References

1. Allen, J.F.: Maintaining knowledge about temporal intervals. *Commun. ACM* **26**(11), 832–843 (1983). <https://doi.org/10.1145/182.358434>
2. Gierth, A., Fetter, D.: Cyclic tag system. In: PostgreSQL wiki (2011). <https://www.webcitation.org/6Db5tYVpi>
3. González López de Murillas, E., Reijers, H.A., van der Aalst, W.M.P.: Everything you always wanted to know about your process, but did not know how to ask. In: Dumas, M., Fantinato, M. (eds.) *Business Process Management Workshops*, pp. 296–309. Springer International Publishing, Cham (2017)
4. González López de Murillas, E., Reijers, H.A., van der Aalst, W.M.P.: Connecting databases with process mining: a meta model and toolset. *Softw. Syst. Model.* (2018). <https://doi.org/10.1007/s10270-018-0664-7>
5. IEEE Standard for eXtensible Event Stream (XES) for Achieving Interoperability in Event Logs and Event Streams (2016). <https://doi.org/10.1109/IEEESTD.2016.7740858>
6. Lederer Antonucci, Y., Goeke, R.J.: Identification of appropriate responsibilities and positions for business process management success: Seeking a valid and reliable framework. *Bus. Process Manag. J.* **17**(1), 127–146 (2011)
7. Mans, R.S., van der Aalst, W.M., Vanwersch, R.J., Moleman, A.J.: Process mining in healthcare: Data challenges when answering frequently posed questions. In: *Process Support and Knowledge Representation in Health Care*, pp. 140–153. Springer (2013)
8. Meyer, B.: *Introduction to the Theory of Programming Languages*. Prentice-Hall, Upper Saddle River, NJ, USA (1990)
9. Müller, O., Schmiedel, T., Gorbacheva, E., vom Brocke, J.: Towards a typology of business process management professionals: identifying patterns of competences through latent semantic analysis. *Enterprise IS* **10**(1), 50–80 (2016). <https://doi.org/10.1080/17517575.2014.923514>
10. Polyvyanyy, A., Ouyang, C., Barros, A., van der Aalst, W.M.: Process querying: Enabling business intelligence through query-based process analytics. *Decis. Support Syst.* **100**, 41–56 (2017). <https://doi.org/10.1016/j.dss.2017.04.011>. <http://www.sciencedirect.com/science/article/pii/S0167923617300787>. *Smart Business Process Management*
11. Van Der Aalst, W.M.: *Business process management: a comprehensive survey*. ISRN Softw. Eng. **2013** (2013)
12. Watson, H.J., Wixom, B.H.: The current state of business intelligence. *Computer* **40**(9), 96–99 (2007). <https://doi.org/10.1109/MC.2007.331>

# Process Instance Query Language and the Process Querying Framework



Jose Miguel Pérez Álvarez, Antonio Cancela Díaz, Luisa Parody,  
Antonia M. Reina Quintero, and María Teresa Gómez-López

**Abstract** The use of Business Process Management Systems (BPMSs) allows companies to manage the data that flows through process models (business instances) and to monitor all the information and actions concerning a process execution. In general, the retrieval of this information is used not only to measure whether the process works as expected but also to enable assistance in future process improvements by means of a postmortem analysis. This chapter shows how the measures extracted from the process instances can be employed to adapt business process executions according to other instances or other processes, thereby facilitating the adjustment of the process behavior at run-time to the organization needs. A language, named Process Instance Query Language (PIQL), is introduced. This language allows business users to query the process instance measures at run-time. These measures may be used both inside and outside the business processes. As a consequence, PIQL might be used in various scenarios, such as in the enrichment of the information used in Decision Model and Notation tables, in the determination of the most suitable business process to execute at run-time, and in the query of the instance measures from a dashboard. Finally, an example is introduced to demonstrate PIQL.

---

J. M. Pérez Álvarez (✉)  
NAVER LABS Europe, Meylan, France  
e-mail: [jm.perez@naverlabs.com](mailto:jm.perez@naverlabs.com)

A. C. Díaz · A. M. R. Quintero · M. T. Gómez-López  
Dto. Lenguajes y Sistemas Informáticos, Universidad de Sevilla, Sevilla, Spain  
e-mail: [acancela@us.es](mailto:acancela@us.es); [reinaqu@us.es](mailto:reinaqu@us.es); [maytegomez@us.es](mailto:maytegomez@us.es)

L. Parody  
Dto. Métodos Cuantitativos, Universidad Loyola Andalucía, Dos Hermanas, Spain  
e-mail: [mlparody@uloyola.es](mailto:mlparody@uloyola.es)

## 1 Introduction

The management of the information regarding the current and past status of an organization is crucial. The analysis of this information can be essential for different purposes, such as to determine whether the objectives of the organization are achieved, to ascertain whether the company evolves as was planned, to verify whether decision-making processes must modify company's evolution [5], and to study how the management of the company can be improved. In this respect, it could be stated that the sooner the state of the company is ascertained, the sooner the decisions to "redirect" the company could be made. The status of an organization is reflected in the information systems used to support its operations and, frequently, by the BPMSs that are often used in business process-oriented organizations.

A BPMS enables the company's activities to be assisted by means of automating and monitoring the technical environment to achieve the defined objectives. In this context, the status of the organization can be extracted from the data generated during the execution of its activities, both stored in external repositories or by the BPMSs [4]. Although the extraction of knowledge from external data repositories is usually performed through SQL as the standard language that queries relational data, the extraction of the status of the organization from a BPMS according to the process executions has no standard form. However, BPMSs also help experts to extract information about processes: how many instances of a process have been executed, how many instances of a process are currently being executed, how the process was executed (when it was started, finished, and/or canceled), and how the activities of a process instance have been executed (who executed it, when it was executed, the specific values of the data consumed and generated, or whether the activity was canceled). In other words, BPMSs allow measuring how a specific process works.

The state of executions of the processes can be inferred from measures of the process instances, and consequently, these measures enable the improvement of processes themselves. Note, however, that these measurements enable not only the process execution improvement or a process model redesign but also the adaptation of the current process instances according to the rest of the instances and available resources at any given moment. These two different usages of measures are aligned with the two types of monitoring at run-time: active and passive monitoring [6]. On the one hand, active monitoring implies the automatic evaluation of measurements since the system includes the adequate tools to detect deviations and notify them at run-time. On the other hand, passive monitoring does not evaluate metrics automatically, but it is the user who proactively requests their evaluation. The monitoring of processes allows companies to ascertain whether certain performance indicators, such as Key Performance Indicators (KPIs) and Process Performance Indicators (PPIs), support the achievement of companies' goals. KPIs represent the companies' business goals and are described in their strategic plans [7], whereas PPIs are related to the processes instantiated in the companies to achieve their goals. The incorporation of measures regarding the process instances into the business



process execution can be crucial since it allows the adaptation of the process execution according to the performance of other business process instances at any given moment.

To measure the degree of efficiency of the process output and also the level of achievement of the company goals supported by KPIs and PPIs, the involved measures must be queried. This chapter introduces Process Instance Query Language (PIQL) as a mechanism to query these measures regarding the business process status, thereby making it possible to adapt the current execution of the instances based on the obtained measures. This language has been designed to be easily understandable for nontechnical people, since the managers may not be familiar with complex query languages. PIQL is close to the natural language and brings flexibility and agility to organizations because it empowers the managers to adapt many aspects of the organization by themselves.

The measures obtained by means of PIQL queries are useful in many scenarios, e.g., internally, within the process instance under execution to make decisions based on its status, or externally, in third-party applications that show the measured values in a dashboard. For example, PIQL allows business experts to extract measures to decide the assignment of a task to a person, depending on the number of activities executed by him/her in the past, decide when a process must be started or stopped, or determine whether a process evolves as expected. We have foreseen three scenarios where the use of PIQL can be useful: decision-making using Decision Model and Notation (DMN) [9], monitoring the evolution of the processes by means of external dashboards, and management of business processes to decide the most suitable process to execute and when it should be executed.

In order to illustrate a case where PIQL is used, an example is employed that consists of a set of business processes related to component assembly in industry. The example highlights the need to incorporate information about the instances of other processes to adapt the execution of the current process according to the needs and available resources at various moments.

The chapter is organized as follows: Firstly, Sect. 2 defines the main concepts needed to understand the rest of the chapter. Secondly, Sect. 3 explains a scenario to show the applicability of PIQL. Thirdly, Sect. 4 describes PIQL. Section 5 then details the implementation of PIQL. Section 6 applies PIQL to the motivating scenario, and Sect. 7 relates PIQL to the Process Querying Framework (PQF) introduced in [11]. Finally, conclusions are drawn and future work is proposed in Sect. 8.

## 2 Background

Processes of companies can be described using *business process models*. These models represent the activities of a company, and they can be automated by means of a Business Process Management System (BPMS). To lay the foundations of the

basic business process terminology used in the chapter, we adopt the following definitions provided by Weske in [12]:

**Definition 2.1** A *business process* consists of a set of activities that are performed in coordination in an organizational and technical environment.

**Definition 2.2** A Business Process Management System (BPMS) is a generic software system that is driven by explicit process representations to coordinate the enactment of business processes.

**Definition 2.3** A *business process model* is an abstract representation of a business process that consists of a set of activity models and execution constraints between them.

**Definition 2.4** A *business process instance* represents a specific case of an operational business of a company and consists of a sequence of activity instances.

Activities in business processes can be manual activities (that is, not supported by information systems), user interaction activities (performed by workers using information systems), or system activities (which are executed by information systems without any human involvement). Activities can also be classified into atomic, those that cannot be decomposed, and non-atomic. An activity instance is a specific case of an activity model. In other words, business process models act as blueprints for business process instances, while activity models act as blueprints for activity instances.

**Definition 2.5** A *task* is an activity that cannot be decomposed.

BPMSs help companies to monitor the execution of business processes by means of performance measures or indicators. These performance measures are usually graphically represented in dashboards, which are software tools that help experts to analyze data, detect business problems, and make decisions. KPIs and PPIs are two kinds of performance indicators. KPIs are employed to describe what the company wants to achieve (e.g., increase in the number of assembled components by 10%). When the indicators are related to the measurements of the processes of the organization, PPIs are used (e.g., reduce by 15% the time of the assembly process). Several of these by measures, above all those related to PPIs, are usually stored by BPMSs and can be obtained by observing processes and analyzing these observations [1, 2]. The aforementioned notions can be defined as follows:

**Definition 2.6** A *dashboard* is a tool commonly used in business to visually represent the indicators that are related to business goal achievements.

**Definition 2.7** A *Key Performance Indicator (KPI)* is an indicator that measures the performance of key activities and initiatives that lead to the success of business goals. A KPI often involves financial and customer metrics to describe what the company wants. Achievement of KPIs indicates whether the company is attaining its strategic goals or not.

**Definition 2.8** A *Process Performance Indicator (PPI)* is an indicator that involves measures of the performance of the instances of the executed business processes. PPIs are also related to the goals of the company but involve the measurement of processes used to achieve them.

In order to support business process execution, every BPMS includes the following components, as shown in the classical architecture published in [3]: an execution engine, a process modeling tool, a worklist handler, and an administration and monitoring tools. The *external services* represent external application or services that are involved in the execution of a business process. The *execution engine* is responsible for enacting executable process instances, distributing work to process participants, and retrieving and storing data required for the execution of processes. The *process modeling tool* is the component that allows users to create and modify process models, annotate them with data elements, and store these models in or retrieve them from a *process model repository*. The *worklist handler* is in charge of offering work items to process participants and committing the participants to work lists. Finally, the *administration and monitoring tools* administer and monitor all the operational matters of a BPMS. These components retrieve and store data from two repositories: the process model repository and the execution log repository. The former stores process models, while the latter stores events related to process execution. Note that in the classical architecture, there is no connection between the data of the execution logs and the modeling module. In our approach, the *PIQL engine* is the component in charge of connecting the *administration and monitoring tools* and the *process modeling tools*. As a consequence, the modeler can execute queries to evaluate the status of business process executions. In addition, the *PIQL engine* is connected to *execution engine*, since the measures can be used internally in the execution of other processes, and it is also connected to *external services* since its measures can be used for different purposes, such as to monitor the status of the BPMS in an external dashboard. Figure 1 shows the classical architecture with the PIQL engine included.

**Definition 2.9** *Process Instance Query Language (PIQL)* is a domain-specific language (DSL) to query process and task instances in order to obtain measures based on historical process executions.

Process modeling tools can use various notations to represent process models. The de facto standard notation used for modeling business processes is Business Process Model and Notation (BPMN), which supports business process modeling by providing a notation that, on the one hand, is comprehensible to business users, and, on the other hand, represents complex process semantics for technical users [8].

BPMN is not well-suited for modeling the decision logic since decisions are often intermingled with the control flow of process models. The Object Management Group (OMG) proposed DMN to decouple decision specifications from process models. The goal of DMN is to provide a common notation that is readily understandable by all business users in order to bridge the gap between the business decision design and its implementation [9]. In other words, DMN can be seen as a

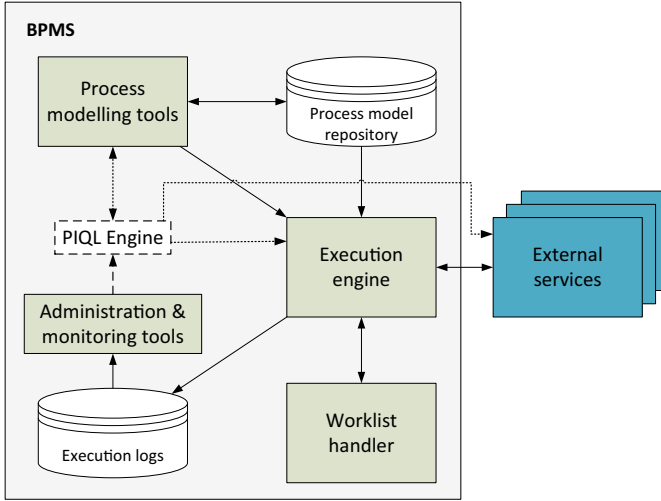


Fig. 1 Architecture of a BPMS that includes a PIQL engine

tool that allows business users to describe repeatable decisions within organizations. DMN provides two levels of modeling: the decision requirement level and the logic level. The former is modeled by means of a Decision Requirement Diagram (DRD), which shows how the decision is structured and what data is needed to make the decision. The latter is modeled using decision tables, which is the standard way of modeling complex business rules.

**Definition 2.10** A *decision table* is a visual representation of a specification of actions to be performed depending on certain conditions. A decision table consists of a set of rules that specifies causes (business rule conditions) and effects (business rule actions and expected results); it specifies which inputs lead to which outputs.

A decision table is represented by means of a table, where one column represents one condition. For example, Table 1 shows a decision table that specifies which assembly station should be used depending on the kind of available pieces and the availability of the stations.

In Table 1, decision #1 states that if there are two or more #1657 pieces, five or more #6472 pieces, and at least one #2471 piece, and only Station 6 is available, then the component should be assembled in Station 6. Another example is Decision #3 that results in the assembly of the component in Station 15. The letter “F” marked with an “\*” means “First” [9], and the decision engine will evaluate the rules in the proposed order and stop once it has found a rule that applies.

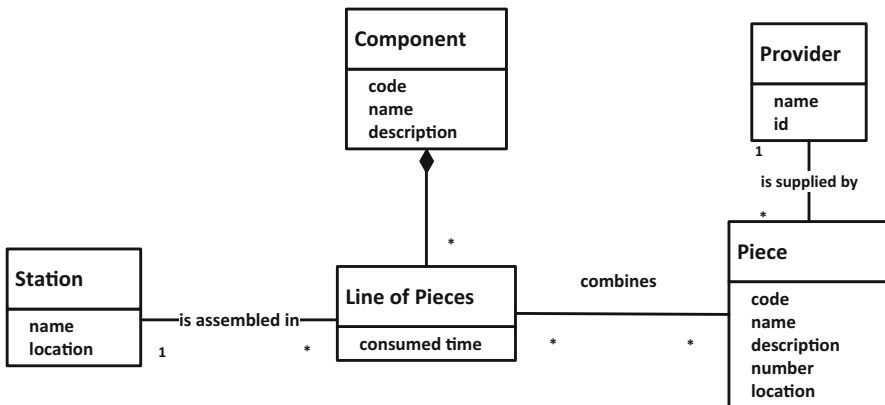
**Table 1** Example DMN table for selecting a task based on the availability of resources

F*	Input						Output
#ID	Nº of pieces code #1657 (int)	Nº. of pieces code #6472 (int)	Nº of pieces code #2471 (int)	Is Station 6 available? (Boolean)	Is Station 12 available? (Boolean)	Is Station 15 available? (Boolean)	Station (string)
#1	>= 2	>= 5	>= 1	True	False	False	Station 6
#2	>= 1	>= 2	>= 6	False	True	False	Station 12
#3	>= 7	Any	>= 1	False	False	True	Station 15
...	...	...	...	...	...	...	...

### 3 Motivating Scenario

In order to show the applicability of PIQL in a real-world context, we introduce a motivating scenario related to the assembly process in a factory, similar to the ones used in the automobile and aerospace industries. A factory produces a set of *Components* in a production line. The production line is composed of a set of actions defined as the *Line of Pieces*. Each *Line of Piece* is executed in a factory *Station* and combines a set of *Pieces*. Figure 2 depicts a conceptual model of this scenario using the Unified Modeling Language (UML) [10]. The conceptual model includes the elements that are the most relevant elements to the assembly process.

The decision regarding which *Components* to assemble depends on the customers' requirements and the availability of the *Pieces* and *Stations* at the time. A specific *Piece* can be used in various *Components*. As a consequence, when a *Piece* is available, different assembly processes could be set up at that moment. Moreover, the same *Station* can be used in different assembly processes, and therefore the manager has to control the availability of the *Station* in each case. The management of *Stations* and *Pieces* involves a set of crucial and risky decisions for the company.



**Fig. 2** Conceptual model of assembly process in UML notation [10]

Considering that there is a good amount of stock and that there are always *Pieces* available, then the main problem is in making decisions about the *Stations*. These decisions depend on all the running process instances, that is, *Components* that are being assembled in parallel, *Components* that are waiting for the assembly to start, or *Components* whose assembly is finishing.

Figure 3 shows two business processes related to the assembly of two different types of components. These processes model the sequence of stations visited by a component for its assembly. Although in both processes the components have to pass through a number of stations, the order and requirements vary. On the one hand, a “*Type A Component*” assembly must always pass through the three stations 6, 12, and 15, although the order does not matter (hence the process model does not require to visit all the three stations). On the other hand, a “*Type B Component*” assembly varies according to the requirements of the component at any given time; it may visit both stations, 12 and 15, or only visit one of the two. Finally, validation of “*Type A Component*” is automated by checking if the component has visited all stations, whereas validation of “*Type B Component*” requires expert supervision. The decisions that route the execution from one station to the other can be seen in Table 1.

Querying data related to the status of the company, such as the number of instances executed or availability of stations, is crucial in different contexts. The following subsections introduce some of the contexts in which these queries are essential.

### Context 1: Dashboards

Business experts monitor and manage the evolution of the company’s business processes, commonly by means of a dashboard. A dashboard visualizes several indicators to help experts carry out correct management. For instance, the “increase of the number of Type A Components by 24%” is an example of KPI defined for measuring the goal “increase market share.” Meanwhile, “the number of instances successfully executed” and “the instantiation time of the ‘Assembly of Type B Component’ process” are examples of PPI. In order to obtain these indicators, various measures should be defined: the number of executions of the “*Assembly of Type A Component*” process indicates whether the KPI “increase of the number of Type A Components by 24%” is reached, whereas the number of executions of the “*Assembly of Type B Component*” process that have not been canceled solves the PPI detailed above. To obtain the execution time of an instance, its start and end times must be analyzed. In all these cases, the use of PIQL in obtaining these measures is essential.

### Context 2: DMN Tables

Another context in which PIQL is crucial is at decision points. In Fig. 3, both diagrams contain decision tasks, such as the “*Decide the tasks according to the availability*” task in the “*Assembly of Type A Components*” process. This task takes into account the availability of Stations 6, 12, and 15 to decide which station should be used to continue with the assembly of the component. The decision logic related to this task can be modeled with a DMN table. In fact, Table 1 shows an example logic behind the decision of the “*Decide the tasks according to the availability*”

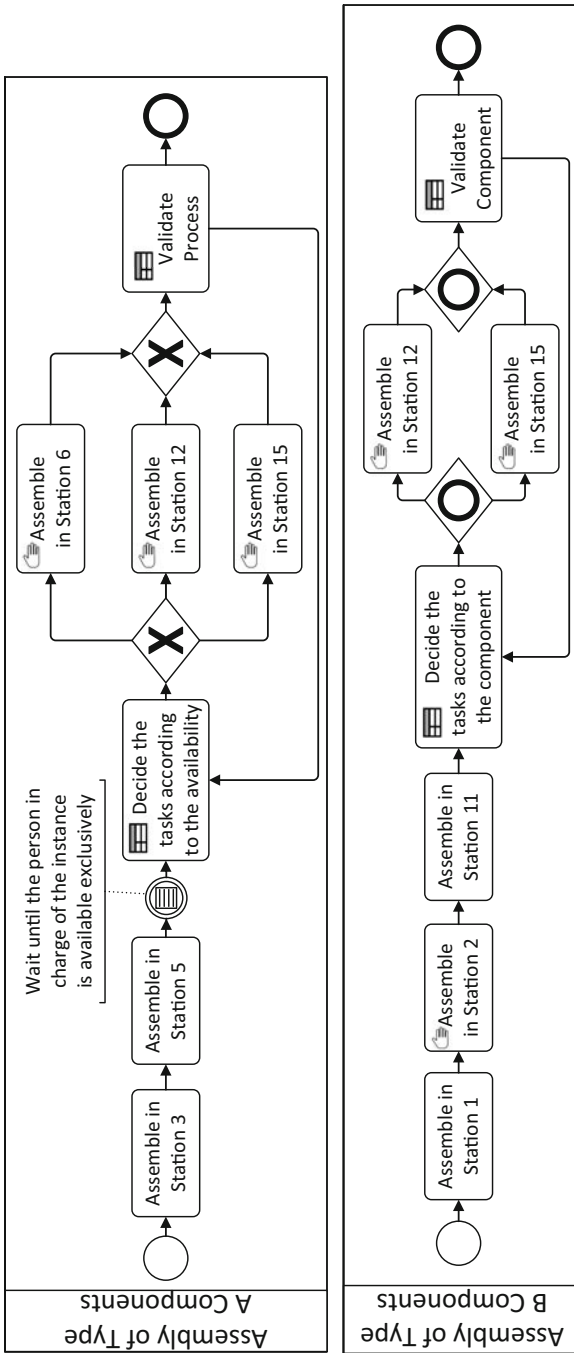


Fig. 3 Business process models for the assembly of two components captured in BPMN [8]

task. Depending on the number of pieces, whose codes are #1657, #6472, and #2471, and on the availability of the stations, the decision varies. The availability of Stations 6, 12, and 15 can be checked by executing a PIQL query since it is necessary to ascertain whether other process instances are using these stations.

### Context 3: Dataflow in Business Process Management

Finally, another context where PIQL becomes decisive is in the kind of information that flows through the process, that is, the dataflow. At certain points, the information related to other instances or resources is vital and should be included as part of the dataflow. Following on with the previous example, after the execution of the “*Assemble in Station 5*” task in the “*Assembly of Type A Components*” process, there is a conditional event. The event needs to ascertain whether the person in charge of the process instance is executing some task. If the person is performing another task, then the process should wait until this person is released. This information can be obtained through the execution of a PIQL query.

## 4 Process Instance Query Language

This section explains the main components of PIQL: syntax, semantics, and notation. Furthermore, as the envisioned users of PIQL are nontechnical people, a set of *Patterns* and *Predicates* is defined in this section to help them write queries in a language that resembles the natural language.

A Process Instance Query Expression (PIQE) (an expression) is used to represent a PIQL query, which is evaluated within the context of processes (P) or tasks (T). The context specifies whether the query recovers information about process instances or about task instances. Note that the result of a PIQE execution is always a measure, that is, a numeric value. For example, Listing 1 shows a PIQE that retrieves information about the number of process instances (note that the expression starts with P to denote the process context). Furthermore, if we look at the PIQE shown in Listing 1, we can find some other features such as (i) every keyword or relation operator is written in uppercase letters and (ii) parentheses could be used to group expressions.

**Listing 1** Process instance query expression

```
P (ProcessName IS-EQUAL-TO "Assembly of Type A Components"
AND (Start IS-GREATER-THAN 2018-01-06)
AND (End IS-LOWER-OR-EQUAL-TO 2018-03-16));
```



## 4.1 Syntax

The syntax of PIQL is defined using the Extended Backus-Naur Form (EBNF) [13]. Thus, if  $x$  and  $y$  are symbols,  $x?$  denotes that  $x$  can appear zero times or once,  $x+$  denotes that  $x$  can appear one or more times,  $x^*$  denotes that  $x$  can appear zero or more times, and finally,  $x | y$  represents that either  $x$  or  $y$  can appear. In addition, non-terminal symbols are enclosed in  $\langle$  and  $\rangle$ , and symbols and keywords are enclosed in single quotes. For the sake of clarity, the definition of the non-terminal symbol  $\langle$ String $\rangle$  has not been included in the grammar shown below, but it should be taken into account that a String is a sequence of characters that starts and ends with double quotes. For instance, "This is a String" is a String.

```

<PIQE> ::= <Context> <Disjunction> ';'
<Context> ::= 'P' | 'T'
<Disjunction> ::= <Conjunction> ('OR' <Conjunction>)*
<Conjunction> ::= <Negation> ('AND' <Negation>)*
<Negation> ::= 'NOT'? <Comparison>
<Comparison> ::= <Addition> (<ComparisonOperator> <Addition>)*
<ComparisonOperator> ::= 'IS-EQUAL-TO' | 'IS-NOT-EQUAL-TO'
    | 'IS-LOWER-THAN' | 'IS-GREATER-THAN'
    | 'IS-LOWER-OR-EQUAL-TO'
    | 'IS-GREATER-OR-EQUAL-TO'
<Addition> ::= <ArithmeticOperand> (('PLUS' | 'MINUS') <ArithmeticOperand>)*
<ArithmeticOperand> ::= '(' '?' <Operand> ')' (('MULTIPLIED-BY'
    | 'DIVIDED-BY') '(' '?' <Operand> ')' '?' ')' '*'
<Operand> ::= '(' <Disjunction> ')' | <Property> | <Value> | <Variable>
<Property> ::= 'ProcessName' | 'TaskName' | 'Start' | 'End' | 'Canceled'
    | 'Who'
<Value> ::= <Number> | <String> | <Date> | <Boolean> | 'NULL'
<Date> ::= <Integer> '/' <Integer> '/' <Integer>
    | <Integer> '-' <Integer> '-' <Integer>
<Boolean> ::= 'true' | 'false'
<Number> ::= <Integer> | <Float>
<Float> ::= <Integer>? '.' <Digits>
<Integer> ::= '-'? <Digits>

```

$\langle \text{Digits} \rangle ::= ' (' '0' .. '9' ') '+$

$\langle \text{Variable} \rangle ::= '\$' \langle \text{String} \rangle$

## 4.2 Semantics

The main concepts related to PIQL are detailed below.

**Expression.** An expression is a combination of one or more values, variables, and operators. Each expression can be seen as one single query, and it should be evaluated within a context: either the process or the task context.

**Context.** A context specifies whether one wants to retrieve information about *process instances* (P) or *task instances* (T). The context determines the kind of information and attributes that can be retrieved and/or used to define a query.

**Process Instance Context.** A process instance is described by a tuple

$\langle \text{CaseId}, \text{ProcessName}, \text{Start}, \text{End}, \text{Canceled}, \text{Who}, \text{ListOfGlobalData} \rangle,$

where

- *CaseId* is an identifier that describes the process instance in an unequivocal way. It is assigned by the BPMS when the instance is created.
- *ProcessName* is the name of the instantiated process model.
- *Start* is the date and time when the instance started.
- *End* is the date and time when the instance finished. If the instance has not ended, this attribute is set to *null*.
- *Canceled* is the date and time when the instance was canceled. If the instance has not been canceled, this attribute is set to *null*.
- *Who* is the person who has started the execution of the instance. If the instance has been started by the system, this attribute is set to *system*.
- *ListOfGlobalData* is a collection of the process model global variables. The instantiation of these variables can be crucial at decision points.

**Task Instance Context.** A task instance represents the information related to the execution of a specific task within a process instance. It is described by the tuple:

$\langle \text{CaseId}, \text{TaskId}, \text{TaskName}, \text{ProcessName}, \text{Start}, \text{End}, \text{Canceled}, \text{Who} \rangle,$

where the elements have the following meaning:

- *CaseId* is the identifier of the process instance of the activity being executed.
- *TaskId* is an identifier that describes the task instance in an unequivocal way. It is assigned by the BPMS when the instance is created.
- *TaskName* is the name of the task.

- *ProcessName* is the name of the process model that contains the task. Note that this property is derived from *CaseId* (by querying the process whose id is *CaseId*).
- *Start* is the date and time when the task started. If the task has not started, then this attribute is set to *null*.
- *End* is the date and time when the task finished. If the task has not finished, then this property is set to *null*.
- *Canceled* is the date and time when the execution of the task was canceled. If the task has not been canceled, then this property is set to *null*.
- *Who* is the person who has started the execution of the task instance. If the instance was started by the system, this attribute is set to *system*.

As already stated, the result of a PIQE is a measure, that is, a numeric value. PIQE supports three types of operators to filter instances: *Logical*, *Comparison*, and *Arithmetic*. Operators are modeled using uppercase letters and the hyphens symbol. The operators supported by PIQL, and grouped by types, are described below.

**Logical Operators** combine two Boolean values. The following logical operators are defined in PIQL:

- *NOT*: logical negation, it reverses the true/false outcome of the expression that immediately follows.
- *OR*: it performs the logic operation of disjunction.
- *AND*: it performs the logic operation of conjunction.

**Comparison Operators** define comparisons between two entities. These comparison operators are applied to the data types specified in the grammar: Date, Number, Float, Integer, and String. For numeric data types, natural sort order is applied; for Date type, chronological order is applied; and the comparison of String data is performed in alphabetical order. The following comparison operators are defined in PIQL:

- *IS-EQUAL-TO* evaluates whether two elements have the same value.
- *IS-NOT-EQUAL-TO* evaluates whether two elements have different values.
- *IS-GREATER-THAN* evaluates to *true* if the first element of the expression has a greater value than the second. For Dates, this means that the first date is later than the second one.
- *IS-GREATER-OR-EQUAL-TO* returns false when the second element of the expression has a higher value than the first one; otherwise, it returns true.
- *IS-LOWER-THAN* is the inverse of the previous operator. It returns true when the first element of an expression has a lower value than that of the second element; otherwise, it returns false. For Dates, this means that the first date is earlier than the second one.
- *IS-LOWER-OR-EQUAL-TO* is the inverse of *IS-GREATER-THAN* operator.

**Table 2** Precedence of PIQL operators

Precedence	Operators	Associativity
5	()	Left to right
4	MULTIPLIED-BY	Left to right
	DIVIDED-BY	
3	PLUS	Left to right
	MINUS	
2	IS-EQUAL-TO	Left to right
	IS-NOT-EQUAL-TO	
	IS-LOWER-THAN	
	IS-GREATER-THAN	
	IS-LOWER-OR-EQUAL-TO	
	IS-GREATER-OR-EQUAL-TO	
1	NOT	Left to right
	OR	
	AND	

**Arithmetic Operators** are binary operators that define mathematical operations between two entities. These operators are applied to numerical data. The following arithmetic operators are defined in PIQL:

- *PLUS*: it performs the addition of the elements surrounding this operator.
- *MINUS*: in the A MINUS B expression, the MINUS operator performs the subtraction of B from A.
- *MULTIPLIED-BY*: it performs the multiplication of the first and second elements of the expression.
- *DIVIDED-BY*: it performs the division of the first element by the second element.

Table 2 shows the precedence and associativity of the different operators defined in PIQL. The column Precedence holds numbers that specify the precedence of the operator specified in the row. The greater the number is, the higher precedence the operators have.

**Variable.** A variable can be seen as a placeholder, as it is replaced with a specific value at run-time. A variable can be used to write PIQEs in a more flexible way. The value of a variable needs to be specified by the user at run-time. However, there is also a set of system variables that do not need to be specified, which are:

- \$yesterday evaluates to the day before the current date.
- \$today or \$current\_date evaluates to the current date.
- \$this\_instant evaluates to the current timestamp.
- \$tomorrow evaluates to the day after the current date.

Two special symbols used in the PIQL grammar are composed of:

- “\$”: this operator is used to indicate a variable.
- “;”: the semicolon operator marks the end of a PIQE.

### 4.3 *Patterns and Predicates*

Since we envision that the main users of PIQL will be nontechnical people, PIQL is enriched with a set of patterns and predicates that help users to write queries in a language that resembles the natural language. The patterns and predicates could easily be defined for several languages. Thus, for example, there could be a set of patterns for English speakers, another set for Spanish speakers, and so on. In this chapter, the set of patterns for English speakers is defined, as shown in Tables 3 and 4. These patterns and predicates can be automatically translated into PIQEs. Therefore, these patterns and predicates are not only the mechanisms that make PIQL more friendly to nontechnical people but also the mechanisms that make the language more flexible, because the patterns can be easily adapted to be closer to the modeler’s mother tongue.

A pattern is a mapping between a sentence, or part of a sentence written in the expert’s mother tongue, and an element of the PIQL grammar. For example, the pattern “The number of instances of processes” is mapped to the Process Instance Context element, in the case of the English language. This means that a nontechnical user can start a query by writing “The number of instances of processes” instead of just writing “P” to specify the context in which the query must be evaluated. Table 3 shows the PIQL patterns defined for English speakers and their relation to the grammar elements.

A predicate is a pattern that represents a Boolean-valued function written in a natural language. For example, instead of writing “The number of instances of processes whose end date is not equal to Null,” the user can write “The number of instances of processes that are not finalized.” Predicates are transformed into patterns, and then these patterns are transformed into PIQEs, written according to the grammar introduced in Sect. 4.1. A set of predefined predicates and their mappings to patterns is shown in Table 4.

Finally, note that the use of patterns and predicates is not mandatory. Thus, third-party applications or technical users can also use the “*raw*” language (without patterns and predicates).

**Table 3** PIQL patterns for English language

Grammar component	Pattern	
Process instance context	The number of instances of processes	
Task instance context	The number of instances of tasks	
Properties	<b>Attributes</b>	<b>Pattern syntax</b>
	idCase	<i>With the case id</i>
	Process_Name	<i>With the name</i>
	Task_Name	<i>With the name</i>
	Start	<i>With the start date and time</i>
	End	<i>With the end date and time</i>
	Canceled	<i>Canceled</i>
	Who	<i>Executed by the user</i>
ARITHMETIC_OP	<b>Operator</b>	<b>Pattern syntax</b>
	PLUS	<i>Plus</i>
	MINUS	<i>Minus</i>
	MULTIPLIED-BY	<i>Multiplied by</i>
	DIVIDED-BY	<i>Divided by</i>
BOOLEAN_OP	<b>Operator</b>	<b>Pattern syntax</b>
	NOT	<i>Not</i>
	AND	<i>And</i>
	OR	<i>Or</i>
COMPARISON_OP	<b>Operator</b>	<b>Pattern syntax</b>
	IS-EQUAL-TO	<i>Is equal to</i>
	IS-NOT-EQUAL-TO	<i>Is not equal to</i>
	IS-LOWER-THAN	<i>Is less than</i>
	IS-GREATER-THAN	<i>Is greater than</i>
	IS-LOWER-OR-EQUAL-TO	<i>Is less than or equal to</i>
IS-GREATER-OR-EQUAL-TO	<i>Is greater than or equal to</i>	

**Table 4** PIQL predicates for English language

Predicate	Transformed pattern
That are finalized	End date is not equal to Null
That are not finalized	End date is equal to Null
That are canceled	Canceled is not equal to Null
That are not canceled	Canceled is equal to Null
That are executed by {name}	The user is equal to {name}
With start before {date and time}	A start date is less than {date and time}
With end before {date and time}	An end date is less than {date and time}
With start after {date and time}	A start date is greater than {date and time}
With end after {date and time}	An end date is greater than {date and time}

## 5 Implementation

In order to validate the approach, an implementation of PIQL has been developed using a set of mature technologies. The core element of the implementation is the PIQL engine, which is in charge of executing queries and returning the results.<sup>1</sup> Note that the PIQL engine can be connected to any system in order to integrate the results of PIQEs in the various contexts in which PIQL may be used: dashboards, DMN tables, and dataflows. Before going into the details of the PIQL engine implementation, it should be mentioned that PIQL provides a dual format of a query: a user format and a machine format, which is less verbose and better processed by machines. The two formats and their relationships are depicted in Fig. 4. Note that the user format is related to the set of patterns and predicates introduced in the previous section, which helps users write queries in a language closer to the natural language.

Figure 4 shows how the PIQL engine works. Firstly, a user writes a query using a language close to English (user format). The PIQL engine then transforms the query into a PIQE by means of a “Grammar preprocessor” (machine format). Note that a PIQE does not contain any patterns or predicates and also that the “Grammar processor” is the component that allows the interaction with third-party applications that may use the machine format. The “PIQL grammar processor” evaluates the PIQE by extracting the information from the BPMS. Finally, the “Platform communication interface” is the component that deals with different BPMS technologies by means of different drivers. Figure 4 shows how the *Camunda*<sup>TM</sup> driver queries the BPMS using a REST API.

As mentioned previously, PIQL can be used in different contexts: dashboards, DMN tables, and dataflows. Hence, the PIQL engine should be integrated in each context. As an example, Fig. 5 shows the architecture defined to include context information in DMN tables. Since the integration of the PIQL engine in the other contexts is similar to this model, the rest of the section details this example.

The BPMS chosen is that of *Camunda*<sup>TM</sup> since it is an open-source platform that includes the workflow engine, the DMN evaluator, and the storage of logs for every process, which are required to implement the proposed architecture. *Camunda*<sup>TM</sup> also includes a set of APIs, which is the mechanism employed to extract the information regarding the processes and tasks needed for the evaluation of PIQEs. The main components depicted in Fig. 5 are:

- *REST Layer*: This component is in charge of managing the communication between *Camunda*<sup>TM</sup>, the DMN evaluator, and the PIQL engine. This component is implemented using a model—view—controller framework and it feeds the DMN evaluator by means of a *REST API* that has been implemented using

---

<sup>1</sup> The readers may test the PIQL engine at <http://estigia.lsi.us.es:8099/piql-tester>.

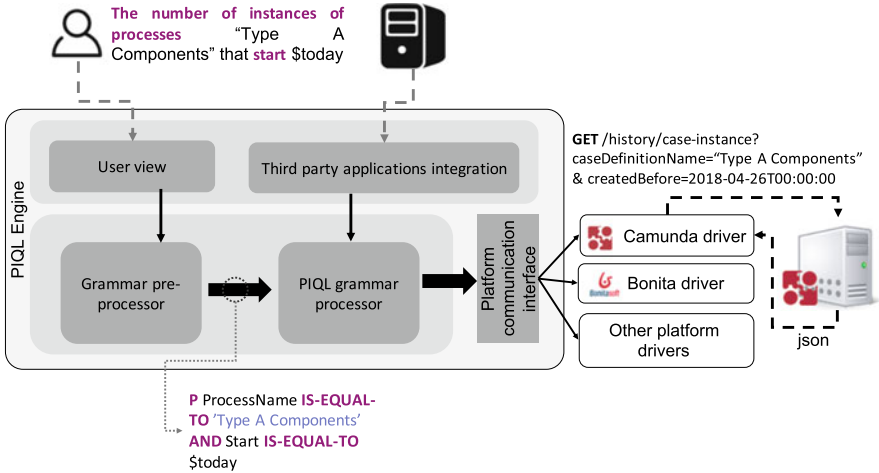


Fig. 4 Query transformation example

*Jersey*.<sup>2</sup> The data exchanged using this component is in *JSON* format.<sup>3</sup> When a DMN decision has to be evaluated, the BPMS requests the information using the *REST Layer* component. This request is then managed by the “Controller” layer.

- *Controller*: This component receives a request from the *REST Layer* and is in charge of using the *Grammar Preprocessor*, if needed, and later, the *PIQE Grammar Helpers*. Note that if the query is not written in the user-friendly notation, then the preprocessor is not needed.
- *Grammar Preprocessor*: This component handles the mapping of the user-friendly PIQL notation to PIQEs.
- *PIQE Grammar Helper*: This component, together with the *PIQL Engine*, is responsible for resolving the PIQEs. The technology employed to implement this component is *xText*, an open-source framework for the development of textual domain-specific languages.<sup>4</sup>
- *PIQL Engine*: This component analyzes the query by means of the *PIQE Grammar Helpers*. It then calculates the information needed to solve the query and extracts that information from the BPMS. Finally, it returns the value of the PIQE to the controller in order to finalize the request. Note that this component does not have direct communication with the platform (*Camunda*<sup>TM</sup> in this case) since, to decouple the engine from the platform, a *driver* is introduced. This driver acts as an abstract interface to access the real platforms.

<sup>2</sup> <https://jersey.github.io/>.

<sup>3</sup> <https://www.json.org/>.

<sup>4</sup> <https://www.eclipse.org/Xtext/>.



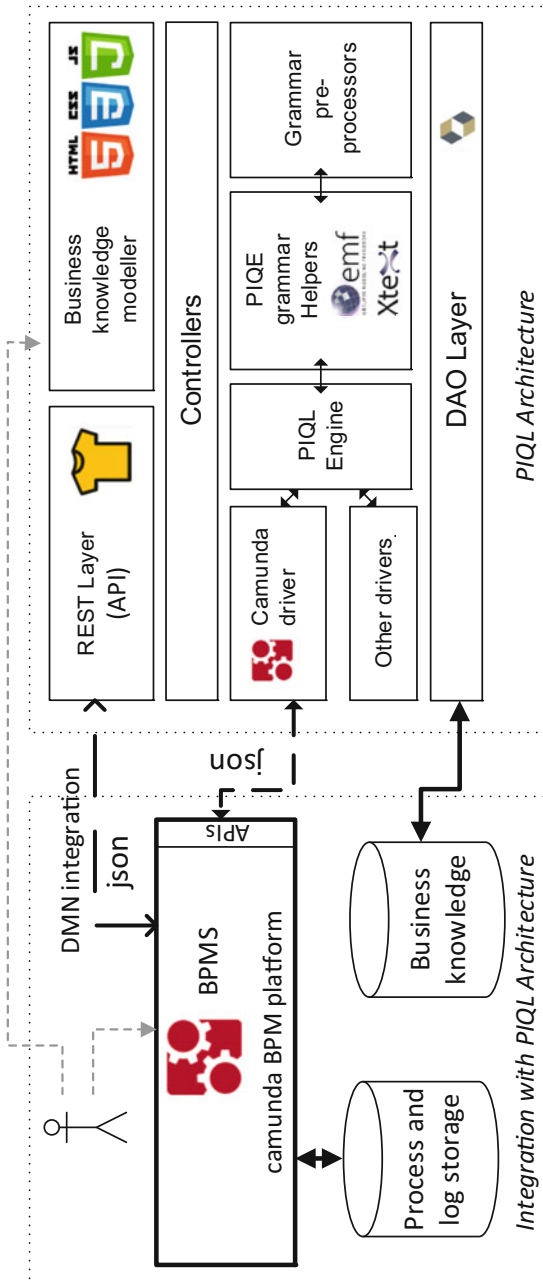


Fig. 5 PIQL architecture

- *Driver*: A driver is responsible for communicating with a real platform and this is the component that knows the specific details of that platform. Note that every system can accept different requests and return different responses. Even worse, there are ad hoc systems which do not provide any API but do provide other ways of retrieving information. This means that one PIQE has to be reformulated, and the reformulation depends completely on the system being used. For example, in some cases, the driver should carry out data processing before returning PIQL data, while in other cases a PIQE needs to be translated into several API requests.
- *Camunda Driver*: In the case of *Camunda*<sup>TM</sup> the driver uses the *Camunda history service*, which, in turn, uses *Camunda*<sup>TM</sup> REST APIs.
- *DAO Layer*: This component is responsible for storing the business knowledge. *Hibernate* is the technology employed to implement and manage the object-relational mapping.<sup>5</sup>
- *Business Knowledge Modeler*: This component allows users to handle PIQEs and to manage the DMN tables. The implementation of this component has taken advantage of the architecture revealed herein and constitutes a Web application implemented using *HTML*, *CSS*, and *AngularJS*.

## 6 Application

The following subsections show examples of queries that are used in the three contexts explained during the chapter. Note that each query is written in the two PIQL notations: the user-friendly format and the machine format.

### 6.1 Dashboard Enriched with PIQL

A dashboard is composed of a set of measures that enables business experts to easily visualize the state of the company by means of different KPIs and PPIs. An example of KPI in the context of the motivating scenario introduced in Sect. 3 could be “increase in the number of Type A Components by 24%.” The following PIQL queries should be executed to obtain the measures that allow the user to verify whether the KPI is reached:

- **Query 1**: The number of process instances with the name “*Assembly of Type A Components*” that start after *2017-12-31* and end before *2019-01-01*.

```
/* PIQE using user-friendly notation */
The number of instances of processes with the name
    'Assembly of Type A Components' that start
    after 2017-12-31 and end before 2019-01-01
```

<sup>5</sup> <http://hibernate.org>.

```

/* PIQE */
P ProcessName IS-EQUAL-TO 'Assembly of Type A Components
' AND Start IS-GREATER-THAN 2017-12-31 AND End
IS-LOWER-THAN 2019-01-01;
    
```

- **Query 2:** The number of process instances with the name “*Assembly of Type A Components*” that start after *2018-12-31* and end before today.

```

/* PIQE using user-friendly notation */
The number of instances of processes with the name '
Assembly of Type A Components' that start after 2018
-12-31 and end before $today
    
```

```

/* PIQE */
P ProcessName IS-EQUAL-TO 'Assembly of Type A Components
' AND Start IS-GREATER-THAN 2018-12-31 AND End
IS-LOWER-THAN $today;
    
```

Note that the comparison of the results obtained from **Query 1** and **Query 2** determines whether the increase of 24% has been reached in 2019; if today is in 2019.

Additionally, an example of a PPI that could be shown in the dashboard is “the number of successfully executed instances of the Assembly of Type B Component process.” The corresponding PIQE calculates the number of process instances with the name “*Assembly of Type B Components*” that are not canceled and that ended before today.

```

/* PIQE using user-friendly notation */
The number of instances of processes with the name '
Assembly of Type B Components' that end before $today
and are not canceled
    
```

```

/* PIQE */
P ProcessName IS-EQUAL-TO 'Assembly of Type B Components'
AND End IS-LOWER-THAN $today AND Canceled IS-EQUAL-TO
null;
    
```

## 6.2 DMN Enriched with PIQL

In the DMN context, PIQL can be applied as an extension of the standard by means of using expressions written in PIQL to define variables. These variables can be included in decision tables. A decision table defines a set of input variables whose values should be taken into account to make the decisions. In our approach, a PIQE

**Table 5** Adaptation of DMN Table 1 with PIQL

F*	Input						Output
#ID	Nº of pieces cod. #1657 (integer)	Nº of pieces cod. #6472 (integer)	Nº of pieces cod. #2471 (integer)	<i>\$avSt6</i> (integer)	<i>\$avSt12</i> (integer)	<i>\$avSt15</i> (integer)	Station (string)
#1	>= 2	>= 5	>= 1	0	–	–	Station 6
#2	>= 1	>= 2	>= 6	–	0	–	Station 12
#3	>= 7	Any	>= 1	–	–	0	Station 15
...	...	...	...	...	...	...	...

can be used to calculate the value of the input variable. For example, the DMN table in Sect. 3 (see Table 1) models the requirements that decide which task must be executed in accordance with the availability of pieces and stations in the context of the motivating scenario. Note that to calculate the availability of the different stations, we need to query not only the running instances of the “*Assembly of Type A Component Process*” but also the running instances of all the other processes that use Station 6, 12, or 15. These queries can be executed using the PIQL engine. Table 5 is an adaptation of the DMN, Table 1, that takes the advantages of using PIQL.

The main difference between Tables 1 and 5 is related to the use of PIQL to answer the questions, “*Is Station 6 available?*”, “*Is Station 12 available?*”, and “*Is Station 15 available?*”. In Table 1, the cells in the “*Is Station 6 available?*” column hold Boolean values, while the same cells in Table 5 hold integer values. These integer values are the results of the evaluation of PIQEs whose values are stored in the “*\$avSt6*” variable. Equally, the “*Is Station 12 available?*” and “*Is Station 15 available?*” columns are replaced with the values that hold the “*\$avSt12*” and “*\$avSt15*” variables. Note that the change of the data type (from Boolean to integer) has been carried out because PIQEs always return numeric values. This requirement is not a problem, because the availability of “*Station 6*” can be obtained by counting the number of task instances with the name “*Assemble in Station 6*” and with a null end date. If the result of this PIQE is 0, then the station 6 is available, that is, nobody is using this station. In contrast, if the result of this PIQE is greater than or equal to 1, then the station is not available. The PIQE that enables us to ascertain whether station 6 is available, and whose value is stored in the “*\$avSt6*” variable, is formulated as follows:

```
/* PIQE using user-friendly notation */
The number of instances of tasks with the name 'Assemble in
Station 6' that are not finalized
```

```
/* PIQE */
T TaskName IS-EQUAL-TO 'Assemble in Station 6' AND End
IS-EQUAL-TO null;
```

Another place in which PIQL can be used is in the context of the “*Validate Process*” task (see the “*Assembly of Type A Components*” process in Fig. 3). This

task checks whether the component satisfies all the requirements to be assembled. For example, one of the main requirements may be that all Type A Components have to pass through the three stations (6, 12, and 15), without a predefined order, to finish the assembly process. Thus, to check this requirement, the corresponding PIQE has to answer the following question: Has a specific process instance already executed the “Assemble in Station 6”, “Assemble in Station 12”, and “Assemble in Station 15” tasks? Note that this question makes sense in a DMN scenario in which a decision has to be made. In order to answer this question, three different queries should be evaluated:

1. The number of instances of tasks with a name that is equal to *Assemble in Station 6* and with a case id that is equal to \$id
2. The number of instances of tasks with a name that is equal to *Assemble in Station 12* and with a case id that is equal to \$id
3. The number of instances of tasks with a name that is equal to *Assemble in Station 15* and with a case id that is equal to \$id

After evaluating the queries, the decision task should check that the three results are greater than zero.

```

/* PIQE using user-friendly notation */
/* $Q_St6 */
The number of instances of tasks with the name 'Assemble in
  Station 6' with CaseId is equal to $id
/* $Q_St12 */
The number of instances of tasks with the name 'Assemble in
  Station 12' with CaseId is equal to $id
/* $Q_St15 */
The number of instances of tasks with the name 'Assemble in
  Station 15' with CaseId is equal to $id

```

```

/* PIQE */
/* $Q_St6 */
T TaskName IS-EQUAL-TO 'Assemble in Station 6' AND CaseId
  IS-EQUAL-TO $id;
/* $Q_St12 */
T TaskName IS-EQUAL-TO 'Assemble in Station 12' AND CaseId
  IS-EQUAL-TO $id;
/* $Q_St15 */
T TaskName IS-EQUAL-TO 'Assemble in Station 15' AND CaseId
  IS-EQUAL-TO $id;

```

### 6.3 Dataflow Enriched with PIQL

Following with the motivating scenario introduced in Sect. 3, the conditional event of the “Assembly of Type A Components” process (see Fig. 3) needs to *determine*

whether the person in charge of the process instance is executing any task. The availability of the person is stored in a variable of the process (whether local or global), and its value can be obtained as a result of evaluating a PIQL expression. Thus, the process uses the value of this variable to verify whether the event should be thrown. The PIQE that is evaluated to calculate the value of the variable mentioned previously should count the number of task instances executed by the user in charge of the process instance that remain unfinished. Thus, if the user in charge of the process is *Lydia Friend*, then the PIQE should be formulated as follows:

```
/* PIQE using user-friendly notation */
The number of instances of tasks executed by 'Lydia Friend'
that are not finalized
```

```
/* PIQE */
T Who IS-EQUAL-TO 'Lydia Friend' AND End IS-EQUAL-TO null;
```

Remember that PIQEs return numeric values, and as a consequence, verification of whether *Lydia Friend* is performing another task implies determining whether the number of tasks that this person is executing is greater than zero or, in other words, if the PIQE returns a number greater than zero.

## 7 Framework

Process Querying Framework (PQF) [11] establishes a set of components to be configured to create a process querying method. As a query language, PIQL implements some of these components. This section details these components and how they relate to the framework. In addition, the answers to the decision questions regarding the design that are implemented in PIQL, as suggested by the PQF, are also included.

PIQL enables the extraction of information from process instances and the tasks executed in these instances. During the execution of a specific process, not only the information that flows through this process is crucial for making decisions about the future evolution of the process, but also the information regarding the execution of other processes. This dependency between process executions is due to information and resources shared among them.

Therefore, since PIQL establishes a set of queries on top of the event log data repository, the functional requirements become a set of Create, Read, Update, and Delete (CRUD) operations over this repository. PIQL can read the system logs at run-time to extract the necessary knowledge about past and current process instances. Thus, the BPMS has to record the process and task executions and provide a mechanism to query these executions, while PIQL establishes a set of “read process” queries that isolates the user from technical details.

Figure 6 shows the main components of the PQF that PIQL supports. The *Model and Record* active components denoted by rectangles are essential for

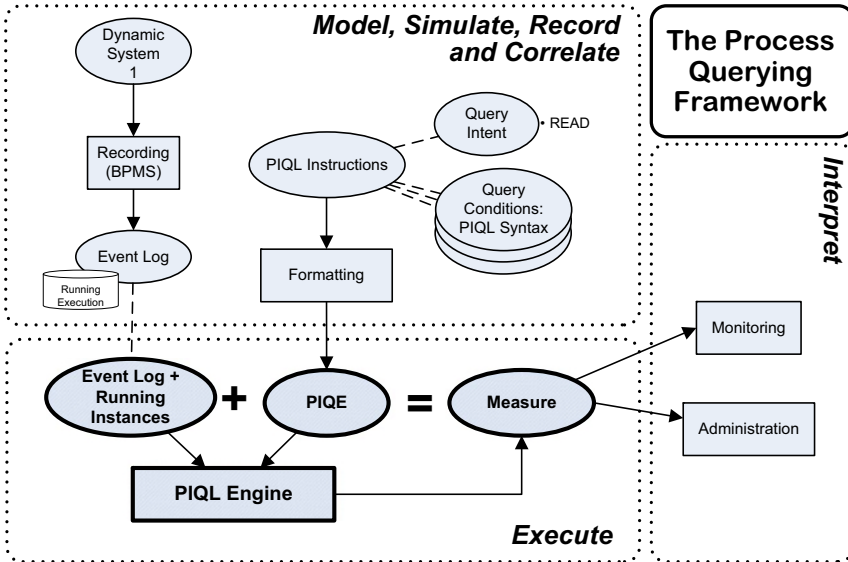


Fig. 6 Instantiation of the process querying framework for PIQL

PIQL. Firstly, a process model is defined in a BPMS, and the event-log repository is acquired automatically through the storage of the information generated by process executions. This storage is carried out by the BPMS itself (see Fig. 1). The majority of BPMSs enable the extraction of information about instances both from past executions and from current executions (although the processes have not finished). Therefore, this event-log repository includes information about finished and unfinished processes. Section 4 defines the set of process querying instructions supported by PIQL. PQF establishes that a Process Query Instruction is composed of a *process query intent* and a set of *process query conditions*. The *process query intent* of PIQL is to read in order to obtain a measurement (a specific quantity value), and the *process query conditions* are its parameters, i.e., the inputs required to execute a specific type of queries.

PIQL delegates the functionality of the “Prepare” component of the framework to BPMSs, which internally have the necessary mechanisms for efficiently querying the stored information and for providing the tools to take advantage of these querying mechanisms. Once both the information stored and the queries are established, the next step is the execution. As explained in Sect. 4, we define a PIQL engine in order to execute the queries. At run-time, the PIQL engine links the queries with the repositories and instantiates the specific values according to the query under execution. The results of the PIQL queries are measurements that are interpretable by users, which means the results may be integrated in a DMN table, visualized in a dashboard, or used to enrich the dataflow, as seen in this Chapter.

Finally, PIQL answers the decision questions regarding the design proposed by the PQF [11] in the following way:

- **DD1. Which behavior models to support?**

PIQL defines queries over process instances and tasks executed in these instances. Not only is the information generated by completed processes, but also the data generated by running processes is considered to be stored. When certain processes share the same resources and execute the same activities, the information related to the running instances becomes crucial. Therefore, this information is stored and queried using PIQL.

- **DD2. Which processes to support?**

PIQL establishes queries on top of finite process semantics, where the collection of processes lead to a terminate state. However, since during the execution of a process the information included in the instance is recorded, PIQL can extract the instance information without the need of finishing the process.

- **DD3. Which process queries to support?**

The *intent* of PIQL is the reading in order to obtain a measurement, that is, a numerical value. The operations in PIQL enable a combination of logic, comparative, and arithmetic operations. PIQL is capable of selecting specific behavior, i.e., process instance from a process repository, and of establishing a measurement.

## 8 Conclusions and Future Work

This chapter introduces a query language called Process Instance Query Language (PIQL) and the corresponding execution engine. In combination, these enable business experts to extract information from BPMSs. The syntax of the language has been specified using the Extended Backus-Naur Form (EBNF) grammar and its semantics has been specified. In order to validate the approach, various artifacts have been developed using a set of mature technologies: an implementation of the grammar, an engine that can be used to extract information from different platforms, and a driver for the integration of the engine with the *Camunda*<sup>TM</sup> BPMS platform. The artifacts are part of a modular and extensible platform ready to be integrated with other BPMS platforms.

In order to illustrate the potential of PIQL, a set of PIQL queries has been presented. Furthermore, a real-world example of an assembly business process in a factory has been introduced to demonstrate how PIQL may help nontechnical people extract information about the status of the factory and, as a consequence, improve the decision-making. The examples illustrate the flexibility of PIQL, and how it can contribute to the organizations. Finally, we show how PIQL implements the components of the PQF introduced in [11].

To conclude, future work will deal with (i) the extension of PIQL to enrich the type of filters that can be included in queries with elements such as the use of



resources, execution times, business load, and security aspects, (ii) the inclusion of other data models to be queried, such as business models or business data. Note that currently only business instances can be queried by means of PIQL, and (iii) the improvement of the PIQL engine performance using data caches, indexing, and other similar mechanisms.

**Reprint** Figures 1 and 5 are reprinted with permission from J. M. Pérez-Álvarez, M. T. Gómez López, L. Parody, and R. M. Gasca. *Process Instance Query Language to Include Process Performance Indicators in DMN*. IEEE 20th International Enterprise Distributed Object Computing Workshop. IEEE, 2016. pp. 1–8 (“© IEEE”).

**Acknowledgments** This work was funded by Junta de Andalucía (European Regional Development Fund ERD/FEDER) with the projects COPERNICA (P20\_01224) and METAMORFOSIS (FEDER\_US-1381375).

## References

1. del Río-Ortega, A., Resinas, M., Cabanillas, C., Cortés, A.R.: On the definition and design-time analysis of process performance indicators. *Inf. Syst.* **38**(4), 470–490 (2013). <http://doi.org/10.1016/j.is.2012.11.004>
2. Dumas, M., García-Bañuelos, L., Polyvyanyy, A., Yang, Y., Zhang, L.: Aggregate quality of service computation for composite services. In: Maglio, P.P., Weske, M., Yang, J., Fantinato, M. (eds.) *Service-Oriented Computing*, pp. 213–227. Springer, Berlin, Heidelberg (2010)
3. Dumas, M., Rosa, M.L., Mendling, J., Reijers, H.A.: *Fundamentals of Business Process Management*. Springer Publishing Company (2013)
4. Gómez-López, M.T., Borrego, D., Gasca, R.M.: Data state description for the migration to activity-centric business process model maintaining legacy databases. In: *BIS*, pp. 86–97 (2014). [http://doi.org/10.1007/978-3-319-06695-0\\_8](http://doi.org/10.1007/978-3-319-06695-0_8)
5. Gómez-López, M.T., Gasca, R.M., Pérez-Álvarez, J.M.: Decision-making support for the correctness of input data at runtime in business processes. *Int. J. Cooperative Inf. Syst.* **23**(4) (2014)
6. González, O., Casallas, R., Deridder, D.: Monitoring and analysis concerns in workflow applications: from conceptual specifications to concrete implementations. *Int. J. Cooperative Inf. Syst.* **20**(4), 371–404 (2011)
7. Maté, A., Trujillo, J., Mylopoulos, J.: Conceptualizing and specifying key performance indicators in business strategy models. In: *Conceptual Modeling - 31st International Conference ER 2012*, Florence, Italy, October 15–18, 2012. Proceedings, pp. 282–291 (2012)
8. Object Management Group: *Business Process Model and Notation (BPMN) Version 2.0*. OMG Standard (2011)
9. Object Management Group: *Decision Model and Notation. Reference Manual*. OMG Standard (2014)
10. Object Management Group: *Unified Modeling Language Reference Manual, Version 2.5*. OMG Standard (2015)
11. Polyvyanyy, A., Ouyang, C., Barros, A., van der Aalst, W.M.P.: Process querying: Enabling business intelligence through query-based process analytics. *Decis. Support Syst.* **100**, 41–56 (2017). <https://doi.org/10.1016/j.dss.2017.04.011>
12. Weske, M.: *Business Process Management: Concepts, Languages, Architectures*. Springer (2012). <http://doi.org/10.1007/978-3-642-28616-2>
13. Wirth, N.: What can we do about the unnecessary diversity of notation for syntactic definitions? *Commun. ACM* **20**(11), 822–823 (1977). <http://doi.acm.org/10.1145/359863.359883>

# **Part II**

## **Process Model Querying**

# The Diagramed Model Query Language 2.0: Design, Implementation, and Evaluation



Patrick Delfmann, Dennis M. Riehle, Steffen Höhenberger, Carl Corea, and Christoph Drodt

**Abstract** The Diagramed Model Query Language (DMQL) is a structural query language that operates on process models and related kinds of models, e.g., data models. In this chapter, we explain how DMQL works and report on DMQL's research process, which includes intermediate developments. The idea of a new model query language came from observations in industry projects, where it was necessary to deal with a variety of modeling languages, complex query requirements, and the need for pinpointing the query results. Thus, we developed the Generic Model Query Language (GMQL) tailored to deal with models of arbitrary modeling languages and queries that express model graph structures of any complexity. GMQL queries are formulas and professionals expressed the need to specify queries more conveniently. Therefore, the next development step was DMQL, which comes with functionality similar to GMQL, but allows to specify queries graphically. In this chapter, we describe both query languages, their syntax, semantics, implementation, and evaluation and come up with a new version of DMQL, which includes new functionality. Finally, we relate GMQL and DMQL to the Process Querying Framework.

## 1 Introduction

Manual analysis of process models has become unfeasible. Process models used in industry often contain thousands or even tens of thousands of elements [15, 19]. Process model querying has become established as a useful means of extracting relevant information out of process models, for instance, to support business process

---

P. Delfmann (✉) · D. M. Riehle · C. Corea · C. Drodt  
University of Koblenz-Landau, Koblenz, Germany  
e-mail: [delfmann@uni-koblenz.de](mailto:delfmann@uni-koblenz.de); [riehle@uni-koblenz.de](mailto:riehle@uni-koblenz.de); [ccorea@uni-koblenz.de](mailto:ccorea@uni-koblenz.de);  
[drodt@uni-koblenz.de](mailto:drodt@uni-koblenz.de)

S. Höhenberger  
University of Münster – ERCIS, Münster, Germany  
e-mail: [steffen.hoehenberger@ercis.uni-muenster.de](mailto:steffen.hoehenberger@ercis.uni-muenster.de)

compliance management [13] or business process weakness detection [34]. Several approaches and tools have been developed to provide automatic or semi-automatic process model querying. Process *model* querying is applied at design time in advance to and independent of the actual process execution [4, 13]. Most process model query languages work in a similar way: one creates a formalized query that describes the model part to be searched. Once the search is initiated, an algorithm processes the search by analyzing the model for occurrences that adhere to the query (matches). As a result, the query returns either TRUE or FALSE to denote that the model matches the query or not, or it returns the entire set of matches depending on the query approach. Figure 1 shows a small stylized example. Here, we query the model for parts in which a document is edited after it is signed, which may be a possible compliance violation or process weakness.

For detecting the described model part, the query in Fig. 1 describes two activities that follow each other (not necessarily immediately, as indicated by the dashed arrow). Both activities have an assigned document. The activities are labeled with the respective verbs and wildcard characters to allow partial label matches. The two documents are equated to enforce the detection of activities dealing with the same document. After the search is finished, the detected match is highlighted, cf. Fig. 1.

Based on our experiences regarding requirements of model querying from industry projects and various studies we conducted on existing query languages [2, 8, 10], we identified a research gap which lead us to develop the Generic Model Query Language (GMQL). GMQL aims to provide a structural query language fulfilling the following requirements:

1. The query language should be applicable to any kind of graph-like conceptual model, regardless of its modeling language or view (e.g., data model, process model, or organizational chart). This means that it should be possible to formulate queries for different model types (note that this does **not** mean that the same GMQL query fits all kinds of models, but it is necessary to formulate an event-driven process chain (EPC [30]) query for EPC models, a Business Process Model and Notation (BPMN<sup>1</sup>) query for BPMN models, a Petri net [24] query for Petri nets, etc.).
2. It should not be necessary to transform a conceptual model into a special kind of representation (e.g., a state machine or the like) before a query can be executed.
3. It should be possible to formulate structural queries of any complexity.
4. The query language should consider attributes of any kind, i.e., attributes of model vertices (such as type, name, cost, etc.) and edges (such as type, label, transition probability, etc.).
5. Every match of a query in the regarded model or model collection should be returned/highlighted.

Thus, the query language should realize *configurable subgraph homeomorphism* working on both directed and undirected, vertex and edge attributed multigraphs,

---

<sup>1</sup> ISO/IEC 19510:2013, <https://www.iso.org/standard/62652.html>.

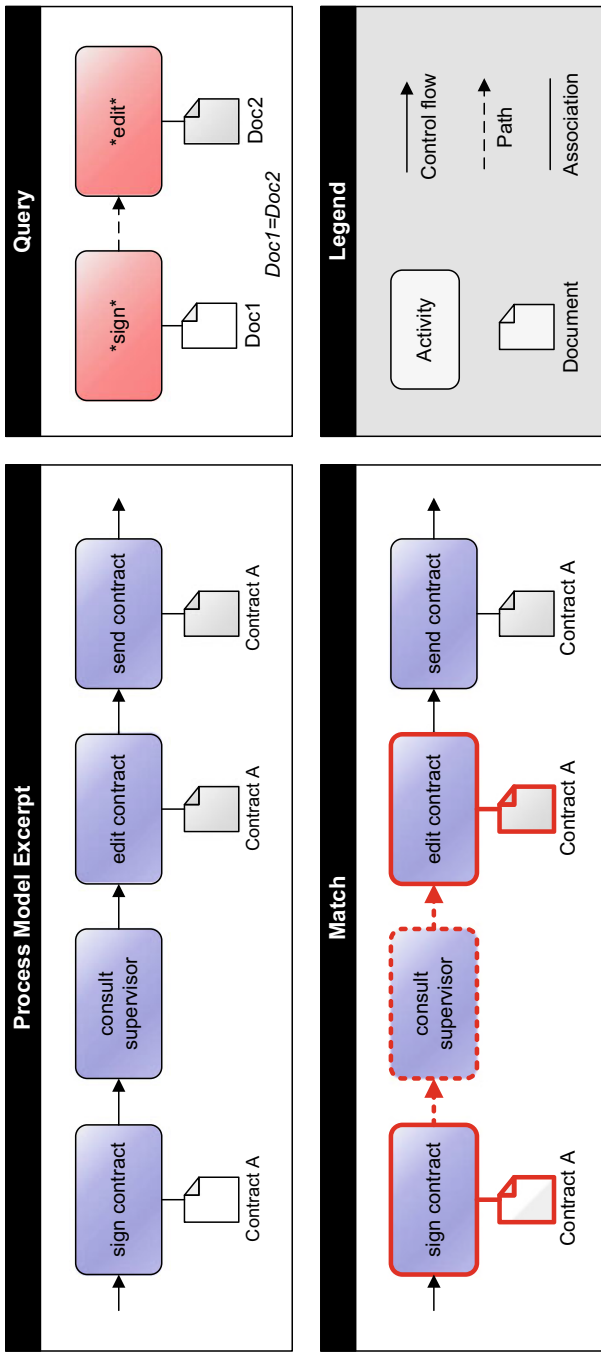
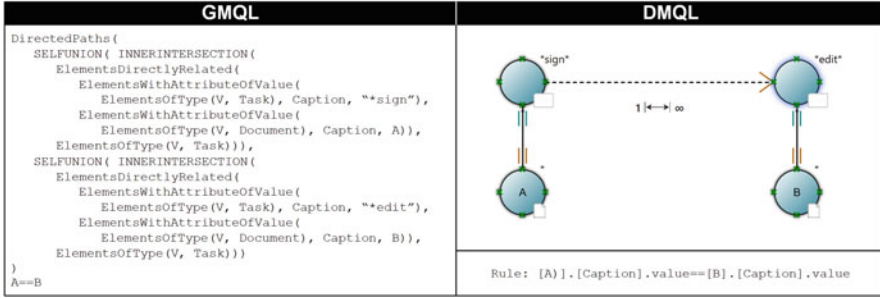


Fig. 1 Model analysis using queries



**Fig. 2** Exemplary queries in GMQL and DMQL

where parts of the attributes of the multigraphs represent the syntax of the examined model's modeling language. Unlike pure subgraph homeomorphism [20], *configurable* means that it must be possible to adjust the kinds of returned subgraphs according to the attributes of the vertices and edges and the length and the vertex/edge contents of the paths that get mapped.

GMQL realizes the mentioned requirements and consists of the query language, a query editor, and a query algorithm (we will further refer to all these components as GMQL). It takes a query as input, which consists of a composition of set-altering functions working on the sets of vertices and edges of the model to be queried. The user defines (nested) functions, which denote specific sets of graph structures, e.g., elements of a specific type, elements with a specific label, or elements that are connected to other elements with specific characteristics. The sets that result from the functions can be combined (e.g., unified, joined, subtracted, or intersected) to assemble a query step by step.

We received feedback that GMQL is indeed helpful, but it is also difficult to use as the query specification is hierarchical and text-based, and people are usually not used to think in sets and formal notations [4]. Moreover, the users emphasized the need for graphical specification of queries.

Therefore, we developed the Diagramed Model Query Language (DMQL) [8]. DMQL meets the aforementioned requirements, but, in addition, it provides a graphical query editor and slightly extended analysis possibilities compared to GMQL. DMQL is not based on GMQL, nor is it just a new concrete syntax for GMQL. It comes with a different way of query specification, query formalization, and search algorithm (for details, cf. Sect. 3). To provide a first impression of GMQL and DMQL queries, the example query of Fig. 1 is formalized with GMQL and DMQL in Fig. 2.

The development of GMQL and DMQL followed the Design Science Research (DSR) methodology proposed by Peffers et al. [21]. First, we developed GMQL and implemented it as a plug-in for the meta-modeling tool [ $\epsilon$ m].<sup>2</sup> GMQL was

<sup>2</sup> <https://em.uni-muenster.de>.

evaluated by applying it in a business process compliance management project in the financial sector [4]. The mentioned issues (mainly the ease of use) resulted in the development of DMQL, which was also implemented in [ $\varepsilon$ m] (partly demonstrated in [25]). DMQL was evaluated in two real-world scenarios, one dealing with compliance checking in the financial sector [18] and the other one dealing with business process weakness detection in several domains [9]. Based on the findings of these evaluations, in this chapter, we introduce DMQL 2.0, which provides enhanced functionality.

The remainder of this work is structured as follows: Sect. 2 introduces preliminaries including definitions of conceptual models, modeling languages, query occurrences, and matches, which are valid for both GMQL and DMQL. Section 3 explains GMQL and its abstract and concrete syntax and its semantics. Section 4 explains DMQL including its basic functionality (Sects. 4.1 to 4.4) and the extensions of DMQL 2.0 (Sect.4.5). Section 5 reports on the runtime complexity of GMQL's and DMQL's matching algorithms, their runtime performance measured empirically, and a utility evaluation. In Sect. 6, we position GMQL and DMQL within the Process Query Framework [23]. Section 7 closes this work with a conclusion.

## 2 Preliminaries

GMQL and DMQL allow to query models regardless of the respective modeling language, as both query languages are based on a meta-perspective of conceptual models [10]. In essence, any model is seen as an attributed graph, which is represented by its objects (vertices) and relationships (edges). The semantics of the model's objects and relationships is defined through attributes in the graph, e.g., by specifying the type or other properties of an object or relationship. The syntax is given through constraints on the graph that prescribe which kinds of vertices and edges can be connected to each other. Given the attributed graph, it is possible to formalize it (respectively, the original model) in a set-theoretic manner. Accordingly, GMQL and DMQL are based on a generic meta-model of conceptual models, shown in Fig. 3. Next, we provide a definition of a conceptual model based on the meta-model.

**Definition 2.1 (Conceptual Model)** A *conceptual model* is a tuple  $M = (V, E, T_v, T_e, C, \alpha, \beta, \gamma)$ , where  $V$  is a set of vertices and  $E$  is a set of edges between the vertices. Here,  $E$  is defined as  $E = E_D \cup E_U$ , where  $E_D \subseteq V \times V \times \mathbb{N}$  is the set of directed edges, and  $E_U = \{\{v_1, v_2, n\} \mid v_1, v_2 \in V, n \in \mathbb{N}\}$  is the set of undirected edges. We denote  $Z = V \cup E$  as the set of all vertices and edges.  $T_v(T_e)$  are sets of vertex (edge) types such as “activity”, “event”, “control flow”, “data input”, and the like.  $C$  is a set of captions, i.e., labels or other attribute values. For assigning vertices (edges) to vertex (edge) types or vertices and edges to their values, we use the functions  $\alpha : V \mapsto T_v$ ,  $\beta : E \mapsto T_e$ , and  $\gamma : Z \mapsto C$ , where  $\alpha$

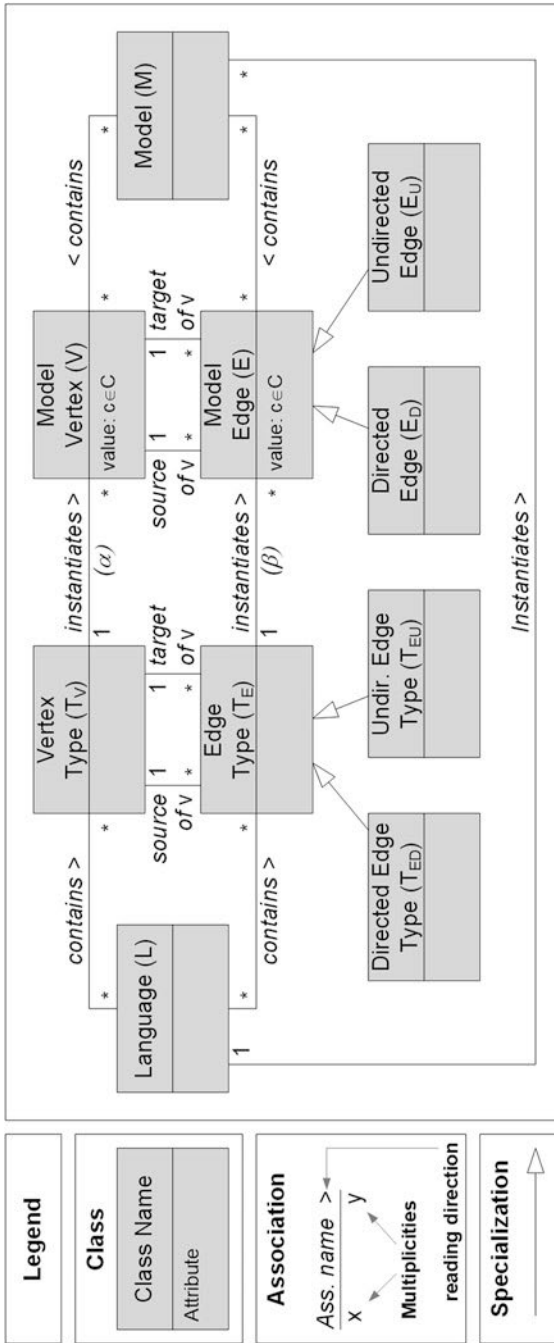


Fig. 3 Meta-model of conceptual models, adapted from [8, p. 479]



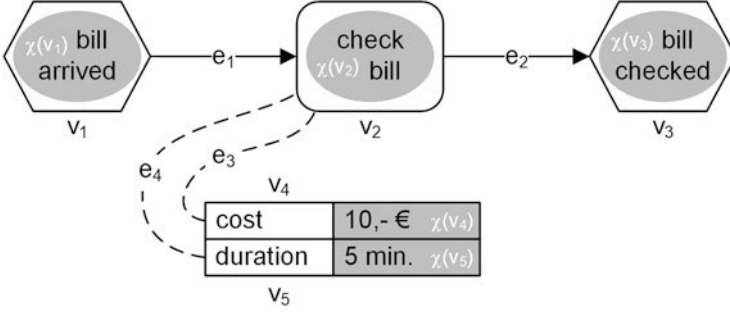


Fig. 4 Exemplary conceptual model

assigns a node in  $V$  to its type in  $T_v$ ,  $\beta$  assigns an edge in  $E$  to its type in  $T_e$ , and  $\gamma$  assigns an element in  $Z$  to a caption in  $C$ .

The reason why we use 3-tuples to define edges is that we need to be able to define multiple, distinguishable edges between the same vertices in multigraphs. This means that for each new edge between the same vertices, we create a new 3-tuple with an increased index  $n \in \mathbb{N}$ . Note that we explicitly refrain from the statement  $v_1 \neq v_2$  in the definition of undirected edges to allow undirected self-loops.

Vertices and edges can both have a value (i.e., their caption). It depends on the modeling language if vertices should have additional attributes (such as *duration*, *cost*, *description*, or the like). In such cases, we define additional attributes in the same way as vertices. That is, such additional attributes *are* vertices. However, they are visualized differently from common vertices. While common vertices are usually visualized as shapes, attributes are not. They can rather be accessed by opening a context menu. Attributes are assigned to vertices via undirected edges. Attributes also have values (e.g., “30 min” for the attribute *duration* of an *activity* vertex).

Figure 4 shows an exemplary process model in EPC notation. It can be formalized via  $V = \{v_1, v_2, v_3, v_4, v_5\}$ ,  $E = \{e_1, e_2, e_3, e_4\}$ ,  $e_1 = (v_1, v_2, 1)$ ,  $e_2 = (v_2, v_3, 1)$ ,  $e_3 = \{v_2, v_4, 1\}$ ,  $e_4 = \{v_2, v_5, 1\}$ ,  $T_v = \{\text{function}, \text{event}, \text{cost}, \text{duration}\}$ ,  $T_e = \{\text{control-flow\_ef}, \text{control-flow\_fe}, \text{function-cost}, \text{function-duration}\}$ ,  $C = \{\text{“bill arrived”}, \text{“check bill”}, \text{“bill checked”}, \text{“10,- €”}, \text{“5 min”}\}$ ,  $\alpha(v_1) = \alpha(v_3) = \text{event}$ ,  $\alpha(v_2) = \text{function}$ ,  $\alpha(v_4) = \text{cost}$ ,  $\alpha(v_5) = \text{duration}$ ,  $\beta(e_1) = \text{control-flow\_ef}$ ,  $\beta(e_2) = \text{control-flow\_fe}$ ,  $\beta(e_3) = \text{function-cost}$ ,  $\beta(e_4) = \text{function-duration}$ ,  $\gamma(v_1) = \text{“bill arrived”}$ ,  $\gamma(v_2) = \text{“check bill”}$ ,  $\gamma(v_3) = \text{“bill checked”}$ ,  $\gamma(v_4) = \text{“10,- €”}$ , and  $\gamma(v_5) = \text{“5 min”}$ .

**Definition 2.2 (Modeling Language)** A modeling language is a pair  $L = (T_V, T_E)$ .  $T_V$  describes a set of vertex types.  $T_E = T_{ED} \cup T_{EU}$  is a set of edge types, where  $T_{ED} \subseteq T_V \times T_V \times \mathbb{N}$  is a set of directed edge types and  $T_{EU} \subseteq \{\{t_{vx}, t_{vy}, n\} | t_{vx}, t_{vy} \in T_V, n \in \mathbb{N}\}$  is a set of undirected edge types (cf. [8]).

The definition above makes it possible to allow different edge types between the same vertex types, as they are present in several modeling languages. For instance, in

EPCs, it is common to use both the edge type *input* and the edge type *output* between functions and data objects. As another example, class diagrams allow four edge types, *composition*, *association*, *generalization*, and *aggregation* between classes. Note that we explicitly refrain from the statement  $t_{vx} \neq t_{vy}$  in the definition of undirected edge types to allow undirected self-loops.

**Definition 2.3 (Query Matches)** A query  $Q$  that is applied to a model  $M$  returns a set of query occurrences, which are subsections of the queried model graph. The details of the formal definitions of GMQL and DMQL queries can be found in the dedicated subsections below.

Given a conceptual model  $M$ , a *query occurrence* of a query  $Q$  in  $M$  is a set  $Z' \subseteq Z$ , where  $Z$  is the set of all vertices and edges (see Definition 1).

GMQL and DMQL can be used to query a conceptual model  $M$  and as a result present the user a set of all query occurrences in  $M$ . We call such a set of query occurrences *query matches*.

Let  $\mathfrak{M}$  be the set of all conceptual models,  $\mathfrak{Q}$  the set of all queries, and  $\mathfrak{P}_M$  the set of all possible query occurrences. Then, the query matches are defined as a function  $Match: \mathfrak{M} \times \mathfrak{Q} \rightarrow \mathcal{P}(\mathfrak{P}_M)$ , which maps a specific model  $M$  to the set of all query occurrences of  $\mathfrak{Q}$  contained in the model.

In other words,  $Match(M, Q)$  returns a set of subgraphs of  $M$ , which correspond to query  $Q$ . Detailed formalizations of how matches are identified are given in Sects. 3 and 4.

### 3 The Generic Model Query Language (GMQL)

GMQL provides set-altering functions and operators, which exploit the set-based representation of a model to execute queries. GMQL queries return result sets, i.e., parts of models, which satisfy the conditions of the queries [10]. A GMQL query can be composed of different nested functions and operators to produce arbitrarily complex queries.

GMQL can be used to find model subgraphs, which are partly isomorphic and partly homeomorphic with a predefined query. The query execution returns every query occurrence and thus allows to present the user all sections of the model that satisfy the query. In the following, we introduce the syntax and semantics of GMQL.

#### 3.1 Syntax

A GMQL query consists of functions and operators, cf. [10]. A GMQL function has an identifier and at least one input parameter. For instance, regard the GMQL function

$$ElementsOfType(X, t). \tag{1}$$

The identifier of this function is “ElementsOfType”. This indicates that the function can be used to return model elements of a certain type. The function takes two parameters. The first parameter is a set  $X \subseteq Z$  (i.e., any set of model elements). This set is the search space. It provides the information that is meant to be queried by the function. In the statement of listing (1), the user could, for instance, provide the set  $Z$  of all elements of a model as an input to the function. This would allow to search the entire model. The second parameter is a parameter expression. In GMQL, parameter expressions are inputs to functions which are not sets, that is, single values. Depending on the specific functions, these could be integers, strings, values of variables, or in the case of listing (1), an element type the elements of  $Z$  should be of (i.e., a value of an enumerated type). For the example in Fig. 4, the query *ElementsOfType*( $Z$ , “Event”) would return the query occurrences  $\{v_1\}$  and  $\{v_3\}$ .

Other than providing sets as inputs to functions, it is also possible to nest functions and operators. To this end, a function can, for instance, be provided as the parameter  $X$ , e.g.,

$$\textit{ElementsOfType}(\textit{GMQLFunction}(\textit{Parameters}), t). \quad (2)$$

Here, an initial search space is modified via the inner *GMQLFunction*, with its respective parameters, and this modified search space is then used as a basis for the introduced *ElementsOfType* function. Nesting functions and operators allows to define complex queries tailored by the user. Next, we define the syntax of GMQL [10] using the Extended Backus-Naur Form (EBNF).<sup>3</sup> A GMQL query  $Q$  is defined as follows:

```
Q = subQueryExpression {", " equationExpression};

subQueryExpression = functionExpression |
                    operatorExpression |
                    setExpression;
```

Each query consists of a *subQueryExpression* that carries the actual query and, optionally, one or more *equationExpression*(s) used to compare variables from the query. A *subQueryExpression* is either a *functionExpression*, an *operatorExpression*, or a *setExpression*.

A *setExpression* is the simplest input of a query and represents the basic input set  $V$ ,  $E$ , or  $Z$ .

```
setExpression = "V" | "E" | "Z";
```

<sup>3</sup> ISO/IEC 14977:1996, <https://www.iso.org/standard/26153.html>.

A `functionExpression` consists of the function's identifier (see the list of possible function identifiers below), followed by an opening bracket and one or more `subQueryExpressions` or `parameterExpressions`. Using the `subQueryExpression` as a function parameter makes it possible to nest queries.

```
functionExpression = functionIdentifier "(" subQueryExpression
["," (parameterExpression | subQueryExpression)] [", "
(parameterExpression | subQueryExpression)] ")";
```

```
functionIdentifier = "ElementsOfType" |
"ElementsWithAttributeOfValue" |
"ElementsWithAttributeOfDatatype" |
"ElementsWithRelations" |
"ElementsWithSuccRelations" |
"ElementsWithPredRelations" |
"ElementsWithRelationsOfType" |
"ElementsWithSuccRelationsOfType" |
"ElementsWithPredRelationsOfType" |
"ElementsWithNumberOfRelations" |
"ElementsWithNumberOfSuccRelations" |
"ElementsWithNumberOfPredRelations" |
"ElementsWithNumberOfRelationsOfType" |
"ElementsWithNumberOfSuccRelationsOfType" |
"ElementsWithNumberOfPredRelationsOfType" |
"ElementsDirectlyRelated" | "AdjacentSuccessors" |
"Paths" | "DirectedPaths" | "Loops" |
"DirectedLoops" | "PathsContainingElements" |
"DirectedPathsContainingElements" |
"PathsNotContainingElements" |
"DirectedPathsNotContainingElements" |
"LoopsContainingElements" |
"DirectedLoopsContainingElements" |
"LoopsNotContainingElements" |
"DirectedLoopsNotContainingElements";
```

A `parameterExpression` is used to input a single value into a function, such as vertex or edge types. It is either an `Integer` allowing arbitrary numbers, an `AttributeValue` allowing arbitrary strings, an `ElementType` allowing one of the element types in  $(T_E \cup T_V)$ , a variable used to make comparisons in equation expressions, or an `AttributeDataType`.

```
parameterExpression = Integer | ElementType |
  AttributeDataType | AttributeValue | Variable;

AttributeDataType = "INTEGER" | "STRING" | "BOOLEAN" |
  "ENUM" | "DOUBLE";
```

An `operatorExpression` is either a `unaryOperatorExpression` or a `binaryOperatorExpression`. Both start with an identifier, either a `unaryOperatorIdentifier` or a `binaryOperatorIdentifier`, followed by an opening bracket, one or two parameters, and a closing bracket. The possible operator identifiers can be taken from the list below. `SELFUNION` and `SELFINTERSECTION` take one parameter, and all the others take two.

```
operatorExpression = unaryOperatorExpression |
  binaryOperatorExpression;

unaryOperatorExpression = unaryOperatorIdentifier
  "(" subQueryExpression ")";
unaryOperatorIdentifier = "SELFUNION" | "SELFINTERSECTION";

binaryOperatorExpression = binaryOperatorIdentifier
  "(" subQueryExpression "," subQueryExpression ")";
binaryOperatorIdentifier = "UNION" | "INTERSECTION" |
  "COMPLEMENT" | "JOIN" | "INNERINTERSECTION" |
  "INNERCOMPLEMENT";
```

Finally, an `equationExpression` is used to compare values of variables that have been used in a query.

```
equationExpression = Variable ("=" | "!=" | "<" | ">" |
  "<=" | ">=") Variable;
```

The values of the variables are calculated at runtime. For instance, the query

$$\text{DirectedPaths}(\text{ElementsOfType}(V, A), \text{ElementsOfType}(V, B)),$$

$$A = B,$$

returns all directed paths that start and end with a vertex of the same type (recall that  $V$  is the set of all vertices of the input model). If the query was applied to an EPC, for instance, it would return all directed paths from function to function, from event to event, from XOR connector to XOR connector, etc. Each path that matches the query is returned as one result set. This means that the overall result of the query can consist of several sets. Each such set contains the start and end vertex of the path as well as all vertices and edges *on* the path.

### 3.2 Semantics, Notation, and Query Example

A GMQL query is performed by applying it to the set-based formalization of a model in order to retrieve the respective matches. GMQL provides an extensive set of functions and operators. Functions allow to query for element types, element values, relations to other elements, paths, and loops. Operators, such as UNION, INTERSECTION, COMPLEMENT, and JOIN, allow to combine functions to create arbitrarily complex queries. Due to space limitations, we omit a full specification of functions' and operators' semantics. Please refer to [10] for details.

To provide an example, we revisit the exemplary function  $ElementsOfType(X, t)$ . As discussed, this function returns a subset of elements from  $X$  which are of type  $t$ . We recall that  $Z$  is the set of all elements of a conceptual model  $M$ ,  $X \subseteq Z$ , and  $\alpha$  and  $\beta$  are functions that assign vertices and edges to their types. The semantics of  $ElementsOfType$  is then defined by  $ElementsOfType(X, t) = \{x \in X | \alpha(x) = t \vee \beta(x) = t\}$ .

When computing matches, query occurrences are determined through an application of the respective function or operator semantics. In case of nested functions, the output of inner functions is passed as input to outer functions, and the semantics is evaluated accordingly. The overall query result builds up incrementally, allowing to traverse the resulting query tree in post order.

To provide an example, we consider a scenario from business process weakness detection. In business process models, certain constructs may be syntactically correct but represent a shortcoming (i.e., represent an inefficient or illegitimate part of the process) and should hence be avoided. To detect such parts, we can apply a corresponding GMQL query. The query returns all occurrences of the weakness, and the user can decide how the weak sections of the process model shall be improved. An example of such a weakness pattern is a document that is printed and scanned later on. The user may be interested in all parts of the process model where such a "print-scan" pair of activities exists.

The corresponding GMQL query would require several functions and operators, the semantics of which we introduce exemplarily. To express that we are searching for execution paths, we use the function  $DirectedPaths(X_1, X_2)$ , which takes two sets of elements, where the first set represents possible starting points of the paths and the second one represents possible end points of the paths.  $DirectedPaths$  returns all directed paths that start in an element of  $X_1$  and end in an element of  $X_2$ , where each path is captured as a set of elements (i.e., both the vertices and edges of the path). To express that the process model's activities are annotated with documents, we make use of the GMQL function  $AdjacentSuccessors(X_1, X_2)$ . It returns all pairs of elements (one from  $X_1$  and the other from  $X_2$ ) that are connected by a directed edge, where the source of the edge is an element from  $X_1$  and the target of the edge is an element from  $X_2$ . To access the contents (i.e., the labels) of model vertices, we use the function  $ElementsWithAttributeOfValue(X, t, u)$  that

takes a set of elements  $X$ , an attribute type  $t$ , and a value  $u$  as input and returns all elements from  $X$  that carry an attribute of type  $t$ , which in turn carries the value  $u$ .

```
DirectedPaths (
  SelfUnion (InnerIntersection (
    ElementsOfType (V, activity) ,
    AdjacentSuccessors (
      ElementsWithAttributeOfValue (
        ElementsOfType (V, activity) , caption, "*print*" ) ,
        ElementsWithAttributeOfValue (
          ElementsOfType (V, document) , caption, A ) ) )
  SelfUnion (InnerIntersection (
    ElementsOfType (V, activity) ,
    AdjacentSuccessors (
      ElementsWithAttributeOfValue (
        ElementsOfType (V, document) , caption, B ) ,
        ElementsWithAttributeOfValue (
          ElementsOfType (V, activity) , caption, "*scan*" ) ) ) ) )
A=B
```

Consider the query shown above. The most inner function *ElementsWithAttributeOfValue(ElementsOfType(V, activity), caption, "\* print \*")* returns all activity vertices that carry “print” in their names. The other inner function *ElementsWithAttributeOfValue(ElementsOfType(V, document), caption, A)* returns all document vertices that carry a specific name not known at this stage, however represented through the variable  $A$ . The next outer function *AdjacentSuccessors* takes the results of the two inner functions as input and returns all model sections where an activity vertex with “print” in its name is connected to a document vertex via a directed edge pointing to the document vertex. The result of the other function *AdjacentSuccessors* (sixth last row of the listing) is calculated analogously. As the results are model sections, *AdjacentSuccessors* consequently returns a set of sets as a result. To find paths that reach from a “print” activity vertex having a document as output to a “scan” activity vertex having a document as input, we search for paths from the former to the latter. *DirectedPaths* takes sets as input and not sets of sets. Thus, we need to extract the activity vertices out of the intermediate result sets before we use them as inputs for the *DirectedPaths* function. We do this through the use of operator *INNERINTERSECTION*, which performs a set intersection of *ElementsOfType(V, activity)* with the inner sets of the results of the *AdjacentSuccessor* function. What remains is the activity vertices we searched for, still in sets of sets. Hence, the last step is to transform the sets of sets into single sets, which is done via *SELFUNION*. The outer function *DirectedPaths* now takes the single sets as input that only carry the activity vertices having documents annotated. The final result is a set of paths, each from an activity with a document annotated to another activity with a document annotated. Each path is encoded as a set of vertices and edges that lie on the path. To assure that we only find sections of a

process where the same document is printed and scanned subsequently, we use the variables  $A$  and  $B$  for the names of the documents and require that their values are equal (cf. the last line of the listing).

The rest of the functions and operators of GMQL work similarly to those above and allow to search for different kinds of structures and to assemble the sub-queries in novel ways. To conclude, we can construct arbitrarily complex queries that address the granularity of elements, the relationships between elements, constraints regarding paths and loops, or specific multiplicities of certain elements or relations, that is, counting. The queries can then be used to search conceptual models for query occurrences and present these to the user.

Results of GMQL queries are highlighted in the model that is currently examined by surrounding the vertices and edges involved in a query occurrence with colored bold lines. If there is more than one query occurrence, the user can browse through the results [10]. The GMQL function names are given in a comprehensible manner to increase usability for unexperienced users. For further details on the semantics of individual functions and operators, refer to [10].

### 3.3 *The Transition from GMQL to DMQL*

While GMQL is a powerful language for querying arbitrary conceptual models in the context of the requirements identified in [8], the queries must be formulated using textual formalisms. A utility evaluation of GMQL [4] has shown that users appreciate graphical specification of queries, as formal textual statements are regarded complicated with low ease of use. Hence, we developed a graphical concrete syntax for GMQL called vGMQL [32]. However, we experienced that assembling queries through defining, reducing, nesting, intersecting, unifying, and joining sets of model elements as it is required by the abstract syntax of GMQL merely with graphical symbols does not increase its ease of use. In other words, we cannot “draw” a query in a way such that it looks like the kinds of model sections we are searching for. Therefore, we created DMQL [8]. This model query language is built on the same formal foundation as GMQL (see Sect. 2) and supports graphical specification of queries. Particularly, the concrete syntax of DMQL is shaped in a way such that a query looks much like the kinds of model sections that should be searched for. Consequently, DMQL is not just a new concrete syntax for GMQL (like vGMQL is), but a completely new language with an own way of specifying queries, an own abstract and concrete syntax, and own semantics. The following section introduces DMQL and discusses its relation to GMQL.

## 4 **The Diagramed Model Query Language (DMQL)**

While GMQL allows the user to define textual queries, DMQL focuses on a visual syntax [8–10]. This helps users to formalize queries in a more user-friendly way. DMQL’s matching algorithm, unlike the one of GMQL, is based on an



adapted graph matching algorithm known from algorithmic graph theory [8, 11]. In particular, the query algorithm extends subgraph homeomorphism [20] working on both directed and undirected, vertex and edge attributed multigraphs, where parts of the attributes of the multigraphs encode the syntax of the examined models. Thus, DMQL does not build up a query through set-altering functions and operators, like GMQL does, but uses a visual query graph as input. DMQL then searches for extended-homeomorphic occurrences of the input graph in the queried models [8]. In order to make the visual graph suitable as input for the algorithm, it is transformed into a formal representation. Like GMQL, DMQL is capable of processing any conceptual model, and therefore it is based on the same meta-model as GMQL (see Fig. 3). Thus, the definition of the conceptual model, the modeling language, and the query occurrence and query match is identical for both query languages and is listed in Sect. 2, Definitions 1 to 3.

DMQL allows to query models of multiple modeling languages. This means that we can define queries whose occurrences might span models of different modeling languages. For instance, a query expressing the so-called *four-eye-principle* requires that a document is checked twice in a process, where the second person who checks is the supervisor of the person who makes the first check. The information about the order of the activities can be taken from the process model; however, the information if the second person is a supervisor of the first one can only be taken from an organizational chart. Thus, a query that checks if the *four-eye-principle* is realized in a business process has to work on multiple modeling languages.

## 4.1 Syntax

A DMQL query is a tuple. It is defined as  $Q = (V_Q, E_Q, P_V, P_E, \delta, \varepsilon, \Gamma, G)$ . In DMQL, a query is a directed multigraph consisting of vertices  $V_Q$  and edges  $E_Q \subseteq V_Q \times V_Q \times \mathbb{N}$ . Each vertex  $v_Q$  in  $V_Q$  and each edge  $e_Q$  in  $E_Q$  can be assigned properties, which define how the query vertices and edges should be mapped to model vertices and edges. The properties are assigned by two functions:  $\delta : V_Q \rightarrow P_V$  and  $\varepsilon : E_Q \rightarrow P_E$ .  $P_V$  and  $P_E$  are sets of tuples and contain vertex and edge properties, where  $P_V \subseteq VID \times VCAPTION \times VTYPES$  and  $P_E \subseteq EID \times ECAPTION \times DIR \times MINL \times MAXL \times MINVO \times MAXVO \times MINEO \times MAXEO \times VTYPESR \times VTYPESF \times ETYPESR \times ETYPESF \times \Theta$ .  $\Gamma$  is the set of modeling languages the query is applicable to, and  $G$  is a set of global rules (see explanation at the end of this section).

A property of a query vertex is a tuple and consists of the following components:  $VID$  is a set of query vertex IDs, and  $vid \in VID$  is a string that represents the ID of one query vertex. IDs are not immediately used for matching but to build rules within a query (similar to the equation expression in GMQL, see section on global rules below).  $VCAPTION$  is the set of all vertex matching phrases. Correspondingly,  $vcaption \in VCAPTION$  defines the caption that a vertex in a model should have to be matched.  $vcaption$  is a string and can contain wildcards.  $VTYPES = \mathcal{P}(T_V)$  is

the set of all possible sets of vertex types. Consequently,  $vtypes \in VTYPES$  is a set of vertex types. Each vertex in a model that has one of these types is a matching candidate.

A property of a query edge is a tuple and consists of the following components:  $EID$  is a set of query edge IDs that are used analogously to vertex IDs. Correspondingly,  $eid \in EID$  is a string that represents the ID of one query edge.  $ECAPTION$  is the set of all edge matching phrases.  $ecaption \in ECAPTION$  defines the caption that an edge in a model should have to be matched. It is a string and can contain wildcards.  $DIR = \mathcal{P}(\{org, opp, none\})$  is the set of all possible combinations of edge directions. Therefore,  $dir \in DIR$  is a subset of  $\{org, opp, none\}$  (i.e.,  $dir \subseteq \{org, opp, none\}$ ) and defines which direction a model edge should have in order to be mapped to the query edge. If a model edge to be mapped must have the same direction as the original query edge,  $org$  needs to be chosen. If we choose  $opp$ , then the model edge must have the opposite direction of that in the query. Lastly,  $none$  means that the model edge to be mapped must be undirected. We can combine these three options in an arbitrary way, for instance, if we choose  $org$  and  $opp$ , we allow the mapped edges to have any direction.

$MINL$ ,  $MAXL$ ,  $MINVO$ ,  $MAXVO$ ,  $MINEO$ , and  $MAXEO$  are sets of natural numbers including  $-1$ , i.e.,  $MINL = MAXL = MINVO = MAXVO = MINEO = MAXEO = \mathbb{N}_0 \cup \{-1\}$ .  $minl \in MINL$  and  $maxl \in MAXL$ , ( $minl \leq maxl$ ) define if the query edge should be mapped to a single edge or a path in the model. If both are equal to 1, then the query edge is mapped to a single edge in the model. If  $maxl > minl$ , then the query edge is mapped to a path with at least  $minl$  and at most  $maxl$  edges. If  $maxl = -1$ , then the paths to be mapped can have any maximum length. The properties explained next are only evaluated if  $maxl \geq 2$ .

$minvo \in MINVO$ ,  $maxvo \in MAXVO$ ,  $mineo \in MINEO$ , and  $maxeo \in MAXEO$  define if a path that gets mapped to a query edge is allowed to cross itself, that is, run multiple times over the same model vertices and/or edges (so-called vertex and edge overlaps). For instance, if  $minvo = 0 \wedge maxvo = 1$ , then the path to be mapped is allowed to cross itself once in one vertex, that is, one vertex on the path is allowed to be visited twice (but does not have to because  $minvo = 0$ ). Edge overlaps are controlled in a similar way by  $mineo$  and  $maxeo$ . If  $maxvo = -1$  ( $maxeo = -1$ ), then the matching algorithm allows any number of vertex (edge) visits.

$vtypesr \in VTYPESR$  (with  $VTYPESR = \mathcal{P}(T_V)$ ) is a set of vertex types and requires that for each vertex type contained in  $vtypesr$  there is at least one vertex belonging to that type on the path.  $vtypesf \in VTYPESF$  (with  $VTYPESF = \mathcal{P}(T_V)$ ) is also a set of vertex types and requires that *none* of the vertices on the path belongs to any of the vertex types contained in  $vtypesr$ .

$etypesr \in ETYPESR$  (with  $ETYPESR = \mathcal{P}(T_E)$ ) and  $etypesf \in ETYPESF$  (with  $ETYPESF = \mathcal{P}(T_E)$ ) work analogously to  $vtypesr$  and  $vtypesf$ , however with edge types. Element types cannot be both required and forbidden on paths, so  $etypesr \cap etypesf = \emptyset$  and  $vtypesr \cap vtypesf = \emptyset$ . All properties that relate to paths are ignored as soon as  $maxl = 1$ . An exception is  $etypesr$ , which relates to a single edge in case  $maxl = 1$ . Then, a single matched model edge must have one of the types in  $etypesr$ .

$\theta \in \Theta$  is a set of tuples, where  $\Theta = \mathcal{P}(\Omega \times \{req, preq, forb, pforb\})$ . Each such tuple refers to another query  $Q'$  in  $\Omega$  and a constraint  $req, preq, forb$ , or  $pforb$ . The constraint determines whether occurrences of  $Q'$  are allowed/required to lie on the path that is mapped to the query edge. For example, a query edge with  $\theta \neq \emptyset$  can be mapped to a path in a model if the following requirements hold [8, p. 485]:

- $(Q', req) \in \theta$ : The path contains at least one complete occurrence of  $Q'$ , i.e., all elements of at least one occurrence of  $Q'$  lie on the path.
- $(Q', preq) \in \theta$ : The path contains at least one partial occurrence of  $Q'$ , i.e., at least one element of at least one occurrence of  $Q'$  lies on the path..
- $(Q', forb) \in \theta$ : The path does not contain any element of any occurrence of  $Q'$ , i.e., no element of any occurrence of  $Q'$  lies on the path.
- $(Q', pforb) \in \theta$ : The path does not contain a complete occurrence of  $Q'$  but can contain parts of it.

Finally,  $G = (gmaxvo, gmaxeo, R)$  is a set of global rules, which apply to the overall query.  $gmaxvo$  and  $gmaxeo$  are numbers handled similarly to  $maxvo$  and  $maxeo$ . They define if the matched paths are allowed to overlap among each other and how often.  $R$  is a set of rules that are operating on the properties of query vertices and edges. A rule  $r \in R$  is an equation that must obey the following syntax given in EBNF [8, p. 485]:

```
rule = ["NOT"] subrule | ["NOT"] "(" subrule boolop subrule ")";
subrule = rule | comparison;
boolop = "AND" | "OR" | "XOR";

comparison = compitem compop compitem;
compop = "==" | "!=" | "LIKE" | "<" | ">" | "<=" | ">=";
compitem = number_expression | string_expression | boolean;

number_expression = number calcop number;
number = "(" number_expression ")" | number_function |
  number_primitive;
number_function = "[" ID "]" . [" ID "]" . value | "[" ID "]" .
  property "." aggregate;
property = "PREDECESSORS" | "SUCCESSORS" |
  "UNDIRECTED_NEIGHBORS" | "NEIGHBORS" |
  "OUTGOING_EDGES" | "INCOMING_EDGES" |
  "UNDIRECTED_EDGES" | "EDGES" | "NODES";
aggregate = "count()" | "max([" TV "])" | "min([" TV "])" |
  "avg([" TV "])" | "sum([" TV "])";
calcop = "+" | "-" | "*" | "/" | "%";
number_primitive = INTEGER | FLOAT;

string_expression = single_string {"+" string_expression};
single_string = string_function | string_primitive;
```

```

string_function = "[" ID "]"."type" | "[" ID "]"."[" ID "]"."value";
string_primitive = STRING;

Boolean = "[" ID "]"."[" ID "]"."value" | "TRUE" | "FALSE";
ID = VID | EID;

```

Such a rule evaluates properties of query matches and returns either *TRUE* or *FALSE*. Only those matches the rules of which evaluate to *TRUE* are returned. The properties of everything that can be given an ID in the query can be compared (i.e., vertices, edges, and paths). For instance, we can compare not only values of attributes (i.e., strings, numbers, or Boolean values) of vertices and edges and types of vertices and edges (i.e., strings) but also other aggregated properties such as the number of adjacent edges or the maximum value of an attribute of the neighboring vertices (i.e., numbers).

Strings, numbers, and Boolean values can be compared via comparison operators (*compop*) that evaluate to *TRUE* or *FALSE*, while *LIKE* only works on strings and  $>$ ,  $<$ ,  $\leq$ , and  $\geq$  only work on numbers. Attribute values can be strings, numbers, or Boolean values. Properties describe the environment of a vertex or edge (*PREDECESSORS*, *SUCCESSORS*, etc.) and can be evaluated with different aggregate functions (*count*, *max*, etc.) and always return numbers. Numbers can be combined with common math operators (*calcop*), and strings can be concatenated using the  $+$  operator.

Consider the following example describing a rule that works on a fictional query containing a vertex  $v_1$ , and an edge  $e_1$ , among others, where  $vid(v_1) = \text{“A”}$ ,  $eid(e_1) = \text{“B”}$ ,  $minl(e_1) = 1$ , and  $maxl(e_1) = -1$ . The following rule would require that the number of edges a vertex mapped to  $v_1$  is connected to must equal the sum of the values of attributes of the type *cost* connected to each of the vertices that exist on the path mapped to  $e_1$ .<sup>4</sup>

$$[A].edges.count() == [B].nodes.sum([cost])$$

## 4.2 Notation

DMQL queries are mainly visualized through graphical symbols (cf. Fig. 5 and [8]). As DMQL queries have several properties, we cannot display all of them with graphical symbols without overcomplicating the query. Hence, some of the properties are specified with lists and menus. Consequently, it only makes sense to use DMQL when it is implemented as a modeling software add-on.

---

<sup>4</sup> A full description of properties and aggregate functions can be found at [https://em.uni-muenster.de/wiki/GraphBasedModelAnalysisPlugin/DMQL#Global\\_Rules](https://em.uni-muenster.de/wiki/GraphBasedModelAnalysisPlugin/DMQL#Global_Rules).

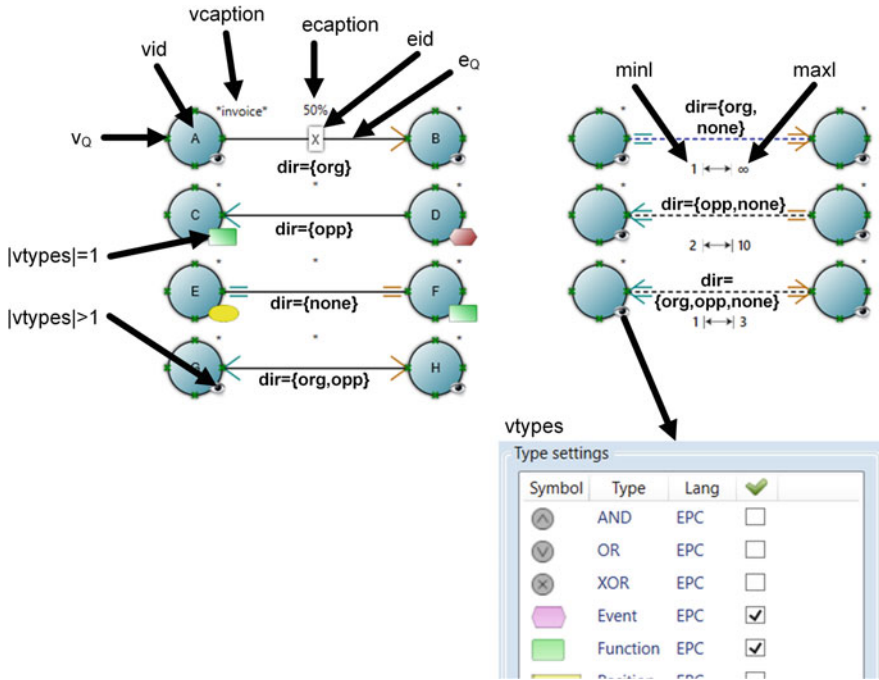


Fig. 5 DMQL concrete syntax examples: Basics

In Fig. 5, shaded circles represent query vertices  $V_Q$  with their IDs placed within them. Their captions  $vcaption$  are placed onto the upper right corner (e.g., “\*invoice\*” in the figure). Depending on the number of allowed vertex types  $vtypes$ , we either see the corresponding symbol of the vertex type at the lower right corner, if  $|vtypes| = 1$ , or a small eye symbol, otherwise. The  $vtypes$  property is set by using a list. The list can also be used to look up the allowed vertex types in case  $|vtypes| > 1$ , refer to the figure.

Edges ( $E_Q$ ) are represented by line segments drawn between vertices. The edges’ IDs ( $eids$ ) are placed onto the lines and their captions ( $ecaptions$ ) onto their side. Edges look different depending on their direction settings  $dir$  (cf. Fig. 5). While edges with  $maxl = 1$  are solid, edges with  $maxl > 1$  are dashed. In the latter case,  $minl$  and  $maxl$  are shown on the side of the edge.

All further properties of edges are set using lists that contain the currently supported property values (e.g., the vertex types of the currently supported modeling languages) or using free-text forms (cf. Fig. 6). In the figure, we show such selection lists for  $etypesr$  with  $maxl = 1$ ,  $etypesr$  and  $etypesf$  with  $maxl > 1$ ,  $vtypesr$ ,  $vtypesf$ , and  $\theta$ . The  $minvo$ ,  $maxvo$ ,  $mineo$ , and  $maxeo$  parameters are adjusted using forms. For global rules, we use forms for  $gmaxvo$  and  $gmaxeo$  and a formula editor for the rule equations  $R$ .

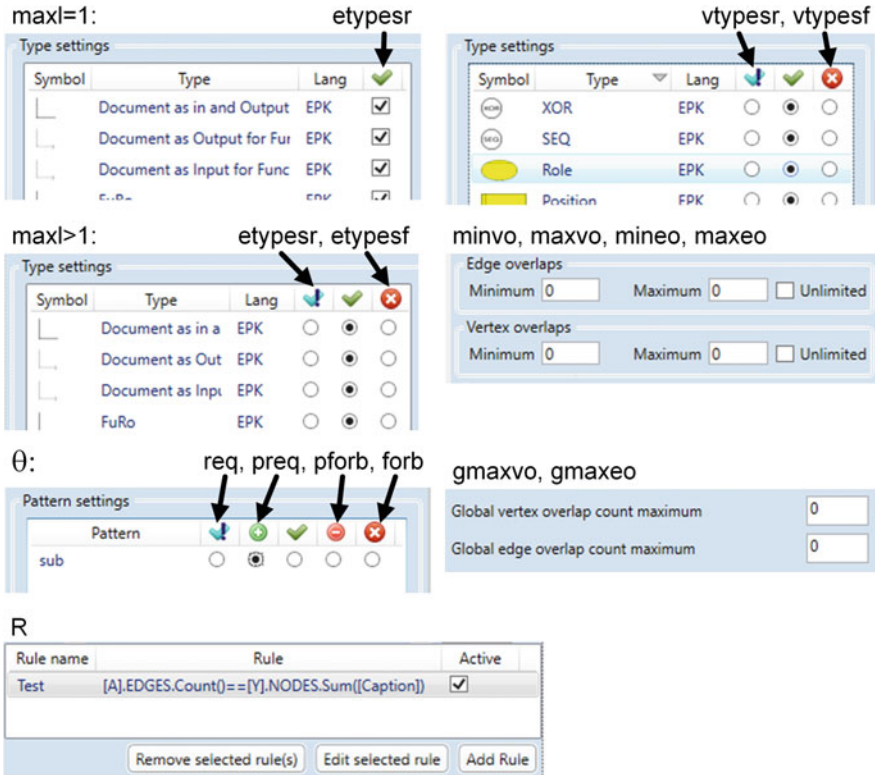


Fig. 6 DMQL concrete syntax: Edge settings and global rules

### 4.3 Semantics

Due to space restrictions, we abstain from defining DMQL’s semantics here and refer the reader to previous work [8]. We have already commented on the impact of each component of a query onto the query results in Sect. 4.1. The corresponding matching algorithm is similar to the brute-force subgraph homeomorphism algorithm [20] implemented using depth-first search. It is extended by several checks that examine whether the vertex, edge, and path mapping candidates in the model graph meet the requirements of the query vertex and edge properties (i.e.,  $P_V$  and  $P_E$ ), which makes it possible to exclude candidates early and this way keep the runtime relatively fast (cf. Sect. 5.1).

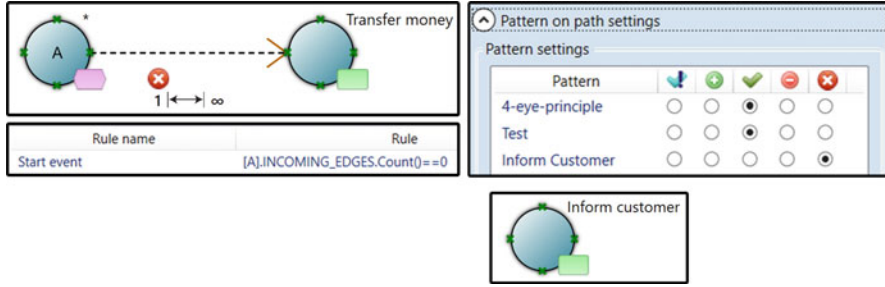


Fig. 7 An exemplary DMQL query

### 4.4 Query Example

Figure 7 shows an example DMQL query tailored for EPC models. It represents a violation of a compliance rule used in credit application processes. The rule requires that a customer has to be informed about all aspects of a granted credit *before* the money can be transferred. In this example, we assume that the models are terminologically standardized, so we do not have to cope with name clashes (e.g., we search for “check bill” but the modeler used “invoice auditing”) as there are several approaches that already solve this problem (e.g., [17]). A violation of the compliance rule means that we find an EPC function somewhere in the model performing the money transfer, but the customer was *not* informed before. In order to identify such violations, we formulate a query that searches for a path from any start event of the process model to the transfer function with *no* function on the path that informs the customer.

The query is assembled as follows: we define a vertex *A* allowing only events as types ( $vtypes = \{event\}$ ), however, arbitrary captions ( $vcaption = "*"$ ). The vertex connects to another vertex via a path of arbitrary length ( $minl = 1$ ,  $maxl = -1$ ), where the direction points to the second vertex ( $dir = \{org\}$ ). The second vertex must be a function ( $vtypes = \{function\}$ ), and its caption should be  $vcaption = \text{“Transfer money”}$ . As we do not access the second vertex in any global rule, we do not have to define any ID for it. To assure that the returned paths start at a start event of the process model, we define a global rule  $[A].INCOMING\_EDGES.Count() == 0$  that requires vertex *A* to have no incoming edges (i.e., to be a start event). The path has a further property that forbids any function on it carrying the caption “Inform customer”. The property is realized through a forbidden sub-pattern, i.e., occurrences of a sub-query that are not allowed to be part of the path. This sub-pattern is named “Inform customer” ( $\theta = \{(\text{“Inform Customer”}, forb)\}$ ; see selection list on the right-hand side of the figure, and note that the selection list lists all available queries of  $\mathcal{Q}$ , which is why we also find other entries in the list, which are not selected as *req*, *preq*, *forb*, or *pforb* for the current query). It consists of a single function with the caption “Inform customer” (see the lower right box in the figure).

## 4.5 DMQL 2.0

The reason why we decided to develop a new version of DMQL was that both GMQL and DMQL were sometimes criticized due to their inability to express queries requiring universal and negation properties. In other words, except the feature that GMQL and DMQL can express that certain elements should not be part of a path, they cannot express that certain elements should **not** be part of a query occurrence in general. In turn, they also cannot express that properties of a model must **always** be true (e.g., if there is activity A, it must **always** be followed by activity B). As such features are common in some related query languages (e.g., those based on Computation Tree Logic [CTL; [7]]), we decided to include such features in DMQL, too. To cover the **always** and **not** requirements, we introduce the following new concepts in DMQL: *forbidden vertices*, *forbidden edges*, and *forbidden paths*.

These three concepts cover situations that require negation, that is, cases where certain elements should **not** be part of a query occurrence. In order to also cover universal statements, we make use of anti-patterns using the “forbidden” concepts. For instance, if it is required that specific activities must always be executed by a specific person, one can search for activities that are connected to a forbidden vertex representing that person. If we get a match, then we have found a violation.

In particular, the extensions are as follows. Firstly, we extend the definition of the properties of query vertices  $P_V$  and query edges  $P_E$ , such that  $P_V \subseteq VID \times VCAPTION \times VTYPES \times FORBV$  and  $P_E \subseteq EID \times ECAPTION \times DIR \times MINL \times MAXL \times MINVO \times MAXVO \times MINEO \times MAXEO \times VTYPESR \times VTYPESF \times ETYPESR \times ETYPESF \times \Theta \times FORBE$ , where  $FORBV$  and  $FORBE$  are sets of Boolean values. When  $forbv = TRUE$  (respectively,  $forbe = TRUE$ ) the vertex (respectively, the edge) becomes forbidden. This means that a query occurrence will only be returned if the forbidden vertex (respectively, the edge) with all its other properties is **not** part of the occurrence. For the forbidden elements, all properties take effect, that is, only those elements are forbidden that match the properties. For instance, if we search for vertices with  $vtypes = \{function\}$  connected to other vertices with  $vtypes = \{document\}$ , where the latter are forbidden, then DMQL would only return functions that are **not** connected to documents. Forbidden edges and paths are handled analogously. If a forbidden vertex is connected to the rest of the query, then its connection (i.e., the query edge) is automatically forbidden, too. Queries may contain multiple forbidden elements. In such a case, only if **all** forbidden elements are found in a prospective query match, the match is excluded from the return set. Forbidden vertices and edges are colored red and tagged with a small “X” in DMQL queries.

Consider the EPC query example in Fig. 8. The query searches for functions that may be connected via a directed path of arbitrary length. Matches are only returned if both of the functions are **not** connected to organizational units (denoted by yellow ovals on the lower right of the vertices) as the organizational units are forbidden.



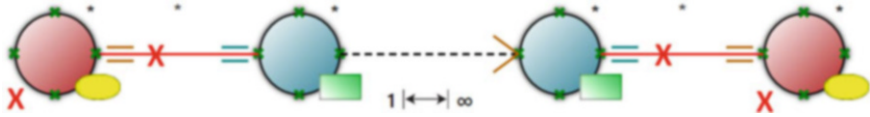


Fig. 8 Example query with forbidden vertices

Symbol	Type	Lang			
	Start Event	BPMN	<input type="radio"/>	<input checked="" type="radio"/>	<input type="radio"/>
	Data Object	BPMN	<input checked="" type="radio"/>	<input type="radio"/>	<input type="radio"/>
	Complex Gateway	BPMN	<input type="radio"/>	<input checked="" type="radio"/>	<input type="radio"/>

Allowed directions

Direction 1

Direction 2

Undirected

---

Set as forbidden

Consider this edge as a path

Path length

Minimum  Maximum   Unlimited

Fig. 9 Example query with a forbidden path edge

Another example shows a BPMN query (cf. Fig. 9). The query searches for subsequent tasks that may follow each other across a path of arbitrary length. These tasks are required to not communicate using data objects. The query realizes this requirement with an additional directed path between the tasks, which is set “forbidden” (see “forbidden” tag in the path properties menu on the right-hand side of the figure; the forbidden path is the lower one), where the path is required to contain at least one data object (see vertex-on-path settings in the selection list on the lower left side of the figure). So, if there is such a path between the tasks, they are not returned as a match.

## 5 Evaluation

The evaluation of GMQL and DMQL is manifold. First, the runtime complexity of a query language’s matching algorithm gives first hints of its applicability. Second, as the runtime complexity is a theoretical construct that does not necessarily consider the actual performance of an algorithm when applying it in practice, an empirical assessment of its performance is also of interest. Third, a query language can be evaluated against its capabilities to express interesting queries, the effort it takes to formalize queries, and the commercial success, that is, how a company can benefit from using the query language. Next, we evaluate these aspects for GMQL and DMQL.

## 5.1 Runtime Complexity

The runtime complexity of GMQL strongly depends on the functions and operators used in the queries. It reaches from  $O(|Z|)$  for *ElementsOfType* to  $O(|Z|!)$  for path functions. The complexity of *AdjacentSuccessors*, for instance, is  $O(|Z|^2 \cdot \textit{degree})$ , where *degree* is the maximum vertex degree of the input model [10].

DMQL uses an extension of a subgraph homeomorphism algorithm that exploits the attributes of the vertices and edges of a query. This way possible matching candidates can be excluded early, and the average runtime complexity decreases. In the worst case, that is, if we define a pattern in which we leave every property open (i.e., we allow any vertex and edge type and caption and any path length), the query turns into a subgraph homeomorphism problem, the complexity of which is superexponential [20].

Such a complexity prohibits any practical use prima facie. However, conceptual models are commonly sparse [22], which influences the runtime positively.

## 5.2 Performance

GMQL and DMQL were implemented as plug-ins for the meta-modeling tool named  $[\varepsilon m]$ , which we have developed over the years.<sup>5</sup> It comes with a meta-modeling environment, where the user can specify the abstract and concrete syntax of modeling languages. Subsequently, the languages can be used to create conceptual models. The users of  $[\varepsilon m]$  are guided to create syntactically correct models.

Both GMQL and DMQL plug-ins provide a query editor and a mapping algorithm, which executes the queries and shows their results by highlighting the query occurrences in the models. Riehle et al. [26] created a video, which shows the whole workflow of specifying a language using  $[\varepsilon m]$ 's meta-model editor, creating models, and running queries with DMQL. Furthermore, a detailed video about the DMQL analysis plug-in is reported in [25].

The performance of the mapping algorithms was improved several times, and we have conducted multiple runtime measurements using real-world model repositories and challenging queries [10, 12]. An experiment consisted of 53 EPCs and 46 Entity-Relationship Models (ERMs [6]) containing from 15 to 264 model elements and 10 different queries of low complexity (i.e., containing few functions of low runtime complexity) to high complexity (i.e., containing several functions of high runtime complexity) for each model. In total, we executed  $10 \cdot 53 + 10 \cdot 46 = 990$  queries. We observed runtimes from about 70 microseconds to about 0.9 s (time for one query execution including the return of all matches of the query). In particular, the queries took 633 microseconds on average for the EPC models

---

<sup>5</sup> <https://em.uni-muenster.de>.

(standard deviation: 3.55 milliseconds) and 10 milliseconds on average for the ERMs (standard deviation: 45.6 milliseconds). The comparatively high deviations result from few queries showing long runtimes for large models, whereas most of the queries showed short runtimes. We argue that because the size of the models used for the experiment was common (i.e., the number of elements of the models was in a range we often see in process models used in practice), and the high-complexity queries were challenging ones, the results of the experiments are satisfactory. DMQL showed a similar performance; however, it was slightly slower for the most complex query [8].

### 5.3 Utility

We define the utility of a query language as its ability to provide useful query results when applied to real-world process models with queries that arise from real-world problems. To evaluate the utility of GMQL and DMQL (version 1.0), we applied them in an experiment for concrete purposes in the field of business process management, namely both *weakness detection* (e.g., [1, 9]) and *business process compliance management* (e.g., [2, 4, 18]). We understand as weakness a process part that hinders the process execution or that has potential to be shaped more efficiently. A compliance violation is a process part which does not obey preset (legal) policies. Once possible weaknesses or compliance rules are known, they can be transformed into queries and searched in process models. There are several works on process model weaknesses [1, 5, 28, 33, 34], which we reviewed in order to identify weakness descriptions that can be formulated as queries for the evaluation. In addition, we analyzed 2000 process models of a public administration manually to derive further weaknesses [9]. In sum, we identified 280 process weaknesses and consolidated them into 111 weakness types. We further divided them into seven weakness categories, which we introduce in the following. The first category named *Modeling Error* is concerned with general modeling and syntax errors, for instance, a decision without a subsequent branch in the model. The second category is *Process Flow*, which contains weaknesses that may hinder fluent process execution such as manual editing after copying a document or redundant activities. The *Information Handling* category deals with inappropriate use of information (e.g., data is digitized twice). The *Technology Switch* category addresses inefficient use of technology (e.g., a re-digitization of a document after it is printed). The *Automation* category is concerned with manual tasks such as calculations that could be automated. The *Environment* and *Organization* categories deal with the organization of people, competencies, and communication. The former is directed at the external environment of a company (e.g., multiple contact persons for one customer), and the latter takes the internal view (e.g., “ping-pong” responsibilities). Figure 10 shows an overview of the derived categories including some examples.

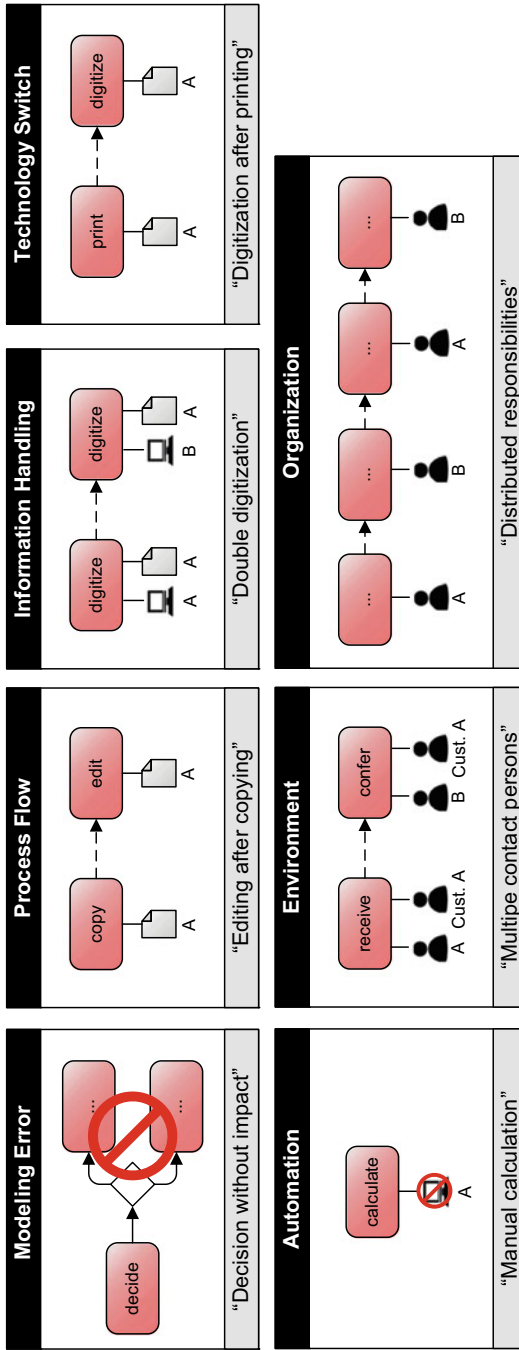


Fig. 10 Weakness categories

We formalized the identified weakness categories in queries and analyzed 85 process models of four companies from the domains of retail, logistics, consulting, and supply. They contained 1995 process activities and 5070 activity attributes such as organizational units and documents [9]. The language of the models was *icebricks* [3], which exists in different versions for different industries and comes with corresponding glossaries that have to be used mandatorily when specifying activity names. A nice side effect was that we did not have to cope with name clashes, as the natural language words used in the queries could be taken from the glossaries.

We analyzed different aspects of utility in the experiment, namely, the amount of issues that can be formalized with GMQL and DMQL, the time it takes to formalize a query, query runtimes, and the quality of the query results.

It was possible to formalize 84% of the derived 111 weaknesses. The remaining 16% could not be formalized; however, this was mainly due to limitations of the modeling language the models of the study were defined in (e.g., there was no object type for IT systems). A minor portion of weaknesses that could not be formalized was due to weaknesses that required negation or universal features that were not included in DMQL 1.0 (cf. Sect. 4.5) and not included in GMQL.

For the DMQL queries, the average formalization time for a single query was 10:18 min. For the same GMQL queries, the formalization took 17:24 min on average. The values include the time needed for the query creation from scratch to the final query without any further need for adaptation (i.e., the queries were correct). We measured the times manually during the creation of the queries. The queries were created by graduate students, who were familiar with the query languages. The final correctness check of the queries was done by the supervisors of the students. In the case of shortcomings, the query had to be revised by the students, and the additional editing time was recorded. We have to note that most of the time was needed for specifying the caption match terms, because the term glossaries contained synonyms.

A single search run (one model, one query, all matches) took 3:45 min on average, even though this strongly depended on the model size. While about 99% of the search runs (nearly 9000 search runs) could be executed in only 27 s on average, the remaining 1% (less than 80 search runs) needed about 4:45 h to execute. These long runtimes were mainly observed in 12 out of the 85 models. In particular, queries in these models took up 94.4% of the total runtime. Correspondingly, Fig. 11 shows that a few queries exhibit a significantly higher search duration than the majority of the search runs. In the figure, only every second query and every fifth process model are explicitly annotated to the axes for reasons of readability. The long runtimes were mainly for queries that searched for unrestricted paths within models. Furthermore, models that provided a high number of path possibilities (several branches and many tasks) induced some of these longer runtimes. The significantly longer runtimes of this experiment compared to those reported in Sect. 5.2 are due

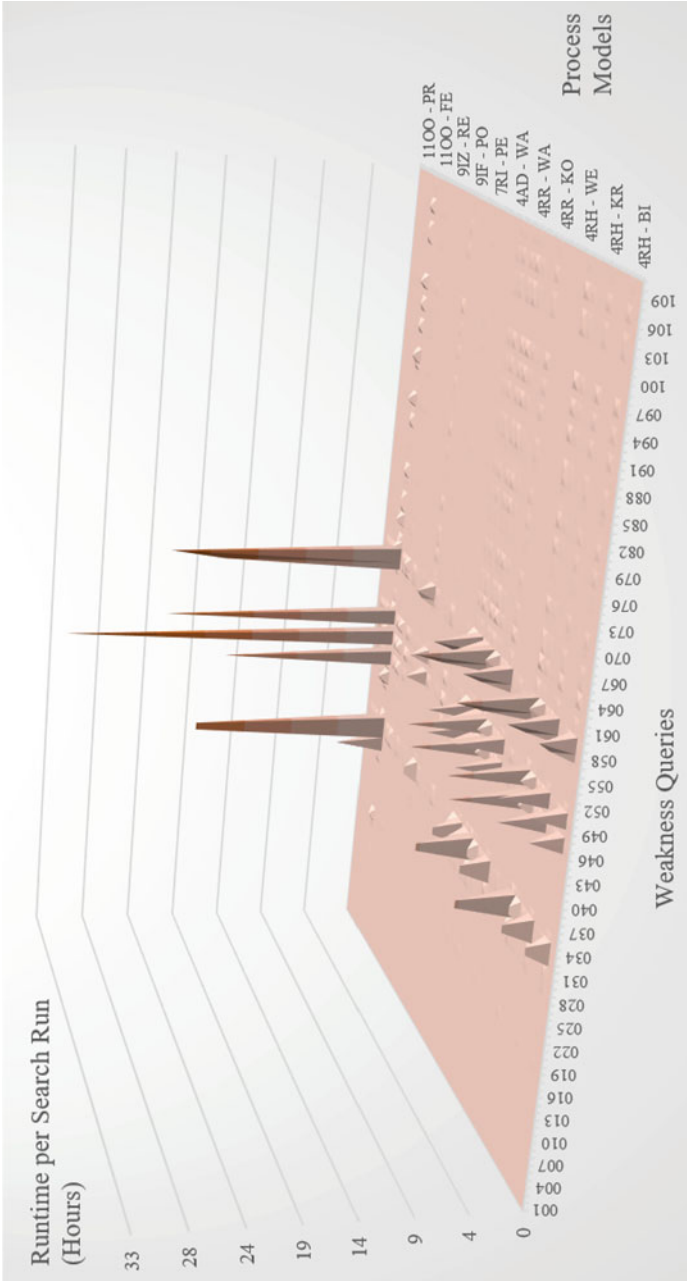


Fig. 11 Runtimes per search run

to significantly larger models examined in the experiment (about five times larger on average).

The searches returned 8071 matches in the process models, of which 998 results were identified as potential weaknesses. This result seems to be fair. However, the results contained 5,364 matches that were duplicates. Duplicates may occur, for instance, when a query contains undirected paths, the start vertices of which have the same properties as the end vertices (because then the path can be “found” from both directions). This insight provided us with valuable information for the further development of the DMQL mapping algorithm, that is, to discard duplicates. As a result, about 12% of the matches were identified as potential weaknesses (37% after removing the duplicates). The potential weaknesses mostly stem from the *Modeling Error* and *Automation* categories (about 85%). Especially, the weakness query “Missing activity attributes” of the *Modeling Error* category, which indicates necessary but missing attributes like organizational units, exhibited a high number of results classified as potential weakness. This is not surprising as we often find such shortcomings in process models, and sometimes such shortcomings are accepted or even regarded as harmless, so, in turn, not regarded as actual weaknesses. Besides possible modeling errors, the *Automation* category mainly concerned with tasks like manual calculations, yielded about 32% of the potential weaknesses. Even though many potential weaknesses stem from only a few queries, approximately one-third of the queries returned at least one match, and we found matches in nearly all models (95%). We further argue that approximately 12 potential weaknesses per process model (998 potential weaknesses in 85 process models) is quite a high number.

In a similar way, GMQL and DMQL can support business process compliance management, which is a well-elaborated topic in the literature, and the need for automatic support is obvious [2, 13, 14, 16, 27, 29, 31]. Thus, we applied GMQL [4] and DMQL [18] in this field. Query-based compliance checking works in the same way as weakness detection. The derivation of compliance rules is different, however, as these are often pre-formulated through regulations or laws. In our work [4], we defined 13 compliance queries textually and in GMQL notation. We later extended the collection to 29 queries and translated them into the DMQL notation. The detailed derivation procedure for DMQL queries (how to turn law texts into queries) is explained in [18]. We further queried process models of an IT service provider for banks with 25 process models containing 3427 activities and 17,616 attributes. The models represented the processes of the software the provider was producing and hosting for affiliated banks. In total, we detected 49 potential compliance violations that finally led the IT service provider to adapt her banking software running in more than 4000 affiliated banks [4].<sup>6</sup>

---

<sup>6</sup> Note that we cannot publish the name of the IT service provider due to a nondisclosure agreement.

The results are promising and suggest that GMQL and DMQL can be applied successfully in real-world scenarios.

## 6 GMQL, DMQL, and the Process Querying Framework

When relating GMQL and DMQL 2.0 (in this section referred to as G/DMQL) to the Process Querying Framework [23], we come to the following result: in the category *behavior model*, G/DMQL address *process models* only. Queries have to be *formalized* by persons or can be taken from existing query catalogues.<sup>7</sup> Depending on the used language, the query is either a *textual* or *visual* formalism. G/DMQL support the *modeling* component of the framework, i.e., they can be used to query any kind of process model. Although G/DMQL are not restrictive in what kinds of process model repositories should be queried, we regard it most reasonable to assign them to repositories containing *semi-formal models* as these are the kinds of models they operate on. G/DMQL's *query intention* is *read (retrieve)* and *read (project)*, as they answer both the questions whether a model fulfills a query and return corresponding details about the model (i.e., they pinpoint each query occurrence by highlighting it in the queried model). G/DMQL's *querying technique* is *graph matching* as both are based on adapted forms of subgraph homeomorphism.

G/DMQL operate over *semi-formal* models, i.e., over the formalized graph structure of the models, but not on their execution semantics like, for instance, token concepts or traces. G/DMQL perform *no kind of data preprocessing*, meaning that they operate on the sheer data of the models contained in a database, where the structure of such a database should preferably have a schema similar to the model shown in Fig. 3. No *filtering* takes place with the exception that G/DMQL queries operate over the subsections of the model repository that the user has selected. G/DMQL do not provide a dedicated *optimizing* mechanism. However, GMQL implements *caching*. It caches sub-queries that occur multiple times in a query structure [12]. After a query is executed, G/DMQL present *visualizations* to the user (cf. Sects. 3 and 4), and the results can be browsed to *inspect* where in the queried model the query occurrences can be found. There exists an option to *animate* the execution of a query, so the user can trace intermediate results of the matching algorithm, which are highlighted in sequence in the queried model.

Figure 12 shows the schematic view of the Process Querying Framework taken from [23], where we marked the components addressed by G/DMQL in black. Components that are only partly addressed by G/DMQL are marked in gray.

<sup>7</sup> See, e.g., <http://www.conceptual-modeling.org>.



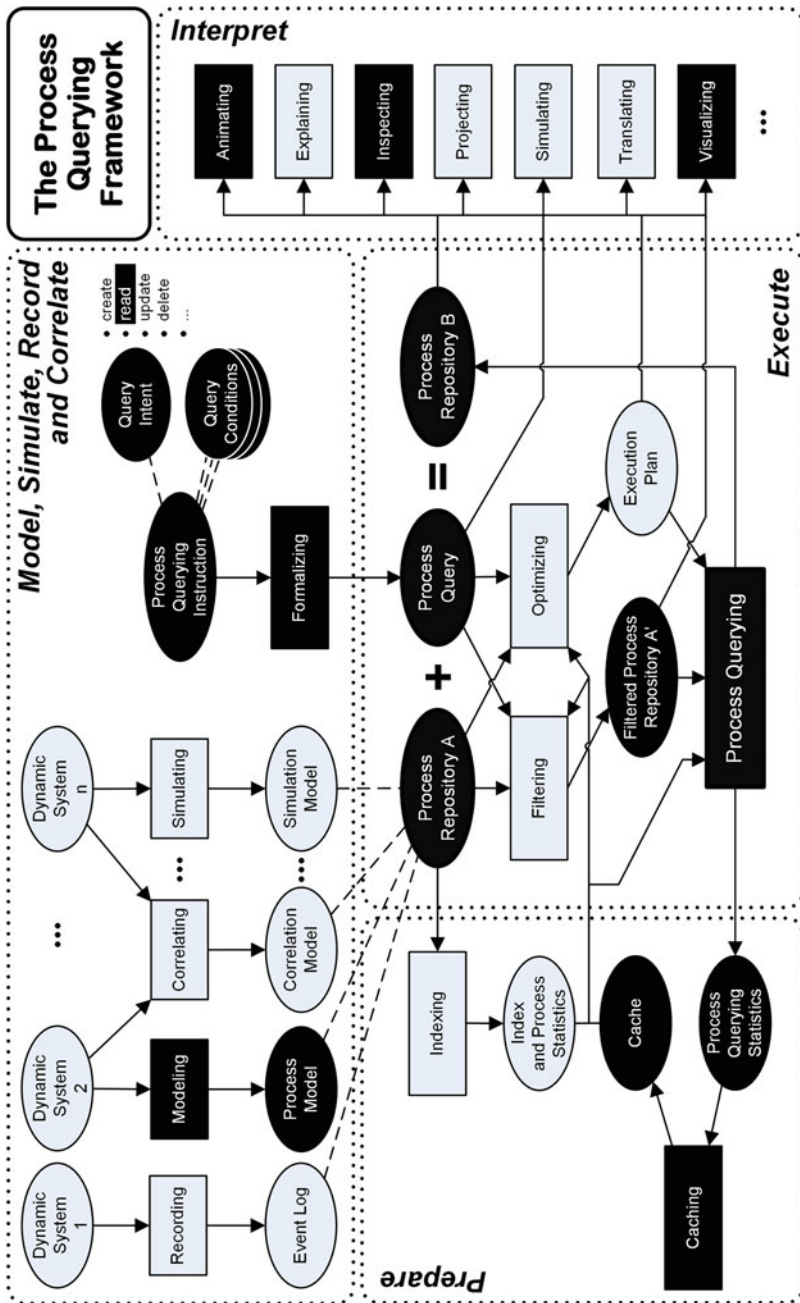


Fig. 12 Framework components addressed by G/DMQL [23]

## 7 Conclusion

In this chapter, we reported on the development and evaluation of the (process) model query languages GMQL and DMQL. The purpose was not to introduce new query languages—we have done that before—but rather to outline how GMQL and DMQL work and through how many and which iterations the development traversed. We have seen that evolving requirements, criticism, hints, and evaluation both from scholars and professionals have resulted in a research process of multiple iterations. With the latest adaptation of DMQL, we addressed the problem of previously missing negation and universal operators. For the future, we plan to evaluate DMQL 2.0 in further field experiments that might result in further adaptations. Through being able to directly compare several query languages with the help of this book, we also aim to promote discussions on GMQL, DMQL, and their related approaches.

## References

1. Becker, J., Bergener, P., Räckers, M., Weiß, B., Winkelmann, A.: Pattern-based semi-automatic analysis of weaknesses in semantic business process models in the banking sector. In: Proceedings of the 18th European Conference on Information Systems (ECIS), Pretoria, South Africa (2010)
2. Becker, J., Delfmann, P., Eggert, M., Schwittay, S.: Generalizability and applicability of model-based business process compliance-checking approaches - A state-of-the-art analysis and research roadmap. *BuR - Bus. Res.* **5**(2), 221–247 (2012)
3. Becker, J., Clever, N., Holler, J., Shitkova, M.: Icebricks. In: vom Brocke, J., Hekkala, R., Ram, S., Rossi, M. (eds.) *Design Science at the Intersection of Physical and Virtual Design*, pp. 394–399. Springer Berlin Heidelberg, Berlin, Heidelberg (2013)
4. Becker, J., Delfmann, P., Dietrich, H.A., Steinhorst, M., Eggert, M.: Business process compliance checking - Applying and evaluating a generic pattern matching approach for conceptual models in the financial sector. *Inf. Syst. Front.* **18**(2), 359–405 (2016)
5. Bergener, P., Delfmann, P., Weiss, B., Winkelmann, A.: Detecting potential weaknesses in business processes. *Bus. Process Manag. J.* **21**(1), 25–54 (2015)
6. Chen, P.P.S.: The entity-relationship model—toward a unified view of data. *ACM Trans. Database Syst.* **1**(1), 1–36 (1976)
7. Clarke, E.M., Grumberg, O., Peled, D.: *Model Checking*. MIT press (1999)
8. Delfmann, P., Breuker, D., Matzner, M., Becker, J.: Supporting information systems analysis through conceptual model query – The diagramed model query language (DMQL). *Commun. Assoc. Inf. Syst.* **37**(24) (2015)
9. Delfmann, P., Höhenberger, S.: Supporting business process improvement through business process weakness pattern collections. In: Proceedings of the 12. Internationale Tagung Wirtschaftsinformatik, pp. 378–392. Osnabrück, Germany (2015)
10. Delfmann, P., Steinhorst, M., Dietrich, H.A., Becker, J.: The generic model query language GMQL - conceptual specification, implementation, and runtime evaluation. *Information Systems* **47**, 129–177 (2015)
11. Diestel, R.: *Graphentheorie*. Springer, Berlin/Heidelberg, Germany (2010)
12. Dietrich, H.A., Steinhorst, M., Becker, J., Delfmann, P.: Fast pattern matching in conceptual models-evaluating and extending a generic approach. In: EMISA, pp. 79–92. Citeseer (2011)

13. El Kharbili, M., de Medeiros, A., Stein, S., van der Aalst, W.: Business process compliance checking: Current state and future challenges. In: Proceedings of the Conference Modellierung betrieblicher Informationssysteme (MobIS). Saarbrücken, Germany (2008)
14. Elgammal, A., Turetken, O., van den Heuvel, W.J., Papazoglou, M.: Formalizing and applying compliance patterns for business process compliance. *Softw. Syst. Model.* **15**(1), 119–146 (2016)
15. Fahland, D., Favre, C., Jobstmann, B., Koehler, J., Lohmann, N., Völzer, H., Wolf, K.: Instantaneous soundness checking of industrial business process models. In: Dayal, U., Eder, J., Koehler, J., Reijers, H.A. (eds.) Proceedings of the 7th International Conference on Business Process Management (BPM), pp. 278–293. Ulm, Germany (2009)
16. Ghose, A., Koliadis, G.: Auditing business process compliance. In: Proceedings of the International Conference on Service-Oriented Computing (ICSOC), pp. 169–180. Vienna, Austria (2007)
17. Havel, J.M., Steinhorst, M., Dietrich, H.A., Delfmann, P.: Supporting terminological standardization in conceptual models – a plugin for a meta-modelling tool. In: Proceedings of the 22nd European Conference on Information Systems (ECIS 2014) (2014)
18. Höhenberger, S., Riehle, D.M., Delfmann, P.: From legislation to potential compliance violations in business processes — Simplicity matter. In: European Conference on Information Systems, ECIS '16, pp. 1–15. Istanbul, Turkey (2016)
19. La Rosa, M., Dumas, M., Uba, R., Dijkman, R.: Business process model merging. *ACM Trans. Softw. Eng. Methodol.* **22**(2), 1–42 (2013)
20. Lingas, A., Wahlen, M.: An exact algorithm for subgraph homeomorphism. *J. Discrete Algorithms* **7**(4), 464–468 (2009). <https://doi.org/10.1016/j.jda.2008.10.003>. <http://www.sciencedirect.com/science/article/pii/S1570866708000968>
21. Peffers, K., Tuunanen, T., Rothenberger, M.A., Chatterjee, S.: A design science research methodology for information systems research. *J. Manag. Inf. Syst.* **24**(3), 45–77 (2007)
22. Pflanzl, N., Breuker, D., Dietrich, H., Steinhorst, M., Shitkova, M., Becker, J., Delfmann, P.: A framework for fast graph-based pattern matching in conceptual models. In: IEEE 15th Conference on Business Informatics, CBI 2013, Vienna, Austria, July 15–18, 2013, pp. 250–257 (2013). <https://doi.org/10.1109/CBI.2013.42>
23. Polyvyanyy, A., Ouyang, C., Barros, A., van der Aalst, W.M.: Process querying: Enabling business intelligence through query-based process analytics. *Decis. Support Syst.* **100**, 41–56 (2017). <https://doi.org/10.1016/j.dss.2017.04.011>. <http://www.sciencedirect.com/science/article/pii/S0167923617300787>. Smart Business Process Management
24. Reisig, W.: Petri nets and algebraic specifications. *Theor. Comput. Sci.* **80**(1), 1–34 (1991)
25. Riehle, D.M., Höhenberger, S., Cording, R., Delfmann, P.: Live query — visualized process analysis. In: Leimeister, J.M., Brenner, W. (eds.) Proceedings der 13. Internationalen Tagung Wirtschaftsinformatik (WI 2017), pp. 1295–1298. St. Gallen, Switzerland (2017)
26. Riehle, D.M., Höhenberger, S., Brunk, J., Delfmann, P., Becker, J.: [εm] — process analysis using a meta modeling tool. In: Cabanillas, C., España, S., Farshidi, S. (eds.) Proceedings of the ER Forum 2017 and the ER 2017 Demo Track, CEUR Workshop Proceedings, vol. 1979, pp. 334–337. Valencia, Spain (2017)
27. Rinderle-Ma, S., Ly, L.T., Dadam, P.: Business process compliance (Aktuelles Schlagwort). *EMISA Forum* **28**(2), 24–29 (2008)
28. Russell, N., ter Hofstede, A.H.M., van der Aalst, W.M.P., Mulyar, N.: Workflow control-flow patterns: A revised view. *BPM Center Rep.* **2**, 6–22 (2006)
29. Sadiq, S., Governatori, G., Namiri, K.: Modeling control objectives for business process compliance. In: Proceedings of the 5th International Conference on Business Process Management (BPM), pp. 149–164. Brisbane, Australia (2007)
30. Scheer, A.W.: ARIS—Business Process Modeling. Springer Science & Business Media (2012)
31. Schumm, D., Turetken, O., Kokash, N., Elgammal, A., Leymann, F., van den Heuvel, W.J.: Business process compliance through reusable units of compliant processes. In: Daniel, F., Facca, F.M. (eds.) Current Trends in Web Engineering, pp. 325–337. Springer, Berlin/Heidelberg, Germany (2010)

32. Steinhorst, M., Delfmann, P., Becker, J.: vGMQL - introducing a visual notation for the generic model query language GMQL. In: PoEM (2013)
33. Van der Aalst, W.M., Ter Hofstede, A.H., Kiepuszewski, B., Barros, A.P.: Workflow patterns. *Distrib. Parallel Databases* **14**(1), 5–51 (2003)
34. Winkelmann, A., Weiß, B.: Automatic identification of structural process weaknesses in flow chart diagrams. *Bus. Process Manag. J.* **17**(5), 787–807 (2011)

# VM\*: A Family of Visual Model Manipulation Languages



Harald Störrle and Vlad Acrețoiaie

**Abstract** CONTEXT: Practical facilities for querying, constraining, and transforming models (“model management”) can significantly improve the utility of models, and modeling. Many approaches to model management, however, are very restricted, thus diminishing their utility: they support only few use cases, model types or languages, or burden users to learn complex concepts, notations, and tools. GOAL: We envision model management as a commodity, available with little effort to every modeler, and applicable to a wide range of use cases, modeling environments, and notations. We aim to achieve this by reusing the notation for modeling as a notation for expressing queries, constraints, and transformations.

METHOD: We present the VM\* family of languages for model management. In support of our claim that VM\* lives up to our vision we provide as evidence a string of conceptual explorations, prototype implementations, and empirical evaluations carried out over the previous twelve years.

RESULTS: VM\* is viable for many modeling languages, use cases, and tools. Experimental comparison of VM\* with several other model querying languages has demonstrated that VM\* is an improvement in terms of understandability. On the downside, VM\* has limits regarding its expressiveness and computational complexity.

CONCLUSIONS: We conclude that VM\* largely lives up to its claim, although the final proof would require a commercial implementation, and a large-scale industrial application study, both of which are beyond our reach at this point.

---

H. Störrle (✉)  
QAware GmbH , München, Germany  
[hstorle@acm.org](mailto:hstorle@acm.org)  
e-mail: [harald.stoerrle@ifi.lmu.de](mailto:harald.stoerrle@ifi.lmu.de)

V. Acrețoiaie  
FREQUENTIS Romania SRL , Cluj-Napoca, Romania

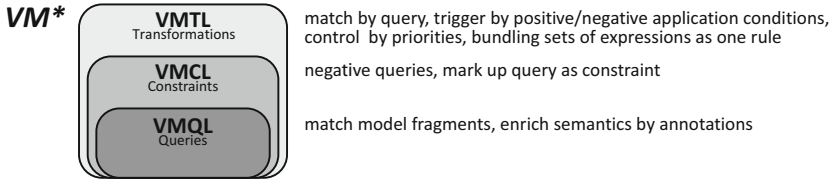
## 1 Introduction

The *Visual Model Manipulation Language* VM\* is a lineage of languages that allows to express queries (VMQL), constraints (VMCL), and transformations (VMTL); refer to Table 1. Figure 1 illustrates the relationship between these three languages. VM\* aspires to be truly useful, addressing the problems faced by the working modeler. Thus, our design goals are usability and learnability, versatility and practical use cases, coverage of all relevant visual modeling notations and compatibility with existing modeling environments. As a consequence, there is a limit to the expressiveness and the scope of application scenarios VM\* addresses. For instance, VM\* is not suited to process ultra large models, or express very complex model transformations. In our experience, those situations are rare.

Starting out as a query language, VM\* has evolved into a full blown model manipulation language that can also handle constraints and transformations. Obvi-

**Table 1** Main publications on VM\* and its precursors, most notably VMQL and VMTL. In column “Intent”, Q, C, and T refer to queries, constraints, and transformations, respectively, whereas bullets indicate the types of instructions addressed in the corresponding publications. In column “Type”, W, C, J, and TR stand for **W**orkshop, **C**onference, **J**ournal paper, and **T**echnical Report, respectively. BSc, MSc, PhD refer to theses of the respective types. References [7, 32, 34] are re-publications, posters, and excerpts

Year	Ref.	Type	Intent			Title (abbreviated)
			Q	C	T	
2005	[24]	TR	•	•		MoMaT: A lightweight platform for MDD
2007	[25]	W	•			A PROLOG approach to representing & querying models
2009	[39]	BSc	•			MQ: A visual query-interface for models
	[26]	W	•			A logical model query-interface
	[27]	C	•			VMQL: A generic visual model query language
2011	[28]	C		•		Expressing model constraints visually with VMQL
2012	[29]	J	•			VMQL: A visual language for ad-hoc model querying
	[3]	MSc	•			An implementation of VMQL
	[5]	W	•			MQ-2: A tool for prolog-based model querying
2013	[30]	W	•			Improving the usability of OCL as an ad-hoc MQ language
	[31]	W	•			MOCQL: A declarative language for ad-hoc model querying
	[35]	W	•			Querying business process models with VMQL
2014	[6]	W	•			Efficient model querying with VMQL
	[8]	W	•	•		Hypersonic: Model analysis and checking in the cloud
2015	[33]	J	•	•		Cost-effective evolution of prototypes: The MACH case study
2016	[10]	J	•	•	•	VMTL: A language for end-user model transformation
	[9]	C	•	•	•	Model transformation for end-user modelers with VMTL
	[4]	PhD	•	•	•	Model manipulation for end-user modelers
	[2]	WIKI	•	•	•	The VM* Wiki



**Fig. 1** VM\* is a family of model manipulation languages that allows to express queries (VMQL), constraints (VMCL), and transformations (VMTL)

ously, there is a natural succession arising out of the symmetry between queries and constraints. Then, transformations are just pairs of an application condition and a consequence, both of which can be expressed like queries. In that sense, there is not just a natural symmetry but also a great practical opportunity by leveraging queries to constraints and transformations. More importantly, however, there is a practical need for constraints and transformations once queries are established, like progressing from text search to search-and-replace. Model querying is useful, but it is incomplete without checks and transformations. From a high-level point of view, we can characterize queries, constraints, and transformations as follows.

**Queries.** Assume that a model  $\mathcal{M}$  is a set of model elements  $ME$ . A Boolean property  $\alpha : \mathcal{P}(ME) \rightarrow bool$  characterizes those parts of  $\mathcal{M}$  that satisfy  $\alpha$ . So, applying  $\alpha$  to all fragments of  $\mathcal{M}$  amounts to querying  $\mathcal{M}$ , and the set of all fragments  $R_\alpha$  that satisfy  $\alpha$  is the result of the query.

**Constraints.** Conversely, the dual of  $R_\alpha$  are those model fragments of  $\mathcal{M}$  that do not satisfy  $\alpha$ . If we formulate  $\alpha$  just so that it yields all *admissible* fragments of  $\mathcal{M}$ , and look at the complement of the result, we have a constraint. So, we need a way of choosing whether we consider  $R_\alpha$  or  $\overline{R_\alpha}$  as the result. Also, we need to express the complement, or, more generally, a form of negation so that constraints may be expressed in a concise way.

**Transformations.** Similarly, model transformations are sets of rules, each of which consists of an application condition (“left-hand side”) and a consequence (“right-hand side”). The application condition is again a query or a constraint, while the right-hand side must express matching and changing, i.e., it is a query with side effects.

The general idea of VM\* is to reuse the syntax of the modeling language at hand (the *host language*) as the syntax of the query, constraint, or transformation language, adding only a small set of textual annotations. Then, a query is matched against model fragments of the host language based on structural similarity. In the process, annotations are evaluated. Any matching fragments are presented as results. Conversely, when evaluating a constraint, fragments not matching the constraint are presented as violations. For transformations, two parts have to be provided per transformation rule, expressing the left-hand side and right-hand side of the rule. Left-hand side expressions are effectively queries and/or constraints. If it is satisfied,

the right-hand side is executed, either replacing or modifying the matched fragment. The unique approach of VM\* provides three main benefits.

**Syntax transparency.** Queries, constraints, and transformations are expressed using the host language, while VM\* only consists of a few *annotations* on models and diagrams. Thus, any modeler is, by definition, already capable of expressing simple queries, constraints, and transformations in VM\*. For more complex expressions, a few new concepts have to be learned.

**Environment transparency.** Grace to syntax transparency, any modeling tool can be used as a front end for VM\*. Therefore, any modeler can, by definition, use the VM\* tool, which is just the editor the modeler uses anyway. Integrating the tool for executing a query or transformation, or checking a constraint, can be seamless.

**Execution transparency.** Unlike other approaches, VM\* is not semantic, but syntactic, that is, it does not consider the meaning of model elements, but the notation alone. VM\* is ignorant to what boxes and lines mean. While this imposes limits on the expressiveness of VM\*, it also avoids semantic problems, makes the language more accessible, and simplifies implementation.

We argue that the restriction in expressiveness is rarely relevant in practice, while universal applicability, learnability, and usability are always a concern. VM\* is applicable for any host language satisfying two conditions.

1. It must have a metamodel. This is trivially the case for any language that is implemented in a tool, in particular, for languages created with metamodeling tools like Adonis [15], EMF [23], or Meta-Edit [36].
2. It must have a way of adding textual comments to model elements, which is true for any modeling tool we have seen in practice.

It cannot be overemphasized that VM\* is completely independent of the semantics of the host language. Conceptually, one may consider a model as a graph with labeled nodes. Unlike the original graph transformation approaches (e.g., [14, 18]), though, this graph is never exposed to the user. This means that VM\* is applicable to many modeling languages, including languages for dynamic models like BPMN, EPCs, Use Case Maps, Simulink, or Role-Activity-Diagrams, as well as languages for static models like ER Diagrams, i\*, KAOS, etc. Thus, VM\* is also applicable to broad-spectrum languages with multiple notations, like UML, SysML, IDEF, or ArchiMate. Similarly, VM\* is applicable to many *Domain Specific Languages* (DSLs). It would be exceedingly difficult to ascertain that VM\* works with *any* visual notation as its host language, but we believe the prerequisites are very modest. We have yet to encounter a notation that cannot serve as a host to VM\*. As a matter of notation, when referring to the instantiation of VM\* for UML, we write VM\*UML, and VM\*BPMN for the instantiation of VM\* for BPMN.

When using the same editor to create source models as well as the queries, constraints, and transformations to be applied to them, it is also irrelevant how models are represented in the editor. In practice, most modeling tools are less than completely compliant to whatever standards they aspire to implement. So, a model query



implementation that relies on standard compliance may be of limited use, or entirely incompatible. This way, many research prototypes are tied to the single modeling environment in which they happen to be implemented. It is hard to overstate the benefit of execution and environment transparency for industrial applications.

## 2 Examples

In this section, we present a high-level process model expressed as a UML Use Case diagram, a low-level process model expressed as a UML Activity Diagram, and another low-level process model expressed in BPMN. Throughout this paper, we denote metamodel concepts and VM\* expressions by typewriter font and CamelCaps, while “elements” from sample models or queries are printed in sans-serif font and enclosed in quotation marks.

### 2.1 High-Level Process Models Expressed as Use Case Diagrams

As a first example, consider Fig. 2. Here, we use UML Use Case Diagrams to model the high-level views of process models, like function trees Value Added Chain Diagrams and process landscapes [13, pp. 239]. The Use Case Diagram at the top left represents the model repository; the other diagrams represent actions on the model: progressing clockwise from the top right, we see an example of a query, a constraint, and a transformation.

The query in Fig. 2 (top right) is a find pattern identified by the looking-glass icon. It matches all its elements against the model repository. For every successful binding of *all* query elements, one solution is generated. The wildcard acts as expected, matching any string. Therefore, the query will yield two results: the use case “request installment loan” and the use case “request revolving loan”. Use cases “specify loan details” and “buy credit insurance” do not match because their names do not match. Use cases “calculate risk” and “request loan” do not match because they are not associated with an actor. Use case “request loan” also does not match because it is abstract, and the use case of the query is not. In this query, only one annotation is required, namely the name pattern of the use case. Since this is a frequent case, we allow using wildcards in names without an explicit annotation. Escaping wildcards symbols allows to use them as proper symbols of names.

The constraint in Fig. 2 (bottom right) is again a find pattern, but this time it has a context annotation which defines the application condition of a constraint. Executing a constraint works just like executing a query: all the elements of the

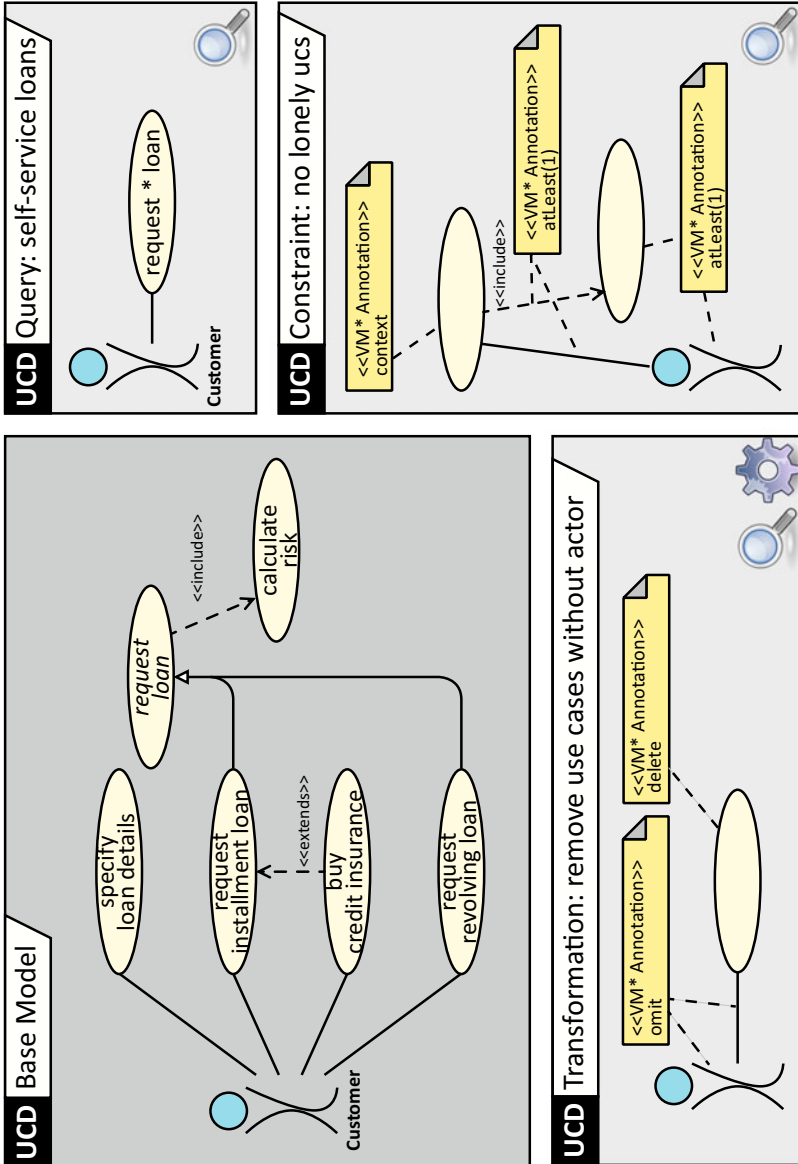


Fig. 2 An introductory example of a model repository (top left), a query, a constraint, and a transformation (clockwise from top right)

constraint are matched against the model repository. However, the results are treated slightly differently. There are three cases:

- If there is no binding for all of the elements of the context, the constraint is not applicable.
- If there is a binding for all context elements, but not for all the other elements, the constraint is violated, and the binding set is returned.
- Finally, if both the context elements and the other elements are matched, the constraint is satisfied.

The following schema summarizes when VM\* constraints are applicable, and when they are satisfied or violated.

Elements matched		Constraint is
Within context	Outside context	
Not all	–	Not applicable
All	None	Violated
All	Some	Violated
All	All	Satisfied

In the interest of compact specification, this rule is modified by `atLeast` annotations. An individual `atLeast` annotation groups together related model elements. The annotation set relates such element groups, specifying how many times such groups must be matched to satisfy the constraint. In the example in Fig. 2 (bottom right), the parameter is 1, meaning that either there is at least one `Includes` relationship or one `association` attached to the “context” use case. The actor and the included use case are required because the UML syntax demands that `Associations` and `Includes` relationships may not be “dangling”. Connected to other model elements as they are in this example, however, they would have to match in order to not violate the constraint. Since the intention is for these elements not to match, we need to add another `atLeast` constraint. This constraint is satisfied for “Base Model”: the first four use cases are associated with “Customer”, “calculate risk” is included in another use case, and “request loan” is abstract, while the context use case in the constraint is not. Hence the constraint in Fig. 2 (bottom right) is satisfied.

Figure 2 (bottom left) shows a simple transformation for cleaning up “orphan” use cases. It consists of a single transformation rule expressed as one diagram defining both the left-hand side and the right-hand side of the rule. The left-hand side is a query for the model elements in the diagram, only that the actor and the association `prevent` matching due to the `omit` annotation. In other words, wherever these elements are present, the rule does not match. The right-hand side consists of the instruction to delete the use case.

In order to distinguish transformations from queries and constraints, yet express their similarity, the transformation is identified by both the looking-glass and the cogwheel icon. Transformations are interpreted similar to queries and constraints:

elements of the base model are matched against the elements of the transformations as explained above. Then, those annotations that indicate updates are triggered, in this case the `delete` clause. In this particular example, there are the following cases:

- If the use case is matched, and there is also an associated actor that matches, the whole transformation fails, and is not executed, because of the `omit` annotation on the actor and its association.
- On the other hand, if the use case is matched, but there is no associated actor, the transformation can be applied, deleting the “orphan” use case.

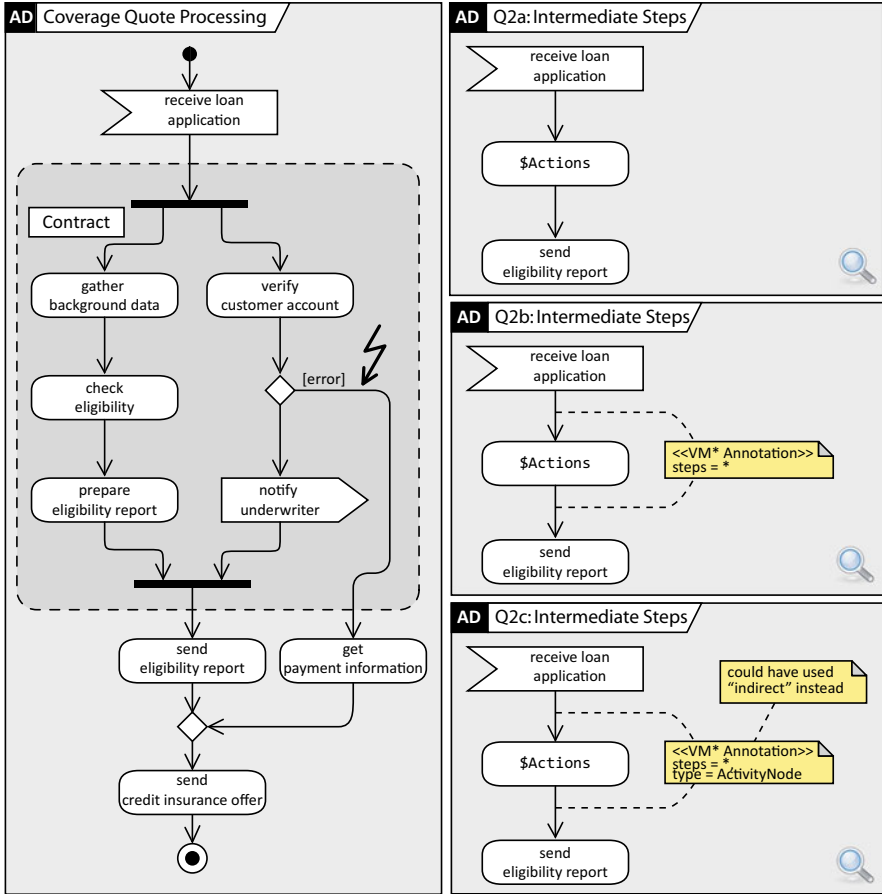
In this example, only “calculate risk” fits into that mold and is deleted. “calculate risk” fits structurally but is abstract.

## 2.2 Low-Level Process Models Expressed as Activity Diagrams

Now consider an example of a process model expressed as a UML Activity Diagram shown in Fig. 3. As before, there is one diagram that we use as the model repository (Fig. 3, left). To illustrate the VM\*UML language capabilities, we use a query that we develop in several steps.

Suppose we want to find out what happens after a loan application is received and before the eligibility report is sent out, i.e., what are the exact steps to determine whether a client receives a loan or not? The general idea is to specify the delimiters “receive loan application” and “send eligibility report”, and find `Actions` between them. A first attempt to express this query may look like Q2a (Fig. 3, top right). Executing this query yields the empty result set, though, as we have specified that there should be a single `Action` that is *directly* connected to both delimiters. In the base model, however, there are *paths* of varying lengths.

In order for this query to find all `Actions` at *any* distance from the delimiters, we have to relax the condition and allow paths of arbitrary lengths instead of directly connected `Flows`. This is achieved by annotating the arcs with `steps = *`, which means “any number of steps” (see query Q2b, at the middle right of Fig. 3). However, this still yields no results. The reason is that the definition of `steps` restricts paths to contain only the kinds of node types adjacent to the `ControlFlow` arc on which it is defined. In this case, there is a plain `Action` (“send eligibility report” and the target node in the middle) and a `ReceiveEventAction` “receive loan application”, but no `ForkNode` or `JoinNode`. However, all the paths in the desired result set do contain fork or join nodes. So, in order to yield the expected result, we need to also relax the types of nodes on paths by using the `type` annotation in the next query (Q2c, at the bottom right of Fig. 3). Going beyond the example, we could be even more relaxed here and allow any type of node by specifying `type_is_any`.

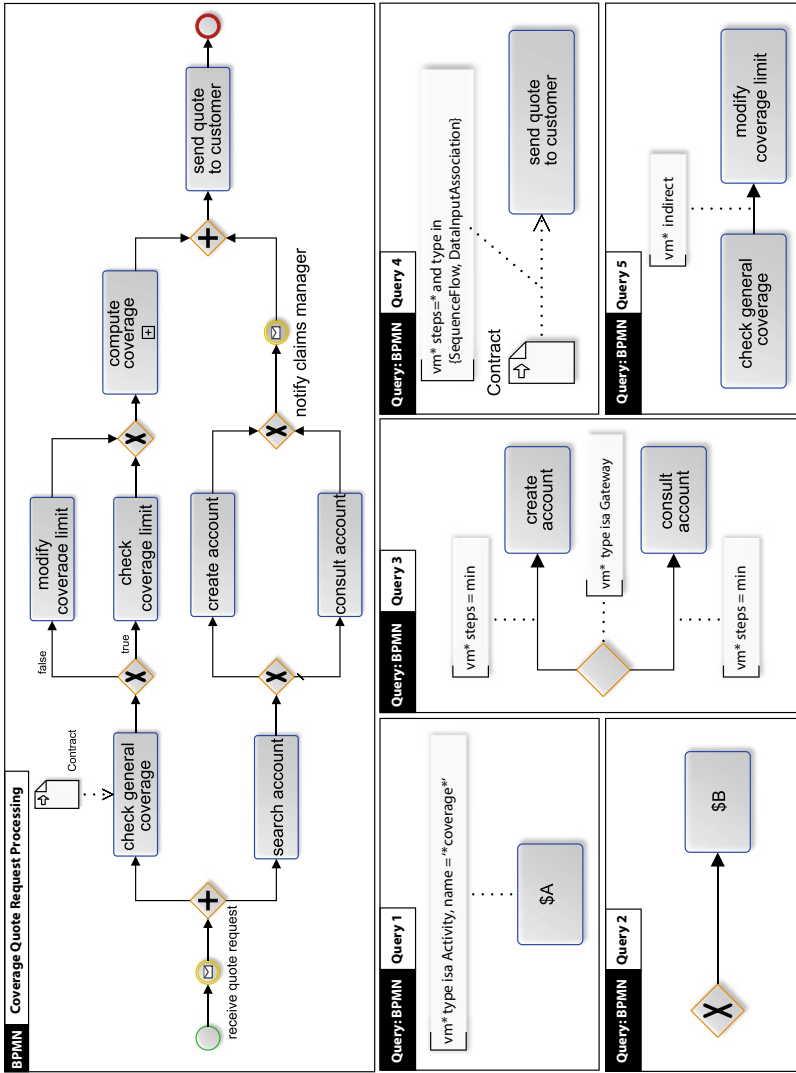


**Fig. 3** A UML Activity Diagram with an exception (left) and related queries (right). Note that in UML, tokens within an InterruptibleActivityRegion (the gray area with a dashed borderline) are discarded upon firing an ExceptionFlow edge (marked with a lightning symbol)

### 2.3 Low-Level Process Models Expressed as BPMN Diagrams

In order to support our claim of general applicability, we now present VM\* queries on *BPMN* (see [35] for more details). The top half of Fig. 4 defines how insurance quote requests are processed by an insurance company, while the bottom half presents five queries.

Suppose a business analyst is interested in finding all activities that deal with insurance coverage. She can succinctly express this request in VM\* using Query 1, consisting of a task named \$A and a comment containing VM\* annotations. The name of the task starts with a “\$” sign, indicating that it is a variable declaration. The name of any matching activity in the source model must be bound to \$A.



**Fig. 4** Handling coverage quote requests in an insurance administration system: Process model expressed in BPMN (top), and five VM\*BPMN queries (bottom)

The text annotation starts with the `vm*` keyword, indicating that it should be interpreted as a VM\* expression. The annotation `type isa Activity` ensures that all of the BPMN activity types and their subclasses are considered, e.g., `Task`, `CallActivity`, `SubProcess`, and `SendTask`. In the example, this annotation enables the query to also return the *compute coverage* collapsed sub-process, which according to the BPMN metamodel is actually not a `Task`. Rather, `Task` and `SubProcess` are both subclasses of the `Activity` abstract meta class, see [19, p. 149]. Finally, the `name = '*coverage*` annotation uses a regular expression to specify that all activities matching the query must contain the string “coverage” in their name. A VM\* implementation parses and evaluates such constraints when computing the result set. In the example, this applies to the first three nodes of the upper branch.

A similar, slightly more complex case is shown in Query 2. Here, the intention is to find all tasks which can only be executed after an `ExclusiveGateway`. This clearly applies to “modify coverage limit”, “check coverage limit”, and “create account”. Observe that it will also find “consult account”, even though the flow from the `ExclusiveGateway` to “consult account” is marked as default, while the flow in the query is not marked as default. The reason is that being a default flow is an optional property of the gateway, and the query does not specify a value for this property, which means it matches all values. On the other hand, “compute coverage” is not matched because its type is `SubProcess` while the type specified in the query is `Task`.

Query 3 detects if the “create account” and “consult account” tasks are executed exclusively, in parallel, or in some other manner, depending on the gateway preceding them. The `type isa Gateway` annotation indicates that the gateway preceding the tasks may be of any type as long as it is a subclass of the `Gateway` abstract meta class. The `default=any` is required to ensure that both default and non-default flows will be matched indiscriminately—this resembles the abstract property of UML `Classifiers`. The `steps = min` annotation limits the number of possible matches by stating that only the gateway-node closest to the two tasks should be returned. Considering the source model, Query 3 will then yield that the two tasks are executed exclusively.

The goal of Query 4 is to determine which part of the process has access to the “Contract” data input before a quote is sent to the customer. Observe that we allow paths of arbitrary length and all relevant types by the `steps` and `type` annotations, respectively. These two kinds of annotations occur frequently together so we have created the `indirect` annotation that is a shortcut for `steps=*` and `type is any`. So, Query 4 returns all paths from “Contract” to “send quote to customer”.

Query 5 illustrates the difference between syntactic and semantic querying. The intention of the query is to verify if the “modify coverage limit” task may be reached after executing the “check general coverage” task. Syntactically, the source model contains a path between the two tasks. Therefore, Query 5 will return this path. However, the question of reachability may not be reliably answered without considering semantics. Indeed, the path connecting the two tasks in the source model contains a false condition on one of its flow arrows, with the

intended meaning that control will never traverse this flow (i.e., task “modify coverage limit” will never execute). Answering this type of inquiry about process execution is a desirable but as of now unavailable feature in VM\*. Also, it is this very feature of VM\* that allows us to express Query 4 as succinctly as we have done, highlighting the trade-off between expressiveness and usability of VM\*.

### 3 Query Language

In this section we discuss the abstract and concrete syntax, and formalize the semantics of VM\*.

#### 3.1 Abstract Syntax

VM\* queries, constraints, and transformations consist of model fragments with (optional) textual annotations as defined above and formalized in the VM\* meta-model, see Fig. 5. A VMStarExpression contains one or more Rules, each of which contains one or more Patterns, each of which contains one or more elements of the host language. All containments are exclusive, and there may be annotations on expressions, rules, or patterns. The components of the VM\* metamodel must be mapped to existing elements of the host language metamodel. If

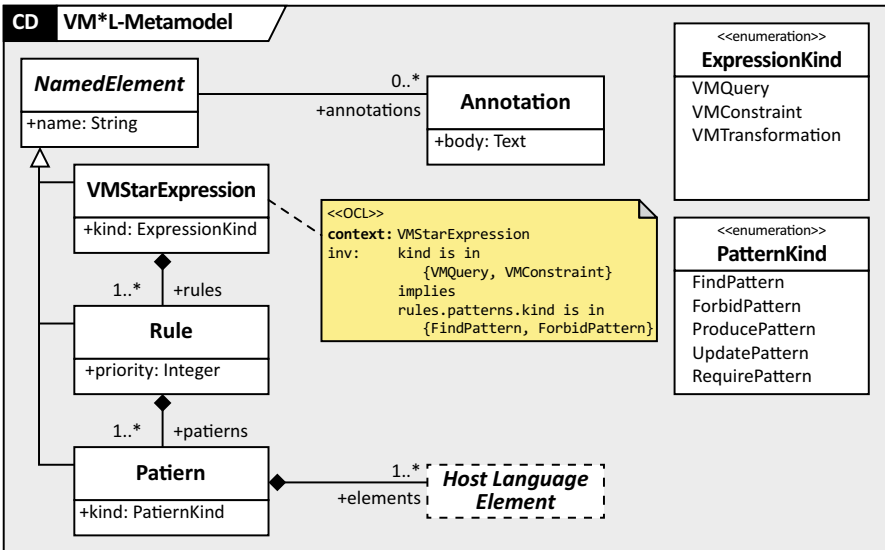







Fig. 5 The VM\* metamodel



**Table 2** Mapping of VM\* metamodel constructs into UML language constructs to create VM\*UML. In order to implement this mapping, a UML profile with these constructs must be created, and applied to any UML Packages containing VM\* expressions

Stereotype	Applies to	Description
«VM* Annotation»	Comment	Annotation for any complete scope, e.g., a package of several expressions
«VM* Query» «VM* Constraint» «VM* Transformation»	Package	Annotation for a complete expression
«VM* Find» 	Package, Comment	Find patterns are used as queries, constraints, or as the <i>left-hand side</i> (LHS) of a transformation rule. A rule may contain at most one find pattern. A rule must contain either a find pattern or an update pattern
«VM* Forbid» 	Package, Comment	Forbid patterns are <i>negative application condition</i> (NAC) for a transformation rule or constraint. A rule can contain any number of forbid patterns and will be executed only if <i>none</i> of these patterns is matched in the source model
«VM* Require» 	Package, Comment	Require patterns represent a <i>positive application condition</i> (PAC) for a transformation rule. A rule can contain any number of require patterns, and will be executed only if <i>all</i> of these patterns are matched in the source model
«VM* Produce» 	Package, Comment	Produce patterns represent the <i>Right-hand side</i> (RHS) of a transformation rule, specifying how the target model is to be obtained from the source model. A rule may contain at most one produce pattern, and if there is a produce pattern, there must also be a find pattern in the same rule
«VM* Update» 	Package, Comment	Update patterns amalgamate a find and a produce pattern. There may be at most one update pattern per rule, and there may not be an update pattern and a find pattern together in a rule

the host language offers extension mechanisms, like the profiles and stereotypes of UML, these should be used. For instance, for VM\*UML, annotations are encapsulated in UML comments annotated by the VM\* Annotation stereotype. Table 2 shows how the VM\* constructs map to the host language UML. For languages lacking extension facilities, naming conventions can be used to the same effect.

VM\* specifies five pattern types: find, forbid, require, produce, and update. Queries and constraints consist of exactly one find pattern, and any number of forbid patterns. Constraints must have at least one annotation with the body context. In

transformations, all kinds of patterns may occur in any number. The transformation-annotations may only be used in update and produce patterns, and must be anchored to at least one model element of the pattern. Transformational patterns correspond to *Left-Hand Side* (LHS), *Right-hand side* (RHS), Negative, and Positive Application Condition (NAC and PAC, respectively) from graph transformation theory [14].

## 3.2 Concrete Syntax

The core of the VM\* language is the set of annotations it provides. In the interest of expressiveness and succinctness, we sometimes cannot avoid referring to elements of the VM\* metamodel explicitly. Also, sometimes execution options for queries, constraints, and transformations must be specified. Both of these can be achieved with annotations. See Table 3 for a complete overview of VM\* annotations.

We start our overview of VM\*'s annotation syntax by describing *user-defined variables*. They can be declared and manipulated within VM\* annotations, and also used as meta-attribute values in pattern specifications. The names of user-defined variables are prefixed by the \$ character. Their *scope* extends across all patterns included in a query, and they are therefore employed for identifying corresponding model elements across different patterns. The type of a user-defined variable is inferred at query execution time. VM\* supports the Boolean, Integer, Real, and String data types, in addition to the Element data type used for storing instances of host language meta classes. Regardless of their type, user-defined variables also accept the *undefined value* (“\*”). A variable with this value is interpreted as possibly storing any accepted value of its respective data type.

For variable manipulation, VM\* supports the arithmetic, comparison, and logic operators listed in Table 3. Logic operators can be expressed using shorthand notations (“,”, “;”, “!”, “->”) or full textual notations (and, or, not, if/then). The implication (“->”) and disjunction (“;”) operators can be combined to form a conditional if/then/else construct. The navigation operator (“.”) accesses model element meta-attributes, operations, and association-ends.

Apart from user-defined variables, VM\* relies on *special variables* as a means of controlling query execution (the injective, precision and steps special variables) and accessing the contents of the source model (the id, self, and type special variables). Special variables have a predefined *scope*, identifying the specification fragment to which they are applicable. With the exception of the injective variable, the scope of all special variables is limited to the annotated model element. The injective variable has a global scope: its value determines how all patterns of a query are matched in the source model.

*Clauses* are the main building blocks of VM\* annotations: each annotation consists of one or more clauses connected by logic operators. The use of clauses is inspired by logic programming languages and benefits annotation conciseness. A clause is an assertion about the pattern model elements to which it is anchored, about its containing pattern as a whole, or about user-defined or special variables.

**Table 3** VM\* annotations grouped by function (top to bottom): features for queries, constraints, and transformations, and generic arithmetic and logic operators

Annotation	Meaning
\$	Declares a variable to be matched (e.g., with the name of a model element)
match	Matches two variables (e.g., a name and a variable)
self	Denotes the model element to which an annotation is attached
id	Stores a model element identifier to match corresponding elements across patterns
?, *	The usual wildcards may be used in matching names of model elements
:=	Assigns a value to a user-defined variable, special variable, or model element meta-attribute
injective	When set to false, query elements may match more than one target model elements (default true)
steps	Used in an expression to restrict path lengths and types of nodes and edges on paths, allowing either comparisons to constants, the value * for “unrestricted”, or the values min or max denoting the paths extending to the shortest or longest paths available
indirect	Syntactic sugar for <code>steps=*</code> and <code>type is any</code>
type is $x$	Specify the type (meta class) of the annotated model element as being $x$ , allowing any for $x$
type isa $x$	Specify the type (meta class) of the annotated model element as being a subclass of $x$
context	Anchor for constraints, must be present at least once in any constraint
omit	Annotated element must not be matched
atLeast ( $k$ )	At least $k$ groups of annotated elements must be matched
atMost ( $k$ )	At most $k$ groups of annotated elements must be matched
either	Syntactic sugar for <code>atMost(1)</code> and <code>atLeast(1)</code>
create	Part of the right-hand side of a transformation rule, create the annotated model element
create if not exists	Like <code>create</code> , but triggers only if the element does not already exist beforehand
delete	Part of the right-hand side of a transformation rule, delete the annotated model element
priority	Sets the priority of a rule (default 1)
+, -, *, /	The usual arithmetic operators. Note that the overloading of * can be resolved by context
=, <>, <, <=, >, >=	The usual comparison operators, where string comparison employs the wildcards * and ? in the canonical way
in	The usual set containment operator, where sets are enumerated between curly braces
like	A similarity-based matching operator for strings (global threshold)
precision	Annotated element is not required to match exactly but does allow a similarity-based matching with the given threshold
and, or, not	The usual logical operators
,, ;, !	Shorthands for logical and, or, and not
if <e> then <c1> else <c2>	The usual (eager) conditional, where the else-branch is optional, and -> is allowed as a syntactic shortcut
.	The usual dot-notation to access attributes of objects

The main role of clauses is to act as additional constraints on pattern matching. Note that variable assignment (“:=”) is treated as a clause.

The `either` clause can only be included in annotations anchored to several pattern model elements. All other clauses listed in Table 3 can be included in annotations anchored to one or more pattern elements. In general, anchoring a clause to several pattern elements instead of creating several annotations containing the same clause leads to more compact specifications. The variable assignment clause (“:=”) can also appear un-anchored to any pattern elements, as variables always have a query-wide scope in VM\*.

The annotations associated with VM\* language elements play one of two roles: (i) When anchored to a host language element of a VM\* pattern, annotations offer additional information or specify constraints related to that specific element. (ii) When anchored to the VM\* expression itself, annotations specify execution options, such as global constraints on identifiers and variables.

### 3.3 Semantics

The central operation of the process of interpreting a VM\* query, constraint or transformation is the *matching* of VM\* patterns with corresponding source model fragments. For queries and constraints, the find patterns are matched against the source model, resulting in a set of intermediate bindings. Then, the forbid patterns are matched against the intermediate bindings, and the result is returned. For efficiency reasons, the annotations should be applied with decreasing strength. For instance, the `context` annotation is executed first.

For transformations, the rule with the highest priority is selected, and its left-hand side part is executed like a regular query. Any resulting bindings are then subjected to the right-hand side part of the rule. The rule application is continued until no more (new) results are yielded from executing the left-hand side. Then, the rule with the next highest priority is selected, and the process is repeated, until there are no more rules to apply. VM\* currently only allows *endogenous* model-to-model transformations, that is, transformations in which the source and target models conform to the same metamodel [12, 16]. VM\* transformations can be executed *in-place* to modify an existing model, as well as *out-of-place* to produce a new model.

As a means of formalizing the identification of matches between VM\* patterns and a source model, it is useful to consider them both as typed attributed graphs. A *model graph* is defined as a typed attributed graph intended for representing models.

**Definition 3.1** A *model graph* corresponding to a model  $M$  is a tuple  $\langle N, E, T, A, V, type, source, target, slot, val \rangle$  where:

- $N$  and  $E$  are finite sets of nodes and directed edges, respectively, with  $E \cap N = \emptyset$ ;
- $T$  is the set of node types corresponding to the meta classes included in  $M$ 's metamodel;

- $A$  is the set of node and edge attributes corresponding to the meta-attributes included in  $M$ 's metamodel;
- $V$  is the set of possible attribute values;
- $type : N \rightarrow T$  is a function assigning a type to each node;
- $source : E \rightarrow N$  is a function defining the source node of each edge;
- $target : E \rightarrow N$  is a function defining the target node of each edge;
- $slot : (N \cup E) \rightarrow 2^A$  is a function assigning a set of attributes to nodes and edges;
- $val : N \times A \rightharpoonup V$  is a partial function associating a value  $v \in V$  to pairs  $(n, a)$ , where  $n \in N$ ,  $a \in A$ , and  $a \in slot(n)$ .

Edges are uniquely defined by their source, target, and slots, i.e.,  $\forall e, e' \in E : (source(e) = source(e')) \wedge (target(e) = target(e')) \wedge (slot(e) = slot(e')) \implies e = e'$ . The “undefined” element denoted  $\perp$  is not a member of any set.

The canonical subscript notation is used in what follows to denote elements of a particular model graph. For example,  $N_g$  and  $E_g$  denote the nodes and edges of model graph  $g$ . Both *bindings* and *matches* between a VM\* pattern and a source model may be represented as model graphs; see Fig. 6 for an example representing Query Q1 from Fig. 2 as the actual query in the editor (top), the internal data structure a modeling tool might use to store the model (middle), and the semantic structure as a graph with labeled nodes (bottom).

**Definition 3.2** Given two model graphs  $q$  and  $m$  representing a VM\* pattern and a source model, respectively, a *binding* is an injective function  $\beta : N_q \rightarrow N_m$  from the nodes of  $q$  to those of  $m$ .

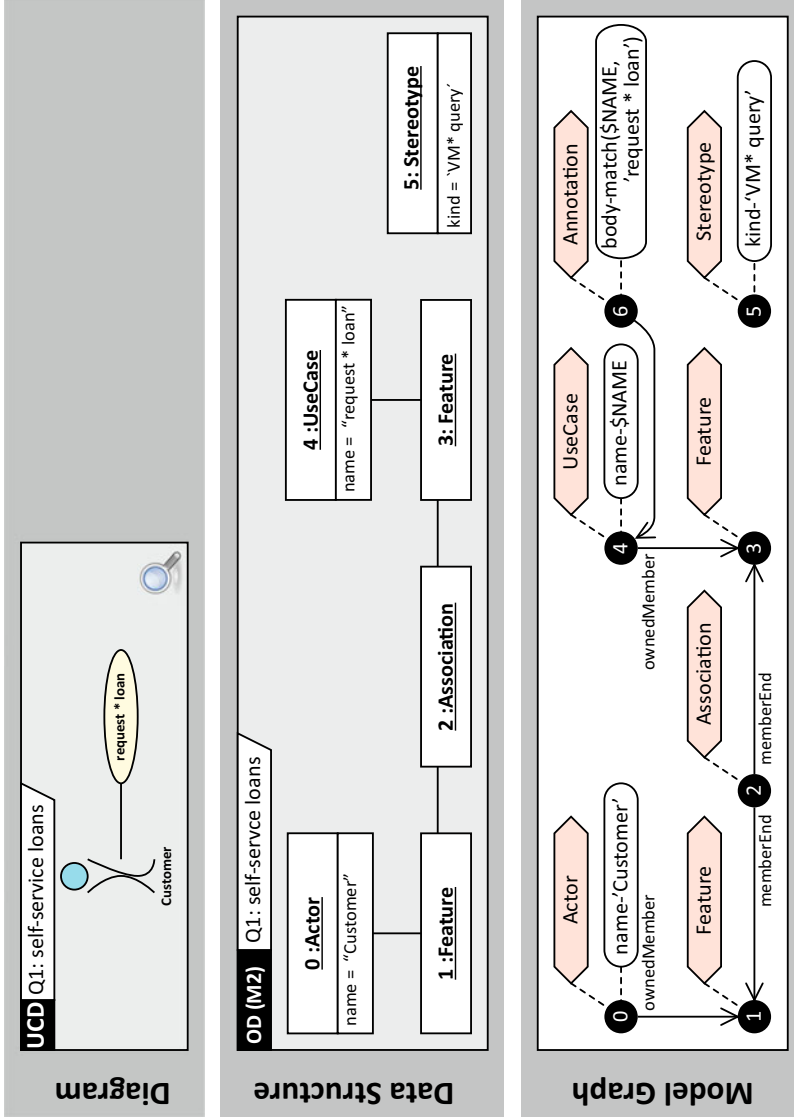
Computing a binding generates potential matches, but actual matches must meet two more conditions. First, nodes mapped by the binding must have the same type. Second, model nodes must have at least the same slots, values, and interconnecting edges as the query nodes they are bound to.

**Definition 3.3** A binding  $\beta$  is a *match* between a VM\* pattern  $q$  and a source model  $m$  iff the following conditions hold:

- (i)  $\forall n \in N_q : (type(n) = type(\beta(n))) \wedge \forall a \in slot(n) : val(n, a) = val(\beta(n), a)$ ,
- (ii)  $\forall e \in E_q : \exists e' \in E_m : slot(e) = slot(e') \wedge \beta(source(e)) = source(e') \wedge \beta(target(e)) = target(e')$ .

We define binding and match in two separate steps to highlight the algorithmic structure of VM\*: Computing the binding generates potential solutions that are then pruned by computing the match. Implemented naively, this approach is computationally inefficient, and practically useless. Informing the binding-algorithm with the matching constraints, however, drastically reduces the complexity.

Most of the VM\* annotations introduced in Sect. 3.2 have no other effect on the above definitions than to simply modify a pattern before it is matched with a source model. The `self`, `type`, and `steps` special variables are such examples. Other annotations such as `either` and `optional` imply that several different versions of a pattern must be matched with the source model. Again, this does



**Fig. 6** Query Q1 from Fig. 2 at three different levels of abstraction: the actual query (top), the internal data structure a modeling tool might use to store the model (middle), and the semantic structure as a graph with labeled nodes (bottom)

not interfere with each individual pattern's matching process. The `unique` clause imposes a filter on match results after they have been computed, i.e., only one match is allowed. All of the aforementioned annotations do not interfere with the match computation, but simply work on the results. The only annotations affecting match computation are the `injective` and `precision` special variables: assigning `false` to `injective` removes the injectivity condition in Definition 3.2. Adjusting `precision` tweaks the matching precision.<sup>1</sup>

## 4 Implementation

Developing VM\* iterated through many cycles of conceptual work, exploratory prototyping, and evaluation not dissimilar to the design science methodology. In this way, three product lines have emerged over the years, implementing (parts of) VM\* in turn.

**moq** We started with exploratory coding in PROLOG to determine the algorithmic feasibility and complexity [25, 26]. This branch later developed into query textual interfaces to study the concepts independent of the notation [30, 31]. The last step in this line is the MACH environment [33] which is available for download.<sup>2</sup> It can also be used without installation on SHARE [37].<sup>3</sup>

**MQ** We started exploring the visual notation aspect for model manipulation, with the **ModelQuery** systems, **MQ-1** [39] and **MQ-2** [3, 5].<sup>4</sup> Both are plugins to the MagicDraw<sup>TM</sup> modeling environment.<sup>5</sup> These implementations allowed to validate the overall approach, the syntax, semantic details, and the query execution performance under realistic conditions. A screenshot of **MQ-2** is shown below in Fig. 8.

**vm\*** Finally, in order to prove the execution transparency of VM\*, we realized it on two fundamentally different execution engines: Henshin [11], and as a REST-style Web-interface [4, 10].

Due to the limited space available, we can only explain one implementation here. We select **MQ-2**, since it is the basis of most evaluations. In the remainder of this

---

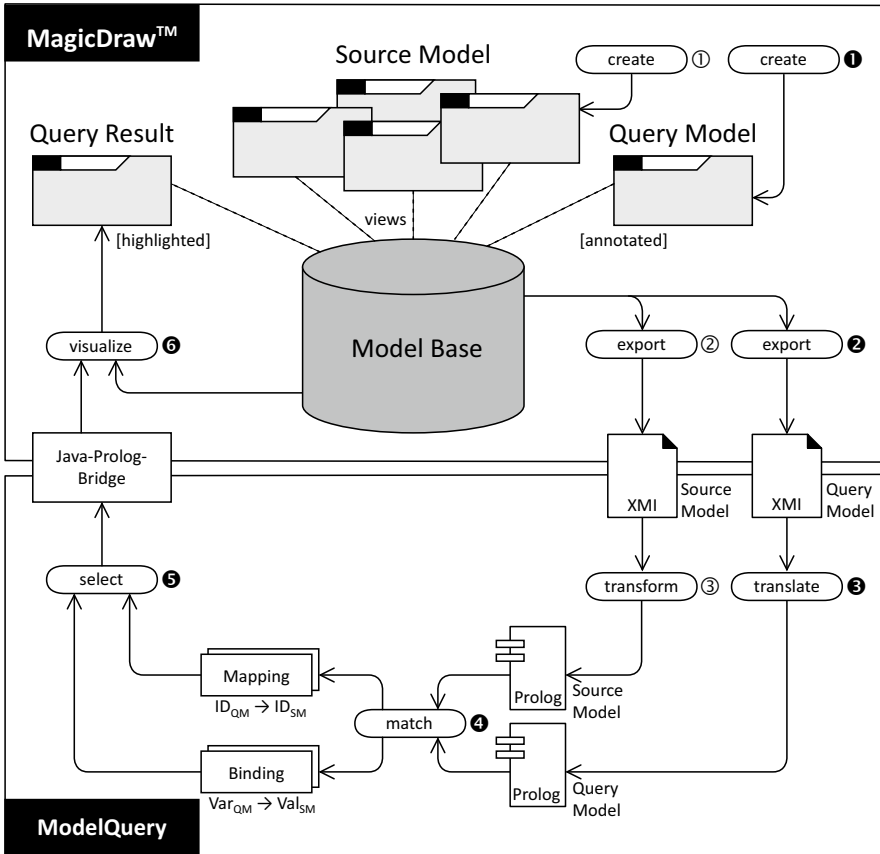
<sup>1</sup> Over the course of the years, the semantics of VM\* languages has been defined in different terminologies and notations, and with slightly different meanings. The view presented here is the one proposed in [4], superseding earlier definitions such as the logic programming-based formalization presented in [29]. There, all annotations are viewed as logic constraints to be checked by an inference engine as part of the match computation process. Here, however, we decouple pattern matching from annotation interpretation, thus allowing a much wider array of existing matching engines, particularly ones based on graph matching.

<sup>2</sup> <https://www.pst.ifi.lmu.de/~stoerle/tools/mach.html>.

<sup>3</sup> <http://fmt.cs.utwente.nl/redmine/projects/grabats/wiki>.

<sup>4</sup> <https://www.pst.ifi.lmu.de/~stoerle/tools/mq2.html>.

<sup>5</sup> [www.magicdraw.com](http://www.magicdraw.com).



**Fig. 7** The architecture of the **MQ-2** system and the main dataflow of executing queries. Numbers in black circles indicate the sequence of steps in creating and executing a query. Numbers in white circles indicate the steps for creating or changing models in the model base. Rectangles are used to represent data. Rectangles with rounded corners are used to represent actions. Arrows indicate dataflow

section, we will refer to VMQL rather than VM\*, because VM\* did not exist yet at the time of implementation. Figure 7 shows an architectural overview of **MQ-2**, while Fig. 8 presents a screenshot. The process of executing a query is shown by the numbers in black/white circles in Fig. 7. We will start with the white circles that highlight the steps for transforming the model base.

- ① A source model is created using some modeling language (UML in this implementation) and stored in the tool's model base.
- ② The model base is exported to an XMI-file, the standard file-representation of UML, using MagicDraw's built-in export facility.
- ③ The XMI-file is mapped into PROLOG predicates (see [29] for details). The mapping is bidirectional and generic, i.e., it does not limit generality of the



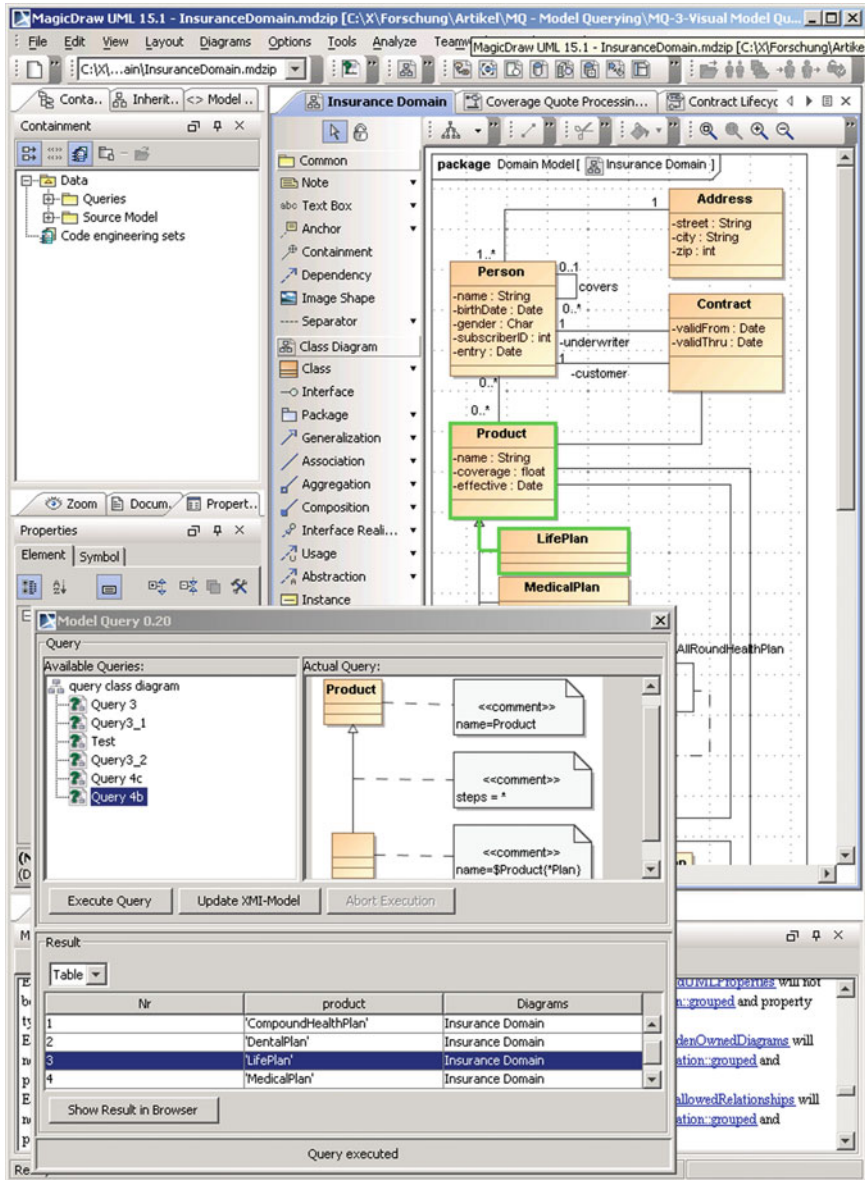


Fig. 8 The prototype implementation of VMQL. In the foreground, MQ-2 presents a list of “available queries” (top left), the one currently selected (top right), and the current result binding (bottom). One of the result bindings is selected, and a diagram in which this binding appears is shown in the background. The bound elements are highlighted with bold green borders

solution. Note that later implementations map models into graphs rather than to PROLOG predicates.

Now we turn to the process of translating and evaluating queries, indicated by numbers in black circles in Fig. 7. Since VMQL queries are annotated fragments of regular models, executing a query on a given model base boils down to finding matches between the query and the model base, and checking the constraints provided by the annotations.

- ❶ A VMQL expression is entered as a regular UML model with constraints packaged in stereotyped comments.
- ❷ ❸ Then, the model query is transformed just like the source model. Constraints are directly mapped to predefined Prolog predicates and added to the predicates yielded from translating the query. Note that the precise way of how the constraint predicates are added is crucial for the computational complexity. In later implementations, queries are mapped to graph transformations instead of Prolog predicates.
- ❹ Next, the predicate resulting from translating the model query is run on the Prolog-database resulting from transforming the source model. The two models are matched and the constraints are evaluated.
- ❺ ❻ Finally, the user selects one of the matches found in the previous step. To support this, a list of all diagrams containing elements of the match is computed. These may be either exact or approximate matches, as controlled by the `precision` constraint. If appropriate, the list is sorted by decreasing similarity. The diagram selected in the previous step is presented, and all elements of the binding are highlighted with fat green outlines (like in Fig. 8). The user may return to the previous step and select another match, and eventually terminate this query.

While the user interaction is tool specific, the query engine and the model interpretation are not (i.e., the lower part of Fig. 7). It should thus be fairly easy to port **MQ-2** to other UML tools, to DSL tools, or, in fact, to *any* modeling tool as long as it provides an open plugin API.

**MQ-2** and the other implementations of (parts of) VM\* have matured over several years of iterative refinement. Still, none of these implementations has reached the level of maturity and quality to compare to commercial products. However, our implementations do serve as evidence for several points. First, they prove feasibility of VM\* in all its parts and aspects, and the soundness of the concepts and ideas behind VM\*. Second, the implementations allowed us to get a better understanding of the computational complexity of executing VM\*. This is particularly important, because executing VM\* is ultimately based on sub-graph matching, a problem well-known to be computationally expensive in the worst case. So, it was not clear from the outset whether VM\* would lead to a viable solution for practical situations. As it turns out, VM\* is indeed viable for the intended use case, namely interactive queries, checks, and manipulations of models by modelers. However, it is likely not capable of supporting online processing of extremely large

model repositories or very complex and large sets of model transformations. It is neither suitable to aggregate trace data into process models (“process mining”). Due to lack of space, we cannot explore this aspect further in the present paper, and refer the reader to [6]. Third, we claim practical value of VM\* particularly for usability. In fact, it is this aspect that led us to develop VM\* in the first place, and it is here that we have placed the greatest emphasis of our work.

## 5 Usability Evaluation

Usability is rarely considered an important concern in many modeling communities.<sup>6</sup> For us, it is the key to the success of any modeling related approach. Therefore, the cornerstone of the model querying research done in the context of VM\* is, of course, the series of user studies to explore the usability of various model manipulation approaches. Table 4 lists the empirical studies we have conducted over the years to evaluate the relative usability of various languages for querying or transforming models. In Table 4, every line corresponds to a distinct study. Obviously, it is impossible to present all experimental results here in adequate detail, so we refer the reader to the original publications once more and restrict ourselves here to a discussion of the insights we obtained.<sup>7</sup>

Our initial hypothesis was that visual query languages should “obviously” be much better than textual languages. This turned out to be wrong in study 1 [30, 32]: All participants (senior practitioners) reached a perfect score on the OCL condition, even though they had never seen OCL before. In fact, they reached perfect scores under *all* treatments, whereas the student participants in study 0 had reached fairly low scores under each treatment. When asked, the participants in study 1 would explain that they had considered it to be some kind of pseudo-code, and simply executed it mentally, based on their intuition of the names of the operators and functions. Based on their extensive personal experience, they apparently understood the concepts, even if the syntax was new to them, and, as they asserted, less comprehensible than the syntax of the other languages tested. This gave rise to the hypothesis that there are at least two relevant factors to understanding model queries: the querying concepts (abstract syntax) and the querying notation (concrete syntax). We also speculated that participants’ scores in our tests should be less relevant than their cognitive load. Therefore, we started asking for preferences as a hint towards load levels.

---

<sup>6</sup> Of late, some different opinions are heard, though, cf. e.g., [1].

<sup>7</sup> We cite the main publications reporting on these studies, though many times results contributed to several publications, and there are several publications presented (parts of) the results. The names of the languages have evolved over time, we use here the name that the respective languages have had at the end of the research program, to make comparison easier for the reader. The first study was an exploratory pilot study to develop the research question rather than to provide meaningful results.

**Table 4** Main empirical studies evaluating VM\* and its precursors. In column “Method”, E, QE, and TA refer to **Experiments**, **Quasi-Experiments**, and **Think Aloud** protocols, respectively. The columns under “Participants” detail the kind and number of participants in the study (**Students**, **Practitioners**, and domain **Experts**). In column “Mode”, R, and W stand for reading and writing of queries or transformations

No.	Ref.	Method	Participants			Languages	Intent	Mode
			S	P	E			
0	[26]	QE	5			VMQL, OCL, NLMQL, LQF	Query	R, W
1		QE,TA		5				
2	[30, 32]	E	12	6	6	VMQL, OQAPI, NLMQL	Query	R
3		E	16					
4	[29]	E	20			VMQL, OQAPI	Query	R, W
5		E	17					
6	[4]	E	24			VM*, OQAPI	Query	R, W
7	[4, 10]	E	30		4	VM*, Epsilon, Henshin	Transformation	R
8		E	44					
9		TA				VM*		

In studies 2 and 3, we switched from an exploratory research method to the classic experimental paradigm. Besides increasing the number of participants, we also explored sub-populations with different levels of qualification, and studied the controls more carefully, with surprising effects. NLMQL is a made-up purely textual model query language that we had introduced as a control, pitching textual vs. visual styles of querying. We had expected that the visual VMQL would outperform not only OCL (including the improved OCL-variant OQAPI) but also NLMQL. But the opposite happened, which led us to speculate that there are more factors at play than just the visual or textual notation of a query language. We attributed the difference to the proximity of the concepts in the query language and the concepts referred to in the experimental tasks. Put in another way: part of the usability of a model query language could be found in the appropriateness of the language concepts.

The subsequent studies 4 and 5 replicated the effects found previously and confirmed our speculations. Thus, we consider it an established fact that there are clear differences in usability of different languages, both with respect to reading and creating queries and constraints. Furthermore, the available evidence suggests that there is indeed a second factor which we call *language concept appropriateness* (LCA). Actually, the influence of LCA seems to be larger than whether a notation is visual or textual. We also stipulate that the effect can be masked by expertise and intellectual prowess, and so manifests itself mostly under stress (e.g., time pressure), and through variation across a population rather than through in vitro experiments.

Re-analyzing our data, we also hypothesize that there might be a third factor at play beyond the syntax and concepts of the queries that were presented to our participants: in all our experiments, the answer options for participants to choose from were given as prose. This might bias our results in favor of the textual,

prose-like notation (NLMQL, see studies 0–3 in Table 4). There is currently no experimental evidence to confirm this speculation, though.

After study 5, a major redesign of the language took place in the context of extending it to covering model transformations, yielding the VMTL-language. Of course, we also had to switch our controls from model query to model transformation languages (MTLs). We picked two best-in-class languages, namely the visual MTL Henshin, and the textual MTL Epsilon. In studies 7 & 8, we obtained similar results than we saw previously for model query languages. These studies showed no substantial advantage of one language over the other, and we currently do not understand why this is the case. In particular, we were surprised by the comparatively good results found for the “Epsilon” treatment. We hypothesized that this is due to the sampled population: Their educational background (CS graduate students) might introduce a bias in favor of Epsilon, which is very similar in style to common programming languages. This would be in line with our very first results, where the strong CS background of the study participants obviously had allowed them to cope with a language like OCL, despite its obvious usability deficiencies [26, 29, 30].

However, this is of course far from the “normal” situation, where domain experts typically do *not* have CS expertise—they have *domain* expertise, and probably some experience in reading models, and maybe even in creating models. Such users, we speculate, should have a much harder time coping with OCL, Epsilon and other languages created by computer scientists for computer scientists. Such users, we believe, outnumber CS experts by far, and they deal with models (in their domains) on a daily basis. They have no experience in programming or Model Driven Development technology, nor do they have an intrinsic motivation to use this technology.

Pursuing this idea, we conducted an observational study involving participants with and without a programming background (study 9). Interested in their opinion and thought processes rather than their scores, we switched to a think aloud protocol to find out how users understood the various languages. As expected, the presence of substantial programming experience lead to a completely different approach as compared to the non-programmers. Given that this is an interactive, observational study based on a very small set of participants, our results do not support strong interpretations and broad generalizations. On the other hand, our results likely generalize from queries to constraints and transformations, based on the structural similarities outlined in the first section of this paper. So, queries are the essential building blocks for all kinds of model manipulations, and likely the most used part, too.

## 6 Applications and Use Cases

In the Software Engineering domain, several types of models occur frequently, e.g., class or entity-relationship models, state machines, interaction models, and so on. Numerous powerful model manipulation languages and tools have been developed in the context of the MDE-paradigm, which places “*model transformation at the*

*heart of software implementation*” [22, p. 42]. Their origin has shaped them in a profound way, particularly considering the application scenarios, model types and the capabilities expected from modelers: Understanding and using MDE-style model manipulation approaches hinges on a perfect understanding of the metamodels underlying the modeling languages. Obviously, this is hardly part of the skill set of most domain experts. This means that MDE-flavored model manipulation languages, while expressive and supported by powerful tools, cater only for a very small audience of MDE-experts.<sup>8</sup>

On the other hand, the BPM community has more readily considered modelers which are experts in the domain modeled rather than technology. Thus, usability of model manipulation tools is a concern of much greater importance here. However, this domain typically considers only one type of models, namely process models. Additionally, existing approaches typically only deal with models of one particular notation. So, while a potentially much greater audience is addressed, a smaller set of models is covered. We believe that conceptual models are abundant today, created and used by knowledge workers without MDE expertise—think of organizational charts, shift plans, mechanical and electrical engineering models, Enterprise Architecture models, and chemical process schemas. These types of conceptual models have complex and long-lasting lifecycles. They are created, refactored, translated, and migrated in much the same ways as software models. End-users working with conceptual models are usually academically trained and highly skilled in their domains. We call them *End-User Modelers* (EUM), and they are at the focal point of our vision of model manipulation. We aspire high degrees of usability and learnability to cater for

EUMs, yet expressive and generic enough to cover their many application intents and modeling languages. This is an instance of the “*Process Querying Compromise*” [21, p. 12]. Pursuing this goal we are prepared to give up a degree of expressiveness.<sup>9</sup> We win, however, a world of applications, as we shall illustrate in the remainder of this section.

Imagine a world where lawyers, mechanical engineers, accountants, and other domain experts have access to a language and tool that allows them to specify model transformations, queries, and constraints in a manner so close to their application domain and so simple to use that they can actually do it themselves. This would empower large numbers of knowledge workers to validate and update their models in a more efficient and less error-prone manner. The economic benefits would be hard to overstate. For instance, in an, as yet unpublished, interview study, a leading software architect from a major automotive supplier said about model querying: “... so I asked them: guys, how long does this and that take ... then I took their hourly rate, the working hours, and so on, calculated how much we would save if

---

<sup>8</sup> This might have contributed to the hesitant industrial adoption of MDE [38].

<sup>9</sup> In fact, some queries cannot be expressed in VM\* [29], and some VM\* queries are not computable [4, Sec. 6.3]. Also, there is a poor worst case performance of the underlying execution algorithm. We argue, however, that VM\* still covers a large part of the *actual* application space.

*we can only shave off 5 minutes a day. Those tens of millions [of €]... they didn't ask any more questions after that."*

As a first example, consider a supplier of financial services. Over the last decade, substantial new legislation has been implemented to regulate financial markets. It is now widely accepted that "*a comprehensive understanding of business processes is crucial for an in-depth audit of a company's financial reporting and regulatory compliance*" [17]. A current trend in making this possible involves audits of the business processes, or, to be more precise, audits of the business processes *models*. Given the large number of processes and applicable regulations, companies are struggling to have fast and cost-effective audits. If auditors can create their own queries on these process models, they will be more effective in narrowing down items to check manually. Since this is already beneficial, imagine the added benefit of replacing non-compliant patterns of activity with compliant ones automatically and consistently.

As a second example, consider an Enterprise Architecture scenario focusing on compliance and change-management. Industrial installations with potential impact on safety and environment are subject to stringent regulation explicitly demanding up-to-date digital models of the installation so that, e.g., malfunctions resulting in environmental pollution can be traced back. Also, emergency response forces need full technical details of a plant, say, to effectively carry out their work if needed. Imagine an oil rig where a maintenance engineer discovers a burst pipe spilling oil and blocking an evacuation corridor on the rig. Even more importantly than repairing the damage is forwarding the information to all people and systems concerned. With models as the backbone of information management of the industrial installation, this amounts to the need for fast and accurate update to several interconnected models. While speed is of the essence, ensuring that all the right elements of the model are found and consistently updated globally is difficult, slow, and error-prone for simple editing or search/replace. On the other hand, pre-defining changes or back-up models is insufficient due to the unpredictability of changes and mitigating actions. Deferring the update to a back office loses vital context information and takes too long time. In such a situation, the maintenance crew should do the update on the spot.

## 7 VM\* and PQF

Polyvany and colleagues have introduced the so-called *Process Querying Framework* (PQF) [21]), which "*aims to guide the development of process querying methods*", where "querying" includes all kinds of model manipulations in their terminology. Using PQF as a frame of references, we discuss VM\* using the concepts and viewpoints defined there.

PQF consists of four parts with a number of activities ("active components") to be executed on models. Of the 16 activities defined explicitly in PQF, the following six are instantiated in VM\*.

**Formalize, Index, Cache.** The process of compiling VM\* specifications into executable Henshin transformations [4] or Prolog programs [3, 5] is fully automated. The translation of models includes manipulations that amount to indexing and caching (see steps ③ and ⑤ in Fig. 7).

**Inspect.** Matched model fragments are presented to users using the same notation and tool used to create the host model and the VM\* query specification. The modeling tool used for creating the target models in the first place is also used to inspect the model.

**Visualize.** The concrete syntax of the host modeling language is used to visualize query results (see Fig. 8 for an example).

**Filter.** PQF's notion of model repository corresponds to a common (large) UML model, so that selecting sub-models by VM\* queries amounts to filtering in PQF's understanding of the term. PQF only considers static reductions of the search space, though.

Other components of the PQF are not instantiated in VM\*. PQF also defines a set of variation points ("design decisions") by which process query approaches may vary. VM\* has the following stance on these design decisions.

**Design Decision 1: Models.** VM\* aspires to be applicable to all model types, and to all (common) modeling languages. The restrictions to our aspirations are discussed below and are fairly limited. As long as a modeling language has a visual syntax, is implemented in a tool, and is defined using a metamodel (or, indeed, a meta-metamodel such as MOF or Ecore, [20, 23]), VM\* is applicable.

**Design Decision 2: Semantics.** VM\* is based on syntactic matching. The underlying semantics is not considered in this process and may only enter the picture in special cases. Considerations of the execution semantics such as fairness, termination, and finiteness are not relevant for VM\*. Conversely, semantic differences along these dimensions cannot be expressed in VM\* other than by additional, semantics-specific annotations.

**Design Decision 3: Operation.** Considering a CRUD context, the VM\* languages address the *query intents* Read, Update, and Delete. The fourth intent (Create) is not supported directly in the sense that VM\* is not able to create models from scratch, though it could be achieved by update rules with empty application conditions. The supported *query conditions* (i.e., VM\* annotations) are designed with learnability and comprehensibility as priorities.

These decisions come at a price. First of all, the generality and usability are achieved, sometimes, at the expense of expressiveness: some queries cannot be expressed in VM\* [29]. We have discussed the trade-off between language expressiveness, usability, and predictability of query results in [4, Section 8.2]. We argue that VM\* still covers a large part of the *actual* application space.

Second, some VM\* queries are not computable [4, Sec. 6.3], and there is a poor worst case performance of the underlying execution algorithm, which, in the end, amounts to graph matching. While, theoretically, this incurs an exponential run-time for the worst case, this case is rarely met in industrial applications. With suitable



optimizations and heuristics, a practical solution has been implemented, as we have shown, at least for the scenarios we have considered.

## 8 Conclusion

We present VM\*, a family of languages for expressing queries, constraints, and transformations on models. Our focal point is the End-User Modeler, i.e., a domain expert working with models, but without programming background. We claim that our approach is feasible and usable and present evidence obtained through many iterations of implementation and improvement as well as a string of empirical user studies. We also claim that our approach is suitable to work with almost any commonly used modeling language, including DSLs. Since 2007, there have been over 20 publications on VM\* and the steps leading up to it (see Table 1). An extensive discussion of the related work is provided in [4, Ch. 3].

The genuine contribution of this line of research is that it is the first to take usability into serious consideration for model manipulation languages. VM\* is also comprehensive in the sense that it applies to all widely used visual modeling languages, covers many use cases, and provides many practical advantages, e.g., it readily adapts to any modeling environment. Pursuing our goal of combining usability with generality, we traded in a degree of expressiveness: some queries cannot be expressed in VM\* [4, Sect. 8.2], and some VM\* queries are not computable [4, Sect. 6.3]. Also, the theoretical worst case run-time of the execution algorithm of VM\* is exponential.

Our line of research claims to provide a greater level of scientific certainty than its predecessors or competitors, as it has been (re-)implemented several times, and evaluated for usability and performance to a much greater extent than other approaches as of writing this. An obvious gap in our validation is the absence of a large scale, real life case study, i.e., an observational study in industry where a sufficiently large set of modelers uses VM\* for an extended period of time on actual work items. Obviously, such a study would require a sufficiently well-developed tool. While we have created many tools over the years implementing (parts of) VM\*, none of them has reached the level of maturity and product quality to compare to professional solutions.

This suggests two desirable avenues of progress: an industry-grade implementation and an observational case study in industry. Both of these will be very hard to achieve, and we consider them long term goals. In the nearer future, we plan to provide a structured literature review (SLR) of this field, and a cognitively informed theory of the usability factors for model querying.

## References

1. Abrahão, S., Bordeleau, F., Cheng, B., Kokaly, S., Paige, R., Störrle, H., Whittle, J.: User experience for model-driven engineering: Challenges and future directions. In: ACM/IEEE 20th Intl. Conf. Model Driven Engineering Languages and Systems (MODELS), pp. 229–236. IEEE (2017)
2. Acreţoiaie, V.: The VM\* Wiki. <https://vmstar.compute.dtu.dk/>
3. Acreţoiaie, V.: An implementation of VMQL. Master's thesis, DTU (2012)
4. Acreţoiaie, V.: Model manipulation for end-user modelers. Ph.D. thesis, Tech. Univ. Denmark, Depart. Appl. Math and Comp. Sci. (2016)
5. Acreţoiaie, V., Störrle, H.: MQ-2: A tool for prolog-based model querying. In: Proc. Eur. Conf. Modelling Foundations and Applications (ECMFA), LNCS, vol. 7349, pp. 328–331. Springer (2012)
6. Acreţoiaie, V., Störrle, H.: Efficient model querying with VMQL. In: Proc. 1st International Workshop on Combining Modelling with Search- and Example-Based Approaches (CMSEBA'14), CEUR Workshop Proceedings, vol. 1340, pp. 7–16 (2014)
7. Acreţoiaie, V., Störrle, H.: Hypersonic - model analysis as a service. In: Sauer, S., Wimmer, M., Genero, M., Qadeer, S. (eds.) Joint Proc. MODELS 2014 Poster Session and ACM Student Research Competition, vol. 1258, pp. 1–5. CEUR (2014). <http://ceur-ws.org/Vol-1258>
8. Acreţoiaie, V., Störrle, H.: Hypersonic: Model analysis and checking in the cloud. In: Kolovos, D., DiRuscio, D., Matragkas, N., De Lara, J., Rath, I., Tisi, M. (eds.) Proc. Ws. BIG MDE (2014)
9. Acreţoiaie, V., Störrle, H., Strüber, D.: Model transformation for end-user modelers with VMTL. In: tba (ed.) 19th Intl. Conf. Model Driven Engineering Languages and Systems (MoDELS'16), no. tba in LNCS, p. 305. Springer (2016)
10. Acreţoiaie, V., Störrle, H., Strüber, D.: VMTL: a language for end-user model transformation. *Softw. Syst. Model.* (2016). <https://doi.org/10.1007/s10270-016-0546-9>. <http://link.springer.com/article/10.1007/s10270-016-0546-9>
11. Arendt, T., Biermann, E., Jurack, S., Krause, C., Taentzer, G.: Henshin: Advanced concepts and tools for in-place EMF model transformations. In: Petriu, D.C., Rouquette, N., Haugen, O. (eds.) 13th Intl. Conf. MoDELS, pp. 121–135. Springer (2010)
12. Czarnecki, K., Helsen, S.: Feature-based survey of model transformation approaches. *IBM Syst. J.* **45**(3), 621–645 (2006)
13. Davis, R.: *Business Process Modelling with ARIS – A Practical Guide*. Springer (2001)
14. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: *Fundamentals of Algebraic Graph Transformation*. Springer (2006)
15. Junginger, S., Kühn, H., Strobl, R., Karagiannis, D.: DUMMY. *DUMMY* **42**(5), 392–401 (2000)
16. Mens, T., Van Gorp, P.: A taxonomy of model transformation. *El. Notes Theor. Comput. Sci.* **152**, 125–142 (2006)
17. Mueller-Wickop, N., Nüttgens, M.: Conceptual model of accounts: Closing the gap between financial statements and business process modeling. In: Fill, H.G., Karagiannis, D., Reimer, U. (eds.) Proc. Modellierung, pp. 208–223. Gesellschaft für Informatik (2014)
18. Nagl, M., Schürr, A.: A specification environment for graph grammars. In: Ehrig, H., Kreowski, G., Rozenberg, H. (eds.) Proc. 4th Intl. Ws. Graph-Grammars and Their Application to Computer Science, LNCS, vol. 532, pp. 599–609. Springer (1991)
19. OMG: *OMG Business Process Method and Notation (OMG BPMN, v2.0.2)*. Tech. rep., Object Management Group (2014). Last accessed from <https://www.omg.org/spec/BPMN/2.0.2/> at 2018-06-22
20. OMG: *Meta Object Facility MOF (version 2.5.1)*. Tech. rep., Object Management Group (2016). Available at [www.omg.org](http://www.omg.org)

21. Polyvyanyy, A., Ouyanga, C., Barros, A., van der Aalst, W.M.P.: Process querying: Enabling business intelligence through query-based process analytics. *Decis. Support Syst.* **100**, 41–56 (2017). <https://eprints.qut.edu.au/106408>
22. Sendall, S., Kozaczynski, W.: Model transformation: The heart and soul of model-driven software development. *IEEE Softw.* **20**(5), 42–45 (2003)
23. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: EMF: Eclipse Modeling Framework, 2nd edn. Addison-Wesley Professional (2009)
24. Störrle, H.: MoMaT – A Lightweight Platform for Experimenting with Model Driven Development. Tech. Rep. 0504, Ludwig-Maximilians-Universität München (2005)
25. Störrle, H.: A PROLOG-based approach to representing and querying UML models. In: Cox, P., Fish, A., Howse, J. (eds.) *Intl. Ws. Vis. Lang. and Logic (VLL)*, CEUR-WS, vol. 274, pp. 71–84. CEUR (2007)
26. Störrle, H.: A logical model query interface. In: Cox, P., Fish, A., Howse, J. (eds.) *Intl. Ws. Visual Languages and Logic (VLL)*, vol. 510, pp. 18–36. CEUR (2009)
27. Störrle, H.: VMQL: A generic visual model query language. In: Erwig, M., DeLine, R., Minas, M. (eds.) *Proc. IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pp. 199–206. IEEE CS (2009)
28. Störrle, H.: Expressing model constraints visually with VMQL. In: *Proc. IEEE Symp. Visual Lang. and Human-Centric Computing (VL/HCC)*, pp. 195–202. IEEE CS (2011)
29. Störrle, H.: VMQL: A visual language for ad-hoc model querying. *J. Vis. Lang. Comput.* **22**(1) (2011)
30. Störrle, H.: Improving the usability of OCL as an ad-hoc model querying language. In: Cabot, J., Gogolla, M., Rath, I., Willink, E. (eds.) *Proc. Ws. Object Constraint Language (OCL)*, CEUR, vol. 1092, pp. 83–92. ACM (2013)
31. Störrle, H.: MOCQL: A declarative language for ad-hoc model querying. In: Van Gorp, P., Ritter, T., Rose, L.M. (eds.) *Eur. Conf. Proc. Modelling Foundations and Applications (ECMFA)*, no. 7949 in LNCS, pp. 3–19. Springer (2013)
32. Störrle, H.: UML model analysis and checking with MACH. In: van den Brand, M., Mens, K., Moreau, P.E., Vinju, J. (eds.) *4th Intl. Ws. Academic Software Development Tools and Techniques* (2013)
33. Störrle, H.: Cost-effective evolution of research prototypes into end-user tools: The MACH case study. *Sci. Comput. Programm.*, 47–60 (2015)
34. Störrle, H.: Cost-effective evolution of research prototypes into end-user tools: The MACH case study. In: Knoop, J., Zdun, U. (eds.) *Proc. Software Engineering (SE), Lecture Notes in Informatics (LNI)*, vol. 57. Gesellschaft für Informatik (GI) (2016)
35. Störrle, H., Acreţoiaie, V.: Querying business process models with VMQL. In: Roubtsova, E., Kindler, E., McNeile, A., Aksit, M. (eds.) *Proc. 5th ACM SIGCHI Ann. Intl. Ws. Behaviour Modelling – Foundations and Applications*. ACM (2013). [dl.acm.org/citation.cfm?id=2492437](https://doi.org/10.1145/2492437)
36. Tolvanen, J.P., Kelly, S.: MetaEdit+: defining and using integrated domain-specific modeling languages. In: *Proc. 24th ACM SIGPLAN Conf. Companion, Object-Oriented Programming Systems Languages and Applications (OOPSLA)*, pp. 819–820. ACM (2009)
37. Van Gorp, P., Mazanek, S.: SHARE: a Web portal for creating and sharing executable research papers. *Proc. Comput. Sci.* **4**, 589–597 (2011). <https://doi.org/10.1016/j.procs.2011.04.062>
38. Whittle, J., Hutchinson, J., Rouncefield, M., Burden, H., Heldal, R.: Industrial adoption of model-driven engineering: Are the tools really the problem? In: Moreira, A., and others (eds.) *16th Intl. Conf. MoDELS*, pp. 1–17. Springer (2013)
39. Winder, M.: MQ – Eine visuelle Query-Schnittstelle für Modelle (2009). In: German (MQ – A visual query-interface for models), Bachelor’s thesis, Innsbruck University

# The BPMN Visual Query Language and Process Querying Framework



Chiara Di Francescomarino and Paolo Tonella

**Abstract** Business needs often demand business analysts to inspect large and intricate business process models for analysis and maintenance purposes. Nevertheless, manually retrieving information in these complex models is not a rewarding error-free activity. The automatic querying of process models and visualization of the retrieved results represents a useful opportunity for business analysts in order to save their time and effort. This demands for languages that can express process model characteristics and, at the same time, are easy to use for and close enough to the knowledge of people working with process models. This chapter provides an overview of BPMN VQL, one of the languages for querying business processes. BPMN VQL allows business analysts to query process models and retrieve both structural information and knowledge related to the domain. Beyond the performance of the query language implementation, we have investigated the benefits and drawbacks of its use. An empirical study with human subjects has been conducted in order to evaluate the advantages and the effort required, both for BPMN VQL query formulation and understanding.

## 1 Introduction

Business process models can be very large in size and retrieving information scattered across the control flow is often resource and time-consuming. One option to retrieve aspects of interest is querying the process model, i.e., matching a query, which expresses the desired concern, against the process elements. The possibility of automatically querying processes and visualizing the retrieved results is potentially very useful for business designers and analysts.

---

C. Di Francescomarino (✉)  
Fondazione Bruno Kessler–IRST, Trento, Italy  
e-mail: [dfmchiara@fbk.eu](mailto:dfmchiara@fbk.eu)

P. Tonella  
Università della Svizzera Italiana (USI), Lugano, Switzerland  
e-mail: [paolo.tonella@usi.ch](mailto:paolo.tonella@usi.ch)

Querying business processes demands for languages able to express process model characteristics and, at the same time, easy to use for the people working with process models. Many of the process query languages in the literature [13], in fact, are visual languages, exploiting a process-like visual representation for expressing the process model properties that constitute the query. *BPMN VQL (BPMN Visual Query Language)* [5] belongs to this stream of languages for querying business processes. *BPMN VQL* allows business analysts to query BPMN process models, possibly annotated with domain semantic information. *BPMN VQL* has a syntax close to BPMN and its semantics is grounded in SPARQL [14], a language for querying RDF, that is, a semantic query language able to retrieve and manipulate data stored in Resource Description Framework (RDF) format.

In this chapter, we present *BPMN VQL*. Moreover, we present an implementation of a framework that executes *BPMN VQL* queries by converting them to SPARQL queries, and an evaluation of the language, both in terms of time performance, as well as ease of use for business analysts. Specifically, we measured the time performance of the query engine implementation on different types of queries. For the second analysis, we performed an empirical study with human subjects, in which we have investigated the advantages/disadvantages of using *BPMN VQL* with respect to natural language queries. The results of the two studies indicate that (i) the time performance of *BPMN VQL* is compatible with the online usage of the language and (ii) the visual language presents advantages both in terms of query formulation and query execution with respect to the same queries expressed in natural language. The results of the empirical study, indeed, show that understanding *BPMN VQL* queries is easier than understanding natural language queries and that formulating *BPMN VQL* queries is easier than matching natural language queries against a process model.

In the next sections, we introduce background knowledge on semantically annotated business process models (Sect. 2) and present *BPMN VQL* (Sect. 3). In Sect. 4, we describe an implementation (Sect. 4.1), the time performance evaluation (Sect. 4.2), as well as an empirical evaluation (Sect. 4.3) of *BPMN VQL*. In Sect. 5, we map the proposed query language onto the process querying framework (presented in [13]). We finally report conclusions and future work in Sect. 6.

## 2 Background

Business process models are mainly focused on the representation of activities, performers, and control- and dataflows. Domain information is conveyed in process models only as informal labels associated with business process model elements (e.g., the textual labels of BPMN activities). However, process element labeling is usually not rigorously performed by designers, thus resulting, e.g., in case of large business processes, in situations of label inconsistency. It may happen, in fact, that tasks with different labels are used to represent the same activity or that different labels are used for describing different specializations of the same activity, adding irrelevant information with respect to the considered abstraction level. Moreover, the

amount of information that can be encoded in a human readable label is necessarily limited.

In order to deal with the problem of providing business process elements with domain knowledge that is both clear for humans and accessible to machines (thus enabling automated reasoning mechanisms), business process elements can be enriched with annotations characterized by a semantics explicitly organized in a structured source of knowledge, i.e., with semantic concepts belonging to a set of domain ontologies [5]. Semantic annotations, in fact, can be used to provide a precise, formal meaning to process elements.

The semantic annotation of business processes and, in particular, of their underlying Business Process Diagrams (BPDs) is graphically represented by taking advantage of the BPMN *textual annotations*. In detail, in such a semantic variant of BPMN, ontology concepts associated with BPD elements are prefixed with the “@” symbol. Figure 1 shows an example business process diagram semantically annotated with concepts from a domain ontology, an extract of which is reported in Fig. 2. For instance, the tasks starting the three parallel flows for the raw product purchase, though exhibiting different labels (*Look for product A in the warehouse*, *Check product B availability* and *Search for product C*), represent the same concept, i.e., all of them check whether a product is available in the warehouse. Assuming that the domain ontology that describes the domain contains a concept `to_check_product_availability` (see Fig. 2), the three tasks can be semantically annotated using this, more general, concept. This semantic annotation allows the unification of the semantics of the three tasks, while preserving their original labels.

Semantic information is crucial for activities that involve reasoning and require automated support [11], as for example documenting or querying a process. In order to enable automated reasoning on a semantically annotated BPD, the process model can be encoded into a logical knowledge base, the *Business Process Knowledge Base* (BPKB) [3, 4]. Figure 3 describes the components characterizing the BPKB.

BPKB contains a BPMN Ontology, BPMNO [15], which describes the structural properties of BPMN process models, and a business domain ontology BDO, used for the semantic annotation of the process models. Moreover, the BPKB also contains a set of constraints (merging axioms and process specific constraints), related to both ontologies and, finally the BPD instances, i.e., the elements of the semantically annotated business process model. The BPKB is implemented using the standard semantic Web language OWL (Web Ontology Language) based on Description Logics [12] (OWL-DL). Description Logics (DL) are a family of knowledge representation formalisms which can be used to represent the terminological and assertional knowledge of an application domain in a structured and formally well understood way. The terminological knowledge, contained in the so-called Tbox, represents the background knowledge and the knowledge about the terminology (classes and properties) relevant for the described domain. In our case, the terminological part, which is the stable description of the domain, is provided by the upper level modules of Fig. 3, i.e., the BPMNO and BDO. The assertional part, the so-called Abox, contains knowledge about the individuals which populate the given domain in the form of membership statements. In our

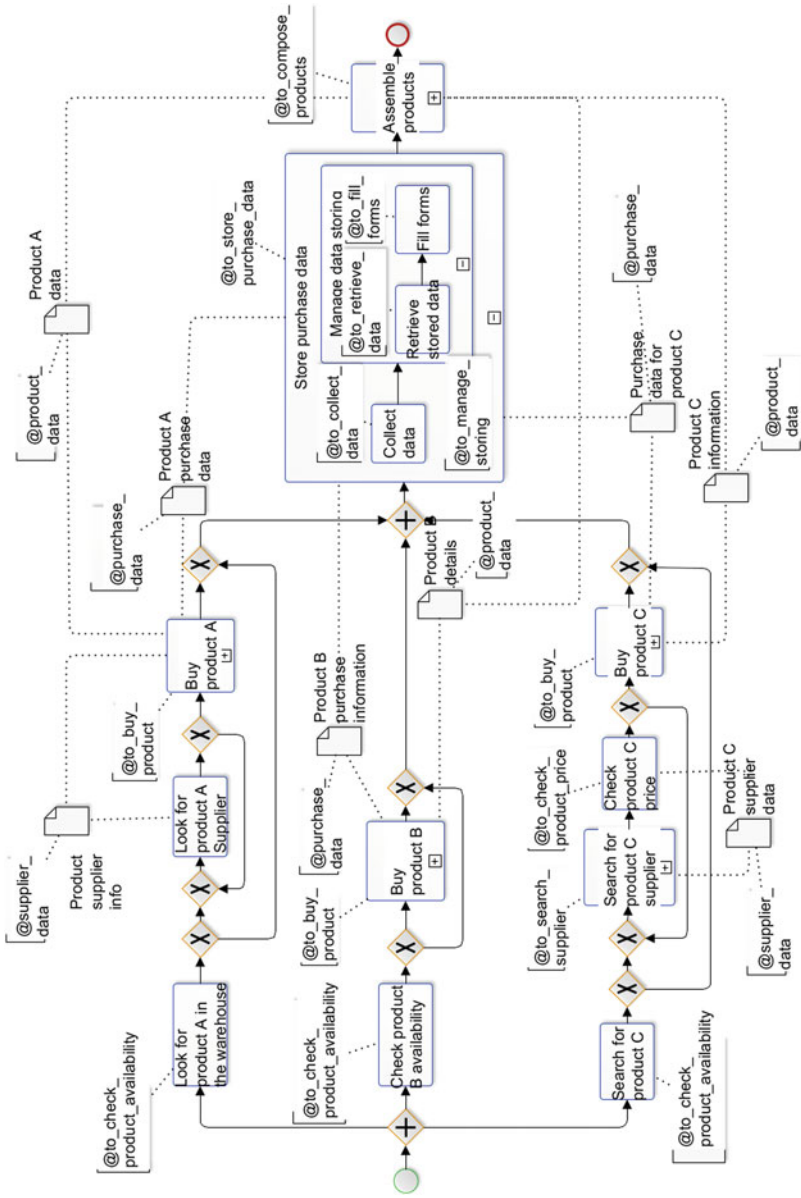


Fig. 1 An example of a semantically annotated BPMN process model

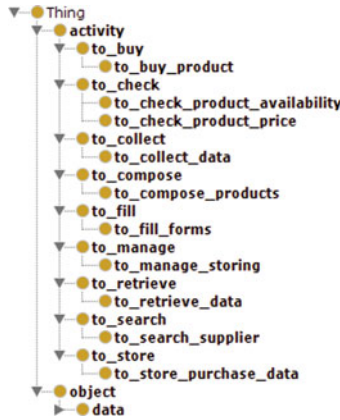


Fig. 2 Domain ontology

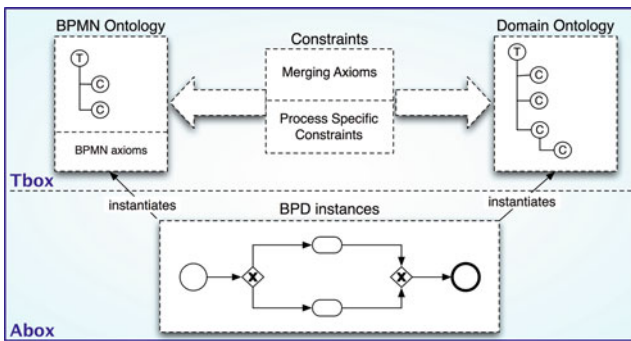


Fig. 3 BPKB overview

case, the assertional part, which is the changeable part, corresponds to a specific process model description. The semantically annotated business process model is encoded in the BPKB by populating it through the assertional knowledge. Each semantically annotated process element in the BPD is instantiated as an individual of both the **BPMNO** concept corresponding to the class of the BPMN construct it represents and of the **BDO** concept corresponding to its domain semantics, i.e., the concept that is used for its semantic annotation. In detail, in the assertional part of the BPKB, for each semantically annotated element of the business process diagram, we have a BPM-type assertion that specifies the **BPMNO** type of the element and a BPM-semantic assertion that specifies its **BDO** type. For instance, assuming that the process element labeled with *Look for product A in the warehouse* in Fig. 1 is called  $t_1$ , we declare that it is a task with the BPM-type assertion  $\text{task}(t_1)$ , while the assertion  $\text{to\_check\_product\_availability}(t_1)$  states that task  $t_1$  is an instance of the concept  $\text{to\_check\_product\_availability}$ . Moreover, BPM-structural assertions are used to store information on how the graphical objects are connected. For instance,



assuming that the first parallel gateway in Fig. 1, the task *Look for product A in the warehouse* and the sequence flow connecting the gateway to the task are called  $g_1$ ,  $t_1$  and  $s_1$ , respectively, the assertion `has_sequence_flow_source_ref` ( $s_1$ ,  $g_1$ ) states that the sequence flow  $s_1$  originates from the gateway  $g_1$  and the assertion `has_sequence_flow_target_ref` ( $s_1$ ,  $t_1$ ) states that the sequence flow  $s_1$  ends in  $t_1$ .

### 3 BPMN VQL

BPMN VQL is a visual language for the automated querying of business processes. It is able to quantify over BPMN business process elements, localize interesting parts of the BPD, and, once identified, present the identified parts to the user by visually highlighting their occurrences in the BPD. Moreover, since a critical issue of the query language is usability (it is going to be used by business designers and analysts), the proposed language is a visual language, as close as possible to what business experts already know, i.e., BPMN itself.

#### 3.1 Syntax

Queries in BPMN VQL are built by using:

- standard BPMN graphical notation, to quantify<sup>1</sup> over BPD objects;
- stereotypes for BPMN hierarchies of BPD graphical objects;
- semantic annotations (and inference reasoning) for BPD objects with a specific business domain semantics;
- composition of semantic annotations by means of the logical operators ( $\wedge$ ,  $\vee$ , and  $\neg$ ), to quantify over specific BPD objects or groups of objects with a precise business domain semantics;
- composition of subqueries by means of the OR and/or NOT operators;
- the transitive closure of direct connections between BPD flow objects, expressed by means of the PATH operator and matching two BPD objects connected by at least one path in the BPD;
- the transitive closure of sub-process inclusions between BPD flow objects, expressed by means of the NEST operator and matching activities nested in sub-processes;
- domain ontology relationships, expressed by means of the DOR (Domain Ontology Relationship) operator, which allows directly referencing a BDO relationship.

---

<sup>1</sup> The term “quantification” is used here, as in Aspect Oriented Programming [6], to indicate the capability of identifying and affecting several places of the process model.

The language supports the creation of queries with a structure similar to those formulated in SQL (Structured Query Language), a language for querying relational databases. Indeed, regardless of the specific graphical notation being used, BPMN VQL queries are characterized by a “matching” part (*matching criterion*), that determines the criterion to match (i.e., the **WHERE** clause or selection in an SQL query), and by a “selection” part (*selection pattern*), for identifying the selected portion of the matching result (i.e., the **SELECT** clause or projection in an SQL query). These two parts can be identified visually in the BPMN VQL query: differently from the components of the *matching pattern*, those of the *selection pattern* have dark background, thick lines and bold font style.

Since BPMN language and syntactic matching on BPMN allow analysts to formulate queries that can trivially obtain, via enumeration, every process subpart, including the whole process, the described visual language is complete. Moreover, the features of BPMN VQL that support quantification and reasoning, allow for the definition of complex queries in a compact form.

### 3.2 Semantics and Notation

In the following, we describe BPMN VQL in more detail using examples that refer to the product assembly process shown in Fig. 1. For each example query, in order to give it a precise semantics, we provide the translation of the query into SPARQL.

#### Queries Using Standard BPMN Graphical Notation

Individual graphical objects in the BPMN notation (e.g., rounded rectangles, diamonds, arrows) are used to match either a specific instance of the BPD with a specific label (if the BPD object in the query is labeled with that label), or all the instances of the corresponding BPMNO class, i.e., of the corresponding BPMNO-type, regardless of their label (if the BPD object in the query is unlabeled). In the first case, it is necessary to specify both the BPMNO-type of the BPD object and the label of the specific instance required. In the second case, it is sufficient to provide the specific BPMN object representation, without any label, thus indicating any instance of the specified BPMNO-type. However, the expressive power of the BPMN notation in the BPMN VQL is not limited to individual graphical objects. By composing together several BPD objects, in particular, by linking flow objects and/or artifacts by means of connecting objects, it is possible to match subparts of the process.

#### Queries Using Stereotypes

Stereotypes, which are indicated within guillemets inside the BPMN activity symbols (i.e., rounded rectangles), can be used to represent (sub-)hierarchies of BPD graphical objects. They match all the BPD instances of the specified BPMNO class or of its subclasses. For instance, an  $\ll Activity \gg$  stereotype will match all the BPD instances of type task or sub-process.

### Queries Using Semantic Annotations

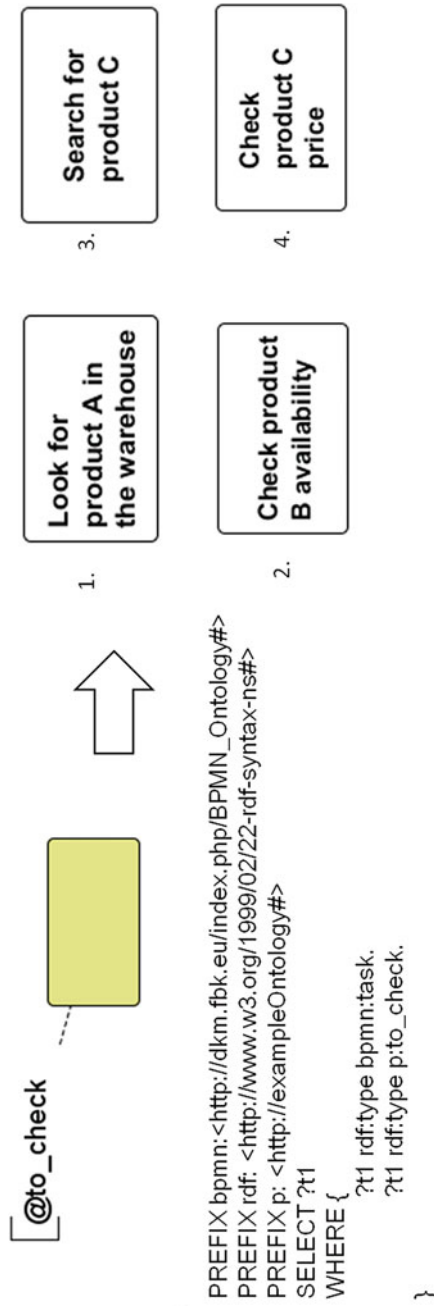
Queries exploiting semantic annotations are used to select instances of the BPMNO (i.e., of BPMNO-type), representing, directly (i.e., without inference) or indirectly (i.e., with inference), a specific ontological concept of a given BDO-type. The BPMNO instances in the query result, hence, are also instances of a class/superclass of the BDO ontology. We can think about a scenario in which, for security reasons, all the activities making checks have to be documented. In order to intervene on these activities, they need to be retrieved in process models. In Fig. 4, for example, we ask for all the tasks that *check* something. Although annotated with different (i.e., more specific) concepts, tasks are added to the result, as long as *to\_check* is an ancestor of their annotations (e.g., their annotation being *to\_check\_product\_availability* and *to\_check\_product\_price*). In another business scenario, it might be necessary to understand (for marketing purposes) what is usually searched, and what happens immediately after. Figure 5 provides an example in which standard BPMN VQL, stereotypes and semantic annotations are used together for retrieving all the pairs of directly connected activities and their connecting sequence flow, such that the source of the pair is a *to\_search* activity. The result of the query run on the process model in Fig. 1 contains the task labeled with *Search for product C supplier*, the task labeled with *Check product C price* and the sequence flow directly connecting them.

### Queries Using Logical Operators for Semantic Annotations

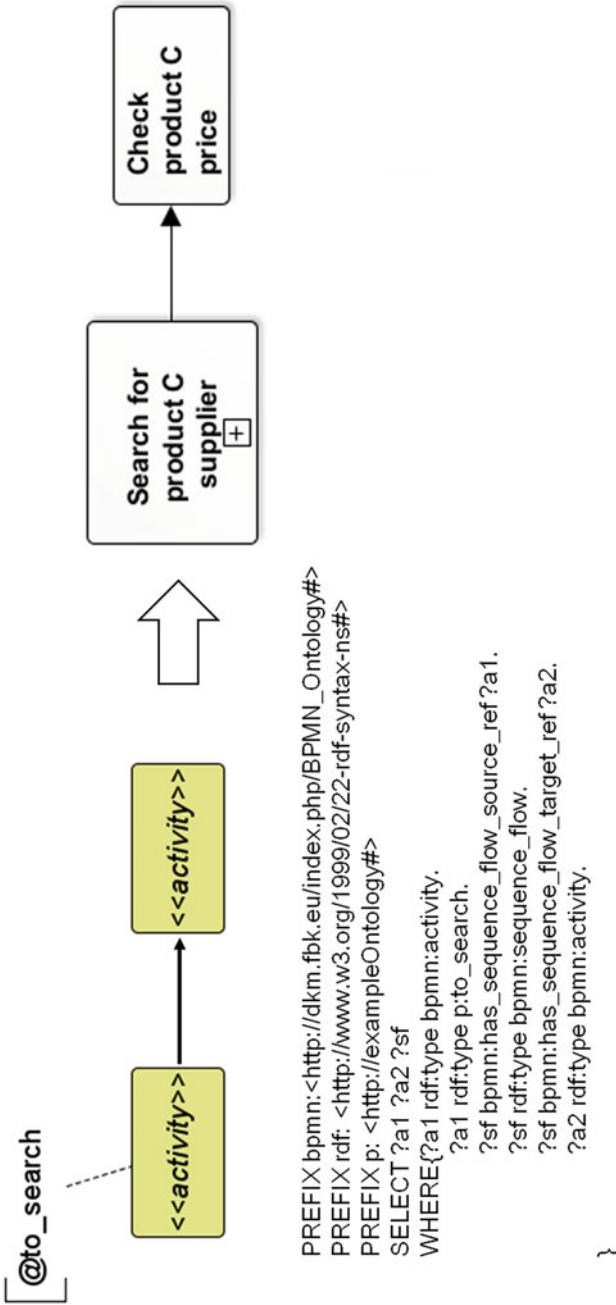
In order to compose query annotations, the standard logical operators ( $\wedge$ ,  $\vee$ ,  $\neg$ ) are used. Their semantics is the following one: the  $\wedge$  (and) operator is a binary operator representing the intersection between two concepts, returning all the instances common to the two operands in the  $\wedge$  expression. The  $\vee$  (or) operator is another binary operator representing the union between two concepts, returning all the instances belonging to one or more operands in the  $\vee$  expression. The  $\neg$  (not) operator is a unary operator representing the negation of a concept. It returns all the instances that do not belong to the set of the concept instances. For instance, in Fig. 6, the  $\wedge$  and the  $\neg$  operators are composed together in order to select all the activities (both tasks and sub-processes) in the process that *check* something, except for the product price. The result of applying the query on the process model of Fig. 1 consists of the three tasks that check product availability.

### Queries Using Operators for Composing Subqueries

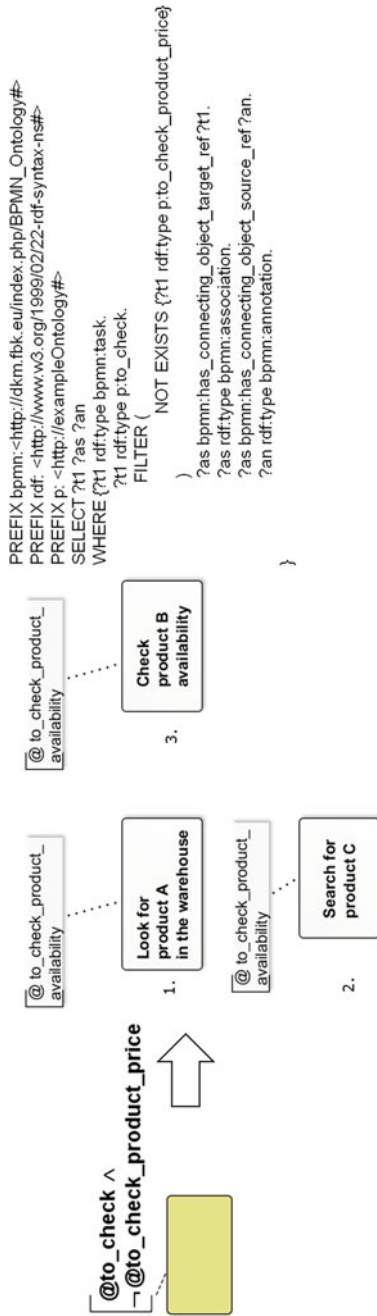
In order to express complex queries, BPMN VQL provides three operators for composing subqueries. The default operator between two or more subqueries is the intersection of the results provided by each subquery. Two more operators are introduced in order to support the union and the negation of subquery results. The OR operator is depicted as a dotted table listing all possible alternative subparts of the query. We can think about a scenario in which the documentation should be delivered before the activities performing checks, as well as before all the decision points. The query in the example in Fig. 7 asks for all the instances of any activity followed by an exclusive gateway or by an activity that *checks* something. The result consists of the eight activities preceding the six exclusive gateways and of the *Search*



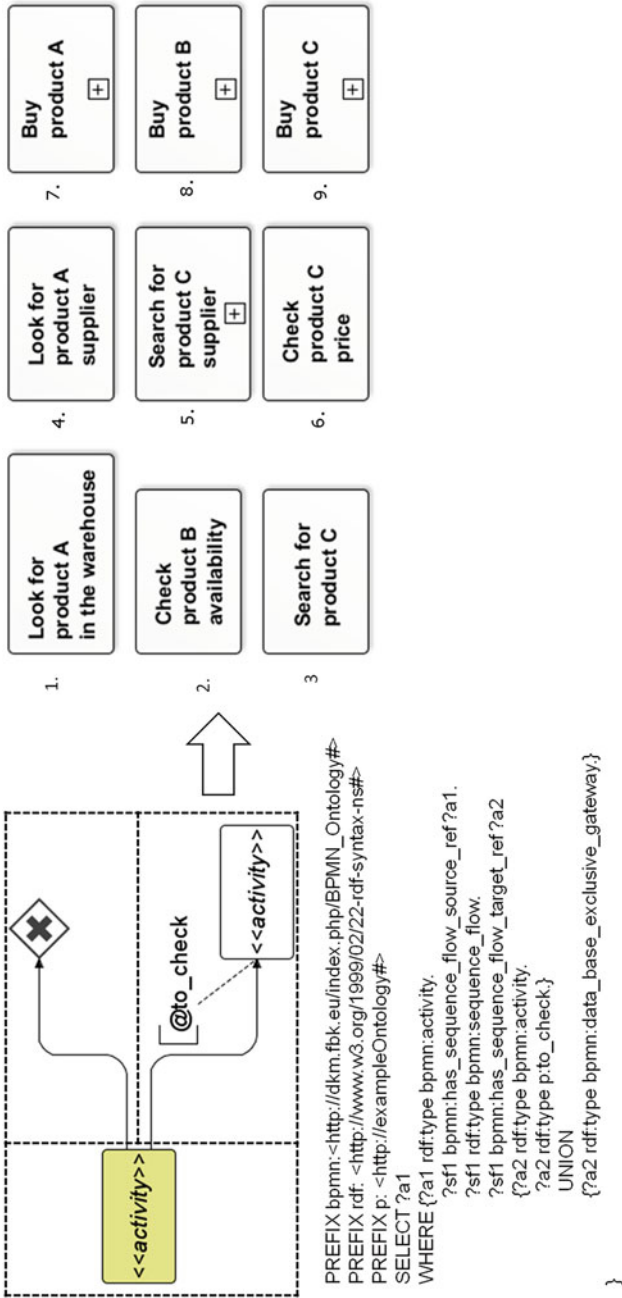
**Fig. 4** Query that uses semantic annotations and its four matches in the process model of Fig. 1. It queries for all the tasks that *check* something



**Fig. 5** Query that uses standard BPMN, stereotypes and semantic annotations and its unique match in the process model of Fig. 1. It queries for all the pairs of directly connected activities and their connecting sequence flow (see the dark background and the bold line style in the query), such that the source is a *search* activity



**Fig. 6** Query that uses logical operators and its three matches in the process model of Fig. 1. It queries for all the tasks that *check* something (see the dark background of the task in the query denoting the projection of the query), except for the product price



**Fig. 7** Query that uses the OR operator and its nine matches in the process model of Fig. 1. It queries for all the activities (see the dark background of the activity on the left) that are connected to a gateway or to a second activity that *checks* something

for *product C supplier* sub-process preceding the *Check product C price* task. The NOT operator is depicted as a cross over the negated (set of) BPD object(s). The query in Fig. 8, for example, looks for all the tasks that *check* something, but are not directly preceded by a gateway. Therefore, the three tasks that are instances of the BDO class *to\_check\_product\_availability* are discarded from the results, while only the *Check product C price* task is returned.

### Queries Using the Transitivity Operators

In order to ensure an easier way to navigate through BPD objects connected by one or more sequence flows or nested in sub-processes, two operators supporting transitivity have been introduced: the PATH and the NEST operators.

The PATH operator allows to match paths connecting two BPD flow objects at the same level of nesting. This operator is depicted as a BPMN sequence flow with two heads, symbolizing any intermediate graphical object (both flow objects and sequence flows) encountered along the path. We can think about a business scenario in which a business analyst is interested in understanding which are the purchase activities that are preceded sometime in the past by a check of the product availability. The query in Fig. 9, for example, asks for all the activities that *buy a product* and for which there exists a sequence flow path starting from a *to\_check\_product\_availability* activity. The result of this query, applied to the product assembly process, consists of the three sub-processes annotated by *to\_buy\_product*, since there exists at least a path from the tasks *Look for product A in the warehouse*, *Check for product B availability* and *Search for product C* to the three sub-processes *Buy product A*, *Buy product B* and *Buy product C*, respectively.

The NEST operator captures BPD graphical objects nested at any level of depth in sub-processes. It is depicted as a small oblique arrow in the upper right corner of the sub-process, with the head pointing to the external part of the sub-process. In case a business analyst, with the purpose of improving the performance of data storing sub-processes, is interested in understanding how many and which retrieval operations are carried out in these sub-processes, the query in Fig. 10 retrieves all the *to\_retrieve* tasks directly or indirectly contained in sub-processes storing purchase data. The result is provided by the task *Retrieve stored data* contained in the sub-process *Manage data storing*, in turn contained in the *to\_store\_purchase\_data* sub-process.

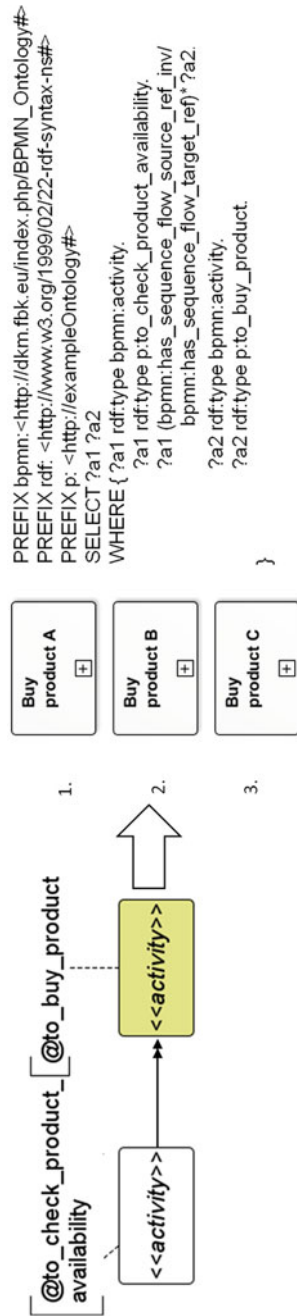
### Queries Using the DOR Operator

Sometimes it is useful to be able to express domain ontology relationships for querying specific business elements related to the process domain. To this end, BPMN VQL provides the DOR operator. It is depicted as a dashed arrow connecting graphical objects or semantic concepts and it represents a domain ontology relationship. In a business scenario in which, for statistical purposes, business analysts need to retrieve the information about product suppliers and corresponding products manipulated by the process activities, the query in Fig. 11 looks for all the pairs of instances of data objects, whose first component refers to supplier and whose second component concerns any of the supplied products. The provides relationship is a domain relation between the instances of the two semantic concepts: supplier





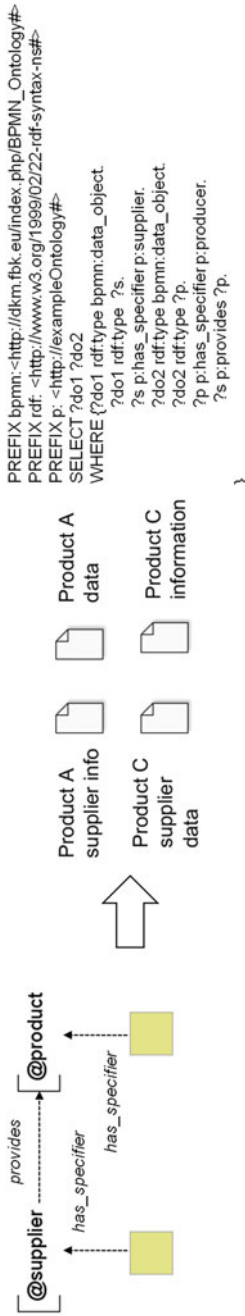
**Fig. 8** Query that uses the NOT operator and its unique match in the process model of Fig. 1. It queries for all the tasks (see the dark background of the task annotated with the concept to\_check) that check something and that are not preceded (via sequence flow) by a gateway



**Fig. 9** Query that uses the PATH operator and its three matches in the process model of Fig. 1. It queries for all the activities that *buy products* (see the dark background of the activity annotated with the concept *to\_buy\_product*) and for which there exists at least a path, consisting of sequence flows, that connects a *to\_check\_product\_availability* activity to such an activity



**Fig. 10** Query that uses the NEST operator and its unique match in the process model of Fig. 1. It queries for `to_retrieve` tasks (see the dark background) directly or indirectly contained in sub-processes that *store purchase data*



**Fig. 11** Query that uses the DOR operator and its four matches in the process model of Fig. 1. It queries for all the pairs of data object instances (see the dark background of the data objects) representing data of suppliers and products connected by a BDO provides relationship

and product. The other two DOR operators, both labeled `has_specifier`, represent the domain ontology relationships between the pairs of data objects' BDO classes (in this case derived from the BDO classes `supplier_data` and `product_data`) and their specifiers (`supplier` and `product`, respectively). In the example shown in Fig. 11, the two pairs of data objects (*Product A supplier info*, *Product A data*) and (*Product C supplier data*, *Product C information*) are returned as results.

## 4 Implementation and Evaluation

A framework enabling the execution of BPMN VQL queries has been implemented. In the next subsections, we detail the aforementioned framework implementation (Sect. 4.1), we report a quantitative evaluation of the performance of the framework (Sect. 4.2), and we provide a rigorous empirical qualitative evaluation of the BPMN VQL language (Sect. 4.3), in order to estimate whether it is easy to use for business analysts.

### 4.1 Implementation

The implementation<sup>2</sup> of the framework enabling the execution of BPMN VQL queries is based on three main components: *BPD Populator*, *Query Translator* and *Query Executor*.

*BPD Populator* is in charge of automatically translating the semantically annotated BPDs into the corresponding sets of axioms of the BPKB, as reported in Sect. 2. Once the BPD has been encoded in the BPKB, a reasoner is invoked.<sup>3</sup> The reasoner is able to infer new knowledge, starting from the asserted knowledge and from the terminological knowledge. For instance, by leveraging the fact that a BPMN task is a subclass of a BPMN activity (terminological knowledge), and the fact that a task  $t_1$  is an instance of the BPMNO class `task` (asserted knowledge), the reasoner is able to infer that  $t_1$  is also an instance of the class `activity`. Similarly, by leveraging the fact that the class `to_check_product_availability` is a subclass of the class `to_check`, and the fact that  $t_1$  is an instance of the BDO class `to_check_product_availability`, the reasoner can infer that  $t_1$  is also an instance of the class `to_check`. Although the inference step can be performed also at querying time, when a process model does not change often, an ontology preprocessing phase can be applied for precomputing an inferred process model starting from the knowledge

<sup>2</sup> The code is available at <https://drive.google.com/drive/folders/IR3-vFF8y8dXMokuLthx4xpLMpCtfuD1r>.

<sup>3</sup> In the current implementation, the Pellet reasoner (<http://clarkparsia.com/pellet/>) is used.

added to the BPKB. Such a preprocessing phase aims at saving time in the querying phase.

*Query Translator* deals with the automatic conversion of BPMN VQL queries into SPARQL (SPARQL Protocol and RDF Query Language) [8, 9] queries. SPARQL is an RDF-based query language standardized by the World Wide Web Consortium [8] and widely accepted in the semantic Web community (thus supported by several implementations). The SPARQL translation of the BPMN VQL queries is based on the BPMNO and BDO ontologies. The translation is obtained by: (i) requiring that each BPD graphical object in the visual query is an instance of the corresponding BPMNO class and each semantically annotated BPD object is an instance of the BDO class corresponding to the annotation (e.g., in Fig. 4, “*?t1 rdf:type bpmn :task*” and “*?t1 rdf:type p :to\_check*”, respectively); (ii) constraining the BPD graphical objects in the query according to the corresponding BPMN structural properties (e.g., “*?sf bpmn :has\_sequence\_flow\_source\_ref ?a1*” in Fig. 5); (iii) using FILTER and EXISTS SPARQL constructs for realizing the NOT operator and the SPARQL UNION construct for the OR operator; and (iv) using the SPARQL property paths for composing ontology properties and/or denoting their transitive closure (e.g., “*?a1 (bpmn :has\_sequence\_flow\_source\_ref\_inv/bpmn :has\_sequence\_flow\_target\_ref)\* ?a2*” in Fig. 9); (v) filling the SPARQL SELECT clause with variables representing the part of the query to be retrieved from the process (i.e., the dark or thick graphical objects and the semantic annotations in bold).

Finally, the *Query Executor* (a SPARQL implementation) is responsible for the execution of the queries over the BPKB populated with the BPD and its objects.

## 4.2 Performance Evaluation

Since the time spent for query answering is a critical factor for business analysts and designers, we have conducted an experiment to evaluate whether BPMN VQL queries have a reasonable response time.

In the experiment,<sup>4</sup> we considered six different process models of increasing size, with a number of process graphical elements ranging from 92 to 475, and, for each of them, a set of seven queries, each aimed at assessing a different construct of the BPMN VQL language. BPMN VQL queries were translated into SPARQL 1.1 queries and executed by means of the SPARQL ARQ implementation.<sup>5</sup> The purpose of the experiment was to study the performance of BPMN VQL as the size of the BPKB (in terms of instances) grows and as the structure (and hence the complexity) of the queries changes. The number of BPD graphical objects, the main characteris-

<sup>4</sup> The machine used for the experiment is a desktop PC with an Intel Core i7 2.80GHz processor, 6 Gb of RAM, and running Linux RedHat.

<sup>5</sup> <http://jena.sourceforge.net/ARQ/>.

**Table 1** BPMN VQL performance

	P1	P2	P3	P4	P5	P6
Process graphical objects	92	175	237	327	387	475
DL expressiveness	<i>ALC</i>	<i>ALC</i>	<i>AL</i>	<i>ALC</i>	<i>AL</i>	<i>ALC</i>
Classes	124	124	101	114	79	124
Class axioms	133	133	101	113	77	133
Ontology loading time (s)	1.445 (0.033)	1.476 (0.037)	1.498 (0.037)	1.513 (0.038)	1.528 (0.037)	1.564 (0.040)
Ontology preprocessing time (s)	4.459 (0.999)	8.318 (0.373)	13.090 (0.892)	15.298 (2.935)	37.237 (14.238)	35.349 (6.393)
Q1 (Fig. 4) (s)	0.003 (0.000)	0.004 (0.000)	0.012 (0.002)	0.006 (0.000)	0.012 (0.001)	0.008 (0.001)
Q2 (Fig. 5) (s)	0.004 (0.000)	0.004 (0.000)	0.006 (0.000)	0.005 (0.000)	0.006 (0.001)	0.006 (0.001)
Q3 (Fig. 6) (s)	0.017 (0.001)	0.019 (0.001)	0.020 (0.001)	0.021 (0.001)	0.021 (0.002)	0.023 (0.002)
Q4 (Fig. 7) (s)	0.011 (0.001)	0.014 (0.001)	0.018 (0.001)	0.022 (0.002)	0.025 (0.003)	0.030 (0.003)
Q5 (Fig. 8) (s)	0.003 (0.001)	0.004 (0.000)	0.004 (0.000)	0.005 (0.000)	0.005 (0.002)	0.006 (0.002)
Q6 (Fig. 9) (s)	0.009 (0.000)	0.010 (0.001)	0.009 (0.001)	0.011 (0.001)	0.012 (0.002)	0.012 (0.001)
Q7 (Fig. 10) (s)	0.003 (0.001)	0.003 (0.000)	0.005 (0.000)	0.004 (0.000)	0.005 (0.000)	0.005 (0.000)
Query average time (s)	0.0071	0.0083	0.0106	0.0106	0.0123	0.0129

tics and the Description Logic (DL) expressiveness of the domain ontologies<sup>6</sup> used to annotate the processes are listed in the top rows of Table 1, while a reference to an example query similar to those used in this experiment is shown within brackets, next to each query, in the first column of Table 1. In detail, the first query (*Q1*) looks for tasks of a given business domain type (see Fig. 4); the second (*Q2*) retrieves direct connections between pairs of flow objects, where the domain type of the first one is specified (see Fig. 5); the third (*Q3*) investigates the use of logical operators for the composition of semantic annotations (see Fig. 6); the fourth (*Q4*) makes use of the OR operator (similarly to the query in Fig. 7); the fifth (*Q5*) contains the NOT operator (see Fig. 8); the sixth (*Q6*) assesses the PATH operator (see Fig. 9); and, finally, the seventh (*Q7*) assesses the use of the NEST operator (see Fig. 10).

<sup>6</sup> Please refer to [1] for more details about the terminology used for denoting the expressive power of ontologies.

**Table 2** BPMN VQL performance on non-preprocessed ontologies

	P1	P2	P3	P4	P5	P6
Query average time (s)	0.1256	0.4769	0.9806	1.5899	3.0777	4.4373
Min query time (s)	0.034	0.083	0.138	0.206	0.295	0.411
Max query time (s)	0.292	0.891	1.817	2.922	5.763	8.492

The results of the experiment, i.e., the time required for the execution of each query on a single model, are reported at the bottom of Table 1. Time is expressed in seconds in the form  $avg(sd)$ , where  $avg$  and  $sd$  are respectively the arithmetic mean and the standard deviation of the execution times obtained over 100 runs on the same input query. Times related to query executions have been collected after an ontology preprocessing phase, in which the inferred model has been computed by the Pellet<sup>7</sup> reasoner (see Sect. 4.1). The preprocessing phase includes also the time for structure construction, required for the execution of the first query. As expected, not only the time required for loading the ontology increases when the process size grows (sixth row in Table 1), but also the time used for ontology preprocessing and for query execution, ranging from a query average time of 0.007 seconds for the process with the smallest size to a query average time of about 0.013 seconds for the process containing 475 process elements (see the last row in Table 1). On the contrary, the type of query does not significantly impact the performance of query execution, though minor differences among the considered types of queries exist. The largest amount of time was taken by the query that uses logical operators (both **and** and **not** operators) for the composition of semantic annotations ( $Q3$ ), and the one exploiting the **OR** operator for the composition of subqueries ( $Q4$ ). The “cheapest” queries, in terms of execution time, are  $Q5$  and  $Q7$ , i.e., the query using the **NOT** and the **NEST** operator, respectively. All types of queries, however, complete their run in a very limited time, though a quite significant time is spent for ontology preprocessing (around 35 s in case of the largest process). Ontology preprocessing is effective when the ontology is rarely modified. When, instead, frequent changes occur in the ontology, queries can be directly executed on the original ontology, on which no reasoning is applied before query execution. We collected the query execution performance also in this case and reported the average values in seconds in Table 2. As expected, query execution on non-preprocessed ontology takes more time than on the inferred ontology and execution time increases as the process model size grows. However, the average and the worst response times for querying a single model (4 and 8 s, respectively, for the largest process model among the ones considered) are still reasonable and compatible with activities involving human interaction.

Given these results, we can state that the use of the language for querying processes is compatible with business designers’ and analysts’ needs. Results

<sup>7</sup> <http://clarkparsia.com/pellet/>.



confirm the applicability of BPMN VQL as a means for supporting the analysts when retrieving relevant parts of a business process model.

### 4.3 Empirical Evaluation

In this subsection, we focus on the evaluation of effectiveness and efficiency, in terms of benefits gained and effort required, associated with BPMN VQL, used for retrieval and documentation purposes. In detail, we aim at comparing the advantages of the adoption of BPMN VQL with respect to a baseline approach, natural language, for retrieving and documenting information scattered across a process model. To this purpose, we conducted an experimental study with human subjects.

In the next subsections we first describe the goal and the design of the experiment (Sect. 4.3.1). We then report the experiment results (Sect. 4.3.2) and, finally, we discuss the results (Sect. 4.3.3).

#### 4.3.1 Experiment Definition, Planning, and Design

In this subsection, we describe the study by following the methodology presented by Wohlin [17].

##### Goal of the Study and Research Questions

The *goal* of the study is to analyze *two approaches*, one based on natural language queries and the other on BPMN VQL queries, with the *purpose* of evaluating query understandability and ease of query formulation.<sup>8</sup> The *quality focus* is the accuracy of the results obtained, the time spent in matching the queries against the process model and the subjective perception of the effort required during query understanding and execution. The *perspective* considered is of both researchers and business managers, interested in investigating the benefits of the adoption of a visual language for supporting business designers and analysts in retrieving parts of process models and documenting them. The *context of the study* consists of two objects (two semantically annotated processes and the ontologies used for their annotation) and a group of researchers and PhD students working at Fondazione Bruno Kessler (FBK), a research center in Trento, Italy, as subjects.

The objective of the study is: (i) comparing the understandability of BPMN VQL queries and of natural language (NL) queries; and (ii) evaluating the performance (in terms of results and effort required) of BPMN VQL queries with respect to NL for retrieving information. To this purpose, we asked the involved subjects to perform two different types of assignment: the *Query Understanding* and the *Query Execution* assignment. The *Query Understanding* assignment is aimed at

---

<sup>8</sup> According to Wohlin, the *goal* of an empirical study has to be defined by specifying: the *object(s)*, the *purpose*, the *quality focus*, the *perspective* and the *context of the study*.

comparing the ease of understanding queries in BPMN VQL and NL, and consists, for both languages, of matching the queries against the process model. The *Query Execution* assignment differs depending on the language. Since the purpose of the *Query Execution* assignment is to evaluate the time required by the humans to eventually achieve query execution, assuming that the starting point is a natural language description of the query we want to execute, the assignment consists of manual matching the query against the process, in case of NL, and in formulating BPMN VQL queries, to be automatically executed by a tool, in case of BPMN VQL.

We conjecture that the graphical notation of BPMN VQL queries, as well as their higher formality with respect to natural language, may help designers and analysts disambiguate and clarify the queries and correspondingly the relevant parts of the process that they represent. Moreover, we also expect that the task of formulating queries in BPMN VQL, which are then executed automatically, should be easier than matching NL queries against the process. These two expectations provide a direction for the research questions and the hypotheses we are interested in investigating:

**RQ1** *Are BPMN VQL queries easier to understand than natural language queries?*

**RQ2** *Is BPMN VQL query formulation easier to perform as compared to matching the results of natural language queries?*

**RQ1** deals with the understandability of BPMN VQL queries with respect to NL queries. The null hypothesis related to this question is: ( $H1_0$ ) *When performing query understanding tasks, understanding BPMN VQL queries is not easier than understanding NL queries.*

We investigated the first research question by taking into account and inspecting three different factors:

- the (objective) impact that query understanding has on the accuracy of the obtained results (we expect higher accuracy for BPMN VQL queries);
- the (objective) effort (time) required to perform query understanding tasks (for BPMN VQL queries we expect to observe a time not significantly higher than for NL queries);
- the perceived (subjective) effort required to perform query understanding tasks (we expect a lower perceived effort for BPMN VQL queries).

Hence,  $H1_0$  can be decomposed into the following three sub-hypotheses:

- ( $H1_{0A}$ ) *The results obtained by performing BPMN VQL query understanding tasks are not more accurate than those obtained when performing NL query understanding tasks;*
- ( $H1_{0B}$ ) *There is no difference between the time required to perform BPMN VQL and NL query understanding tasks;*
- ( $H1_{0C}$ ) *The effort perceived when performing BPMN VQL query understanding tasks is not lower than the one perceived when performing NL query understanding tasks.*

**RQ2** deals with the effort required for executing queries. In detail, relying on the assumption that queries are provided in natural language, **RQ2** deals with the formulation of BPMN VQL queries that can be then automatically matched against the process by means of our tool, compared to the manual matching of NL queries. Similarly to **RQ1**, the hypothesis for **RQ2** is: ( $H_{20}$ ) *When performing query execution tasks, formulating BPMN VQL queries is not easier than matching NL queries against the process model.* Also in this case, in order to deal with the research question, we considered and evaluated three main factors:

- the (objective) impact that query formulation/matching has on the accuracy of the results obtained by respectively formulating/matching the query in query execution tasks (we expect higher accuracy for BPMN VQL queries);
- the (objective) effort, in terms of time, required to perform query execution tasks (for BPMN VQL queries we expect a time not higher than NL queries);
- the perceived (subjective) effort required to perform query execution tasks (we expect a lower effort for BPMN VQL queries).

Similar to  $H_{10}$ ,  $H_{20}$  can be decomposed into three sub-hypotheses ( $H_{20A}$ ,  $H_{20B}$ , and  $H_{20C}$ ) corresponding to the investigated factors. Hence, the corresponding hypotheses in which  $H_{20}$  can be decomposed are the following:

- ( $H_{20A}$ ) *The results obtained by formulating BPMN VQL queries are not more accurate than those obtained by matching NL query matching in query execution tasks;*
- ( $H_{20B}$ ) *There is no difference between the time required to formulate BPMN VQL queries and matching NL queries in query execution tasks;*
- ( $H_{20C}$ ) *The effort perceived when formulating BPMN VQL queries is not lower than the one perceived when matching NL queries against the process in query execution tasks.*

## Context

The objects of the study are two semantically annotated business process models describing real-life procedures: *Bank Account Process* and *Mortgage Process*. The *Bank Account Process*<sup>9</sup> represents the exchange of information between the customer and the bank for opening and activating a bank account. It is composed of 2 pools (the *Bank* and the *Customer*) and contains 30 activities, 16 events and 16 gateways.<sup>10</sup> The associated ontology used for its annotation contains 77 concepts and 30 of them were used for process semantic annotation.<sup>11</sup> The *Mortgage*

<sup>9</sup> The *Bank Account Process* is based on a process used as example in the book by Havey [10] (it is reported in Havey's article "Modeling Orchestration and Choreography in Service Oriented Architecture" available at <http://www.packtpub.com/article/modeling-orchestration-and-choreography-in-service-oriented-architecture>).

<sup>10</sup> The interested reader can find the models online at <https://drive.google.com/drive/folders/IeFTwhCKUdLxI0ZijM4H2-PyHjAASOVb3>.

<sup>11</sup> The material is available online in the experimental package at <http://selab.fbk.eu/difrancescomarino/BPMNVQLEval/material/RepetitionPackage.zip>.

**Table 3** Study balanced design

	$L_1$		$L_2$	
	NL	BPMN VQL	NL	BPMN VQL
$G_A$	<i>Bank Account Process</i>			<i>Mortgage Process</i>
$G_B$		<i>Bank Account Process</i>	<i>Mortgage Process</i>	
$G_C$	<i>Mortgage Process</i>			<i>Bank Account Process</i>
$G_D$		<i>Mortgage Process</i>	<i>Bank Account Process</i>	

*Process*<sup>12</sup> is instead a process describing the procedure regulating the acceptance or the refusal by the “Mortgage Co.” company of mortgage requests formulated by potential customers. It is also composed of two pools (the *Mortgage Co.* and the *Potential Customer*) and it is slightly larger than the *Bank Account Process*: it contains 35 activities, 26 events and 18 gateways. The associated ontology has 99 concepts and 31 of them are used for semantically annotating the process.

The subjects involved in the study are 12 persons working at FBK in the domain of software engineering or knowledge management: 5 PhD students and 7 researchers.

**Design, Material, and Procedure**

The design adopted in this study is a *balanced design* [17]. Subjects are divided into four groups ( $G_A$ ,  $G_B$ ,  $G_C$ , and  $G_D$ ) and asked to perform two types of assignment (*Query Understanding* and *Query Execution*) on two different objects (*Bank Account Process* and *Mortgage Process*) with two *treatments* (NL or BPMN VQL queries) in two laboratory sessions ( $L_1$  and  $L_2$ ). Each group worked with both treatments and with both objects, by performing both the *Query Understanding* and *Query Execution* assignment, on one process with NL queries in one laboratory and on the other process with BPMN VQL queries in the other laboratory. The schema adopted in the study is reported in Table 3. Such a schema allows us to limit the impact of the learning effect on the objects and to limit the interactions between learning and treatment. During the experiment, subjects received the following material<sup>13</sup> to perform the required tasks: a pre-questionnaire collecting information about knowledge and experience of subjects; a BPMN quick handbook; a BPMN VQL handbook; a semantically annotated process (the *Bank Account Process* or the *Mortgage Process*); the domain ontology used for annotating the process; an extract of the BPMN ontology for clarifying relationships among BPMN constructs; a description of the tasks to perform; the *answer book* (a set of 10 sheets) for reporting the answers related to the 10 required tasks; a post-questionnaire investigating personal judgments of the subjects about tasks and process; a final

<sup>12</sup> The *Mortgage Process* is based on one of the process models used as running examples in the BPMN book by White et al. [16].

<sup>13</sup> The experimental package, containing the material used in the experiment, is available online at <http://selab.fbk.eu/difrancescomarino/BPMNVQLEval> for repetition purposes.

post-questionnaire collecting subjective judgments about the BPMN VQL benefits versus effort. In detail, the description of the tasks to perform consists of:

- 10 queries: 6 NL queries to be matched against the process for the *Query Understanding* assignment and 4 NL queries to be matched against the process for the *Query Execution* assignment; or
- 10 queries: 6 BPMN VQL queries to be matched against the process for the *Query Understanding* assignment and 4 queries in natural language to be translated into BPMN VQL queries for the *Query Execution* assignment;

Before the experiment execution, subjects were trained in seminars on BPMN, ontologies, semantic annotation of business process models and BPMN VQL. Moreover, subjects were also provided with a description of the two process models in order to let them familiarize with the domain.

After the training session, in the first laboratory, subjects were asked to fill a pre-questionnaire. Then, for both assignments, subjects were asked to mark the starting time before executing each task and the ending time after the task execution. The *Query Understanding* consists, for both treatments, of matching the queries against the process. The *Query Execution* for NL queries differs from BPMN VQL *Query Execution*. The reason is that our goal is to evaluate, for each of the two approaches, the effort required for executing the query. Hence, in case of NL queries, we have to evaluate the effort required for matching the NL query, while, in case of BPMN VQL queries, we have to evaluate the effort required for formulating the BPMN VQL query, assuming that the query is then executed by an automatic tool at no further cost for the user. An additional benefit of automatic query execution (not evaluated in this study), is that by visualizing the retrieved results, the query tool can reveal possible false positives/negatives associated with the formulated query, hence supporting users in successive query refinement.

The use of the tool in the experiment, however, would not have been completely fair with respect to manual natural language matching, because the manual matching process cannot reliably point to false positives/negatives. Hence, we decided to partially penalize subjects involved in the BPMN VQL treatment. We allowed them to match the formulated BPMN VQL query, in order to verify its correctness, only manually. We asked them to refine it, if necessary, and to record separately the percentage of time spent in matching the query vs refining the query. At the end of both laboratory sessions, subjects were asked to fill the final post-questionnaire.

### Variables

The independent variable considered in this study is the type of query language used for performing the assignments. The independent variable, hence, can assume only one of two values, i.e., the two treatments: NL or BPMN VQL.

The number of dependent variables in the study is higher, since for the evaluation of the two research questions we analyzed both objective and subjective factors. In

**Table 4** Dependent variable description

Hp	Sub-hp	Det. sub-hp	Variable	Unit/Scale	Description
$H1_0$	$H1_{0A}$	$H1_{0A_P}$	$P_{QU}$	[0, 1]	Precision
		$H1_{0A_R}$	$R_{QU}$	[0, 1]	Recall
		$H1_{0A_{FM}}$	$FM_{QU}$	[0, 1]	f-measure
	$H1_{0B}$	$H1_{0B_T}$	$T_{QU}$	sec.	Time
	$H1_{0C}$	$H1_{0C_{PEQU}}$	$PEQU$	[0, 4]	Perceived effort in query understanding
		$H1_{0C_{PEOU}}$	$PEOU$	[0, 4]	Perceived effort in ontology understanding
		$H1_{0C_{PEOM}}$	$PEQM$	[0, 4]	Perceived effort in query matching
$H2_0$	$H2_{0A}$	$H2_{0A_P}$	$P_{QE}$	[0, 1]	Precision
		$H2_{0A_R}$	$R_{QE}$	[0, 1]	Recall
		$H2_{0A_{FM}}$	$FM_{QE}$	[0, 1]	f-measure
	$H2_{0B}$	$H2_{0B_T}$	$T_{QE}$	sec.	Time
	$H2_{0C}$	$H2_{0C_{PEQE}}$	$PEQE$	[0, 4]	Perceived effort in query execution
		$H2_{0C_{PESU}}$	$PESU$	[0, 4]	Perceived effort in specification understanding

detail, we used the accuracy of the results of the *Query Understanding* and *Query Execution* assignments as well as the time spent to perform the tasks as objective measures; the personal judgments expressed by subjects about the effort required by each task as subjective measures.

The set of dependent variables defined to answer the two research questions, as well as the corresponding descriptions, are reported in Table 4. For each of the two hypotheses,  $H1_0$  and  $H2_0$  (answering research questions **RQ1** and **RQ2**, respectively), the related sub-hypotheses have been considered (column “Sub-hp” in Table 4). In turn, each sub-hypothesis has been further decomposed, according to the different measures (e.g., precision, recall) considered for its evaluation, into detailed sub-hypotheses (column “Det. sub-hp” in Table 4), each corresponding to a dependent variable (column “Variable” in Table 4).

We evaluated the results obtained in the *Query Understanding* assignment in order to investigate the query language understandability **RQ1** and the results obtained in the *Query Execution* assignment for **RQ2**. For the evaluation of the accuracy of the task results, we exploited two metrics widely used in Information Retrieval: precision and recall. In case of the *Query Understanding* assignment, for each subject  $s_j$  and for each query  $q_i$ , we identified the set of correct results ( $CR_{q_i}$ ) of the query  $q_i$  and the set of results reported by subject  $s_j$  for query  $q_i$  ( $RR_{q_i,s_j}$ ). In case of the *Query Execution* assignment,  $CR_{q_i}$  and  $RR_{q_i,s_j}$  are identified for each NL query  $q_i$  and subject  $s_j$  exactly as above, while the corresponding values, for each task  $t_i$  involving the BPMN VQL treatment, are collected by automatically executing the corresponding BPMN VQL query  $q_i$  formulated by subject  $s_j$ .

Starting from these values we computed precision  $P_{q_i, s_j}$ , recall  $R_{q_i, s_j}$  and F-Measure  $FM_{s_j, q_i}$  for each query  $q_i$  and for each subject  $s_j$  as follows:

$$P_{q_i, s_j} = \frac{|CR_{q_i} \cap RR_{q_i, s_j}|}{|RR_{q_i, s_j}|} \quad (1)$$

$$R_{q_i, s_j} = \frac{|CR_{q_i} \cap RR_{q_i, s_j}|}{|CR_{q_i}|} \quad (2)$$

$$FM_{q_i, s_j} = \frac{2 * P_{q_i, s_j} * R_{q_i, s_j}}{(P_{q_i, s_j} + R_{q_i, s_j})} \quad (3)$$

Since in the performed study the size of the set of correct results is not the same for all the queries and each query is read and interpreted independently by subjects, we chose to evaluate each query separately, by computing precision and recall for each of them, thus avoiding to penalize and emphasize too much possible misunderstandings/perfect understandings in the specification of queries with a high number of correct results.

In order to get a global result for each assignment  $k$  (i.e.,  $k \in \{Query\ Understanding, Query\ Execution\}$ ), and for each subject  $s_j$ , we computed the average of the three values ( $P_{k, s_j}$ ,  $R_{k, s_j}$  and  $FM_{k, s_j}$ ) over all the queries in the set of queries for assignment  $k$  ( $Q_k$ ), that is:

$$P_{k, s_j} = \frac{\sum_{q_i \in Q_k} P_{q_i, s_j}}{|Q_k|} \quad (4)$$

$$R_{k, s_j} = \frac{\sum_{q_i \in Q_k} R_{q_i, s_j}}{|Q_k|} \quad (5)$$

$$FM_{k, s_j} = \frac{\sum_{q_i \in Q_k} FM_{q_i, s_j}}{|Q_k|} \quad (6)$$

In the objective evaluation, we also considered the time spent for completing each assignment. In case of **RQ1**, the time required for performing the *Query Understanding* assignment is collected. In detail, for each query  $q_i$  the time spent by the subject  $s_j$  ( $T_{q_i, s_j}$ ) is computed by considering both the time spent for reading the NL/BPMN VQL query and for retrieving the query results against the process. For **RQ2**, it is the time required for performing the *Query Execution* task: in case of the NL treatment,  $T_{q_i, s_j}$  represents the time spent by subject  $s_j$  for retrieving the results of the query  $q_i$ . In case of BPMN VQL queries,  $T_{q_i, s_j}$  is the time spent by subject  $s_j$  for formulating and, if necessary, refining the query. The time spent in

matching the query is, however, excluded from  $T_{q_i, s_j}$  because, in principle, it can be performed automatically.

As for Information Retrieval metrics, the average value per assignment type  $k \in \{Query\ Understanding, Query\ Execution\}$  and per subject  $s_j$  has been computed:

$$T_{k, s_j} = \frac{\sum_{i \in Q_k} T_{s_j, q_i}}{|Q_k|} \quad (7)$$

With respect to the subjective evaluation, a set of answers was collected through the post-questionnaire. In detail, each subject was asked to express her evaluation on a 5-point Likert scale (from 0 to 4, where 0 is very low and 4 is very high) about the perceived effort in query understanding ( $PEQU_{s_j}$ ), ontology understanding ( $PEOU_{s_j}$ ), query execution ( $PEQE_{s_j}$ ) and specification understanding ( $PESU_{s_j}$ ).

### 4.3.2 Experimental Results

Due to the violation of the preconditions for parametric statistical tests (small number of data points and non-normal distribution), we decided to apply a non-parametric test to compare the distributions of data obtained with the two different treatments. Moreover, since subjects performed the assignments with both NL and BPMN VQL treatment, we could perform a paired statistical test. Starting from these considerations, we resorted to the Wilcoxon test [17], a non-parametric paired test. Then, depending on the direction of the hypotheses to verify, i.e., whether a direction was already present in the hypothesis ( $H1_{0A}, H1_{0C}; H2_{0A}, H2_{0C}$ ) or not ( $H1_{0B}; H2_{0B}$ ), we opted for a one-tailed or two-tailed analysis, respectively.

In order to evaluate the magnitude of the statistical significance obtained, we computed also the *effect size*, which provides a measure of the strength of the relationship between two variables. To this purpose we used the *Cohen's d* formula [2] (the effect size is considered to be small for  $0.2 \leq d < 0.5$ , medium for  $0.5 \leq d < 0.8$  and large for  $d \geq 0.8$ ).

The analyses are performed with a level of confidence of 95% (p-value < 0.05), i.e., there is only a 5% of probability that the observed differences are due to chance.

#### Research Question 1

Table 5 reports the descriptive statistics of the data related to **RQ1**, i.e., the *Query Understanding* assignment of the study. Figure 12a reports the boxplots of precision, recall, and F-Measure for the same assignment. We can notice that the values of precision, recall, and F-Measure obtained in case of BPMN VQL queries are higher than those obtained in case of NL queries. However, while in case of precision and F-Measure, the first quartile for BPMN VQL queries is very far from the first quartile of NL queries, the two values are much more closer in case of recall.

As shown in Table 6, two out of the three null sub-hypotheses related to  $H1_{0A}$  (i.e.,  $H1_{0Ap}$  and  $H1_{0AFM}$  in Table 4) can be rejected. In detail, though we are not able to reject the null hypothesis  $H1_{0AR}$  (corresponding to the variable  $RQU$ ), we



**Table 5** Descriptive statistics for the *Query Understanding* assignment

Det. Sub-hp	Variable	Mean		Median	
		NL	BPMN VQL	NL	BPMN VQL
$H1_{0A_P}$	$P_{QU}$	0.870349326	0.978505291	0.916666667	1
$H1_{0A_R}$	$R_{QU}$	0.92037	0.951852	0.958333	1
$H1_{0A_{FM}}$	$FM_{QU}$	0.861113	0.951918	0.883333	0.974074
$H1_{0B_T}$	$T_{QU}$	196	202	175	189
$H1_{0C_{PEOU}}$	$PEQU$	2.083333333	1.08333333	2	1
$H1_{0C_{PEOU}}$	$PEOU$	1.583333333	1	1.5	1
$H1_{0C_{PEOM}}$	$PEQM$	1.833333333	1.33333333	1.5	1

are able to reject  $H1_{0A_P}$  (p-value = 0.02959) and  $H1_{0A_{FM}}$  (p-value = 0.04118), both with a medium effect-size value. Overall, by considering the F-Measure as a global measure of the query answers and hence  $H1_{0A_{FM}}$  as the main detailed sub-hypothesis, we can reject  $H1_{0A}$ , i.e., we can affirm that the use of BPMN VQL allows to get more accurate results than NL.

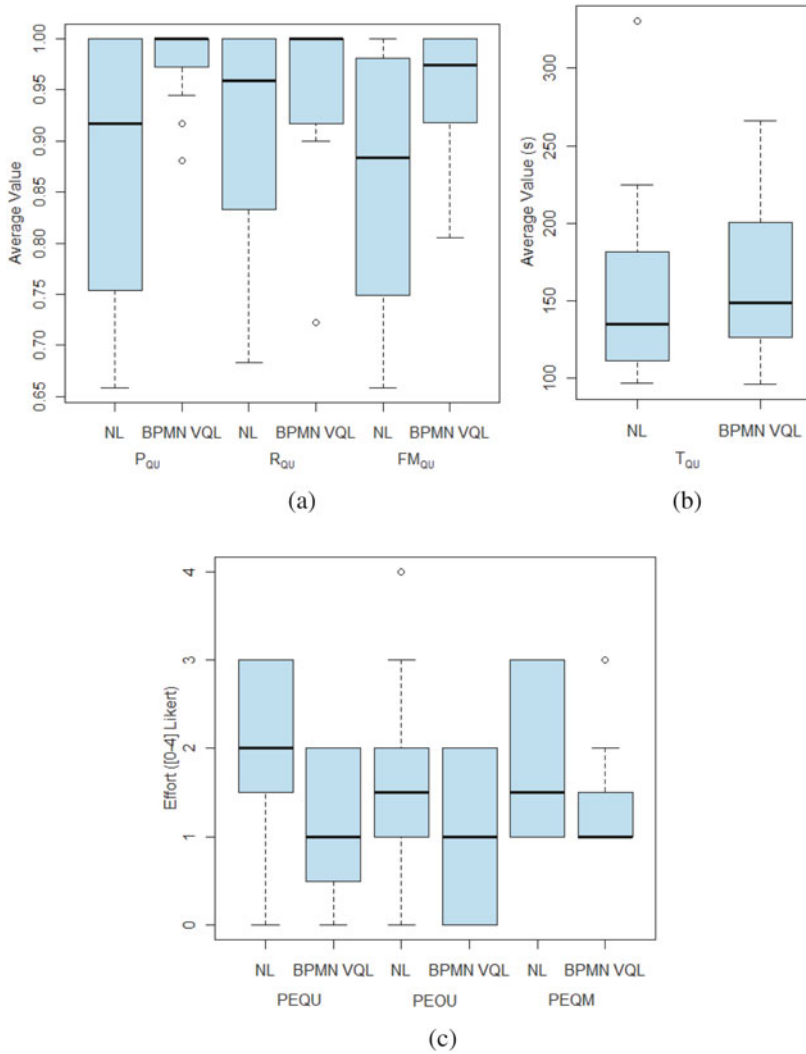
Figure 12b depicts the boxplot of the time spent for the *Query Understanding* assignment. In this case, for the BPMN VQL queries, the values of the time spent in understanding BPMN VQL queries are only slightly higher than the time spent for understanding NL queries. We applied a two-tailed Wilcoxon test for investigating the null hypothesis  $H1_{0B}$ , related to the time. In this case, we were not able to reject the two-tailed null hypothesis (p-value = 0.8501 in Table 6), hence we cannot affirm that there is a difference of effort, in terms of time, to perform the *Query Understanding* assignment in case of NL versus BPMN VQL queries.

Finally, Fig. 12c reports the boxplots of the perceived effort required for understanding the queries, understanding the ontology used for process annotation and matching the queries against the process, both with NL and BPMN VQL. In all three subjective ratings of the perceived effort with BPMN VQL, the boxplot is mostly concentrated in the bottom part of the plot (i.e., in the interval [0, 2]), meaning that subjects perceived a low/medium effort. The corresponding NL values, instead, are higher. We applied a one-tailed Wilcoxon paired test in order to investigate the sub-hypothesis  $H1_{0C}$  for all three dependent variables. The results allowed us to reject each of the null hypotheses (see Table 6) and hence the whole  $H1_{0C}$ , i.e., the effort perceived in understanding BPMN VQL queries is lower than the one perceived in understanding NL queries. In the specific case of query understanding, the result related to the perceived effort ( $PEQU$ ) is particularly strong, being associated with a large Cohen  $d$  effect size ( $d = 0.886$ ).

## Research Question 2

The descriptive statistics of the data related to the query execution assignment are reported in Table 7, while the corresponding boxplots are reported in Fig. 13.

Figure 13a shows that the boxplots related to precision, recall, and F-Measure obtained by automatically executing the BPMN VQL queries formulated by subjects



**Fig. 12** Boxplots for the *Query Understanding* assignment. (a) Precision, recall and F-Measure. (b) Time. (c) Query and ontology understanding, query matching perceived effort

are squeezed into the maximum possible value, of one. On the contrary, NL boxplots are spread across the interval [0.8, 1]. BPMN VQL values for precision, recall, and F-Measure are substantially higher than the values obtained by manual matching of NL queries. Such visual impression is confirmed by the one-tailed Wilcoxon test, whose results are reported in Table 8. In all three cases, we were able to reject, with a level of confidence of 95%, the null hypothesis related to the specific dependent variable. Moreover, in case of recall and F-Measure, the result is strengthened by

**Table 6** Paired analysis for the *Query Understanding* assignment. Statistically significant p-values ( $< 0.05$ ) and large values of effect size ( $> 0.8$ ) are reported in bold

Det. Sub-hp	Variable	Wilcoxon p-value	Cohen d
$H1_{0AP}$	$P_{QU}$	<b>0.02959</b>	0.698703
$H1_{0AR}$	$R_{QU}$	0.1308	
$H1_{0AFM}$	$FM_{QU}$	<b>0.04118</b>	0.698885
$H1_{0BT}$	$T_{QU}$	0.8501	
$H1_{0CPEQU}$	$PE_{QU}$	<b>0.009864</b>	<b>0.8864053</b>
$H1_{0CPEOU}$	$PE_{OU}$	<b>0.02386</b>	0.6479058
$H1_{0CPEQM}$	$PE_{QM}$	<b>0.04734</b>	0.6267832

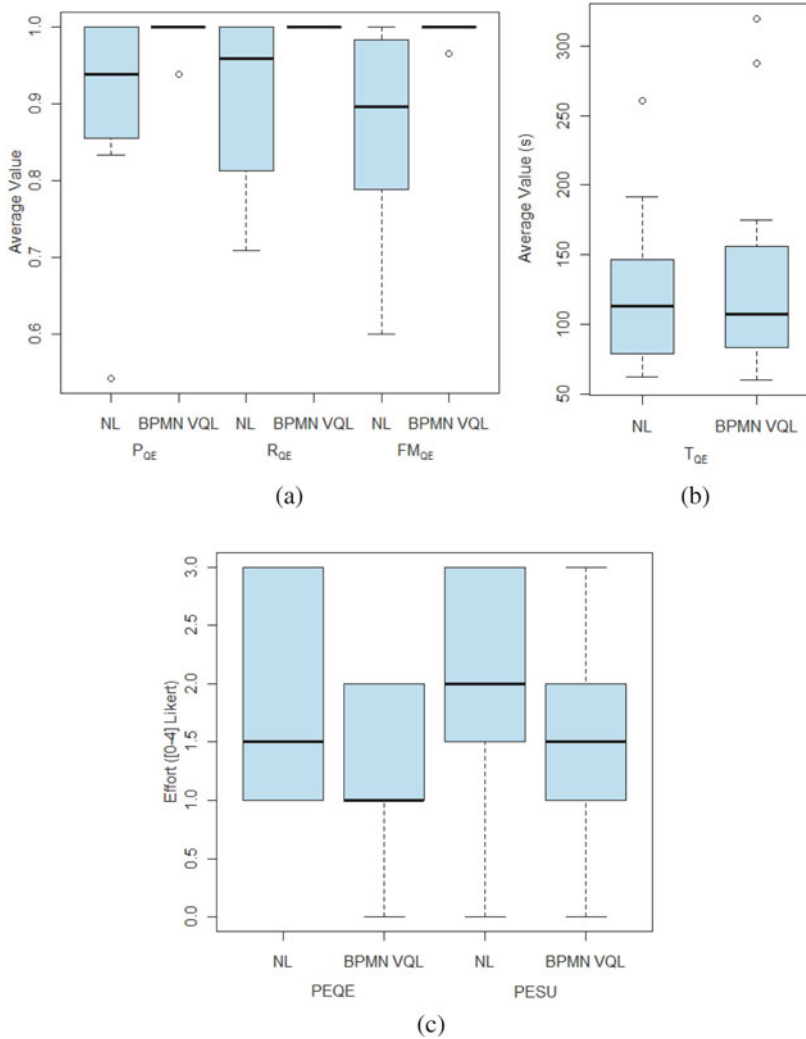
**Table 7** Descriptive statistics of the *Query Execution* assignment

Det. Sub-hp	Variable	Mean		Median	
		NL	BPMN VQL	NL	BPMN VQL
$H2_{0AP}$	$P_{QE}$	0.90625	0.989583	0.9375	1
$H2_{0AR}$	$R_{QE}$	0.90625	1	0.958333	1
$H2_{0AFM}$	$FM_{QE}$	0.869742	0.994048	0.895833	1
$H2_{0BT}$	$T_{QE}$	125	138	113	108
$H2_{0CPEQE}$	$PE_{QE}$	1.833333333	1.333333333	1.5	1
$H2_{0CPESU}$	$PESU$	2.083333333	1.583333333	2	1.5

a high value of the Cohen  $d$  measure. Overall, the  $H2_{0A}$  sub-hypothesis can be rejected, i.e., the results obtained by automatically executing BPMN VQL queries formulated manually by subjects are much more accurate than those obtained by manually matching NL queries against the process.

By looking at Fig. 13b, we can observe that the time spent for formulating and refining the BPMN VQL queries is almost the same as the time spent for matching the NL queries against the process. In Table 8, we can also find statistical evidence for this visual observation. In fact, we were not able to reject the two-tailed null sub-hypothesis  $H2_{0B}$ , i.e., we cannot affirm that there is a difference in the time spent to match NL queries against the process vs. the time spent for translating them into BPMN VQL.

For what concerns the effort perceived when executing the queries (i.e., either matching the NL queries or formulating BPMN VQL queries), in Fig. 13c we can observe that, differently from the NL boxplot, in case of BPMN VQL queries, the boxplot is mainly distributed around the lower half of the Likert scale. The shapes of the NL vs. BPMN VQL boxplots (two rightmost boxplots) for the perceived understanding effort follow a similar pattern. By applying a one-tailed Wilcoxon test, we were not able to reject the null hypothesis related to the perceived effort in query execution (p-value = 0.07023), but we are able to reject the null hypothesis related to the perceived specification understanding effort (p-value = 0.02054)



**Fig. 13** Boxplots for the *Query Execution* assignment. (a) Precision, recall and F-Measure. (b) Time. (c) Query execution and task specification understanding perceived effort

**Table 8** Paired analysis for the *Query Execution* assignment. Statistically significant p-values ( $< 0.05$ ) and large values of effect size ( $> 0.8$ ) are reported in bold

Det. Sub-hp	Variable	Wilcoxon p-value	Cohen d
$H2_{0A_P}$	$P_{QE}$	<b>0.02099</b>	0.588606
$H2_{0A_R}$	$R_{QE}$	<b>0.01802</b>	<b>0.8436</b>
$H2_{0A_{FM}}$	$FM_{QE}$	<b>0.007001</b>	<b>0.903492</b>
$H2_{0B_T}$	$T_{QE}$	0.6377	
$H2_{0C_{PEQE}}$	$PEQE$	0.07023	
$H2_{0C_{PESU}}$	$PESU$	<b>0.02054</b>	0.7416198

### 4.3.3 Discussion

The analysis shows that, on average, the results obtained when matching BPMN VQL queries are more precise, more complete and, hence, more accurate than those obtained when manually matching NL queries ( $H_{10A}$ ). Although the result obtained for recall shows only a trend, being not statistically relevant ( $p$ -value of  $R_{QU} > 0.05$ ), the one obtained for precision clearly shows that BPMN VQL allows business analysts to be more precise in understanding requests related to semantically annotated processes. This finding could be explained by the higher precision of BPMN VQL with respect to NL. Indeed, BPMN VQL is able to precisely capture a BPMN business process description enriched with semantic annotations.

Moreover, the positive result obtained for the accuracy of the answers is not excessively penalized by the required effort ( $H_{10B}$ ). On average, the time spent for performing the *Query Understanding* assignment in case of BPMN VQL queries is only 3.8% more than the time spent for performing the same assignment in case of NL queries, as shown in Fig. 12b. However, such slight extra time is not perceived as an additional effort by subjects. As shown in Table 8, in fact, the effort perceived in BPMN VQL query understanding is lower ( $p$ -value  $< 0.05$ ) than the one perceived for understanding NL queries. Moreover, results show that also the perceived effort for understanding the ontology, as well as for matching the query, is significantly lower ( $p$ -value  $< 0.05$ ) for BPMN VQL queries than for NL queries. The former result could be due to the formal structure in which ontologies are organized, which is closer to BPMN VQL than to NL. Similarly, we can speculate that having in mind a graphical representation of the pattern to look for, as well as having a clear and formal description of the semantics of the searched pattern components, could relieve the effort required by the matching task. By looking at Fig. 12c we can also note that, when matching NL queries against the process, the most expensive aspect of the activity seems to be query understanding (on average 2.08 in the Likert scale, i.e., the perceived effort is slightly higher than the “medium” effort), followed by query matching (on average 1.83 in the [0, 4] scale). This observation is in-line with the qualitative answers provided by subjects. The main difficulties found by subjects in understanding NL queries, in fact, are the ambiguity and the lack of precision of the natural language, as well as the difficulty in mapping natural language to structural properties of the process (5 subjects). When matching BPMN VQL queries, instead, the activity requiring more effort is the actual query matching (on average 1.33 in the Likert scale), that is, however, lower, on average, than the effort required by NL query matching ( $H_{10C}$ ). On the other hand, the difficulties encountered by subjects in query understanding are not common to more than one subject and they tend to differ from subject to subject.

By considering the collected results and taking into account the different factors analyzed, we can hence affirmatively answer the research question **RQ1**: understanding BPMN VQL queries is easier than understanding NL queries.

With respect to the research question **RQ2**, we found several statistically relevant results. The results obtained when automatically executing BPMN VQL queries formulated by subjects are not only more specific than those obtained by manually

executing NL queries, but they are also more sensitive and correspondingly more accurate ( $H2_{0A}$ ).

The good results obtained in terms of accuracy are not penalized by the associated time performance ( $H2_{0B}$ ). The time spent for formulating BPMN VQL queries, in fact, is, on average, just slightly higher than the time spent for matching NL queries (around 10% more). However, the high average time of BPMN VQL queries is mainly due to two outliers. As shown in Table 7, in fact, the median value of the BPMN VQL query formulation time is lower than the median time required for matching NL queries. Since the time required for BPMN VQL query formulation remains unchanged as the process size increases and the time spent for automatic BPMN VQL query execution increases only slightly (a few milliseconds), the BPMN VQL approach scales better (with respect to its time performance) than the NL approach, which is dominated by the manual query matching time.

Other interesting findings are related to the perceived effort in the *Query Execution* assignment. Although results are not statistically significant, the effort perceived in formulating BPMN VQL queries is overall lower (on average 27% lower) than the effort required for matching NL queries, as shown in the boxplots in Fig. 13c. Moreover, the perceived effort required for understanding natural language specifications seems to be positively influenced by the type of activity to be executed: understanding natural language specifications with the aim of transforming them into BPMN VQL queries is perceived as easier than understanding the same specifications with the aim of matching them against the process. Moreover, by inspecting the boxplots in Fig. 13c, we can observe that, as for the *Query Understanding* assignment, most of the effort is spent in understanding the natural language, both for BPMN VQL and NL tasks. The perceived effort required for understanding natural language specifications is higher (in the same Likert scale) than the effort perceived when formulating BPMN VQL queries or matching NL queries against the process ( $H2_{0C}$ ).

The qualitative answers related to the main difficulties faced in formulating BPMN VQL queries for this assignment mainly concern the lack of the use of a tool for inspecting the ontology (reported by three subjects) and the poor experience with BPMN VQL (three subjects). These answers seem to confirm that results could be further improved with tool availability and with more practice on BPMN VQL.

By taking into account these observations, we can, hence, also provide an affirmative answer to **RQ2**.

As confirmed by the answers given to the last question in the post-questionnaire, we can conclude that, overall, the proposed BPMN VQL language is easier to understand than natural language and makes it easier to retrieve results scattered across process models, i.e., it is better suited for solving the query task, thus providing good support to business designers and analysts for documenting as well as retrieving specific information scattered across large process models.

## 5 Framework

This section maps BPMN VQL into the Process Querying Framework (PQF) [13].

The prerequisite for being able to make queries in BPMN VQL is the existence, in the *Model, Simulate, Record, and Correlate* part of the framework, of (at least) one process model described in the BPMN language and semantically annotated with concepts from a domain ontology BDO (see Sect. 2). The semantically annotated process model is then transformed into the BPKB (see Sect. 2). This activity, which is performed by the *BPD Encoder* and focuses on the encoding of the business process diagram into the RDF format, can be seen as a particular active component of the *Model, Simulate, Record, and Correlate* part of PQF, whose passive output will be the BPKB containing the RDF representation of the BPMN process model, i.e., an RDF version of the process repository.

BPMN VQL queries implement a *read intent*, i.e., they do not create new process models or update existing ones. Rather, BPMN VQL queries return parts of process models (i.e., the *selection pattern* described in Sect. 3) matching the formulated query (i.e., the *matching criterion*). Once queries have been formulated in the BPMN VQL language, they are formalized in the corresponding SPARQL queries to be executable on the BPKB. Passive components of the PQF's *Model, Simulate, Record, and Correlate* part (query intent, query condition and process querying instruction), as well as the active query formalization component (corresponding to the *Query Translator* module), are hence implemented in BPMN VQL.

Before being queried, an active component of the *Prepare* part of PQF pre-processes the BPKB, so as to make it faster to be queried. In detail, the reasoner of the *BPD Encoder* enriches the knowledge base by materializing inferences (see Sect. 3). In this way, at query time, no time-costly reasoning is required and the query response time is not high. The passive output of this step is the BPKB enriched with the materialized triples.

In the *execute* part, the process query is transformed to SPARQL by the *Query Translator* and executed by the *Query executor* on the BPKB containing the process model diagram description and the inferred triples (see Sect. 4.1). The result of process querying is a set of RDF triples representing the parts of the process model matching the query.

Finally, in the *Interpreter* phase, query results could be visualized as parts of the BPMN process model.

## 6 Conclusion and Future Work

In this chapter, we presented BPMN VQL, a visual language for automated querying of semantically annotated business process models. While aiming to provide reasonable response time, BPMN VQL also aims at being easy to use for business people familiar with business process modeling languages, such as BPMN.

A performance evaluation and an empirical study with human subjects have shown that BPMN VQL queries can be executed online and that the language is easier to use than the natural language.

In our future work, we plan to improve the implementation, for instance, by exploiting recent and optimized RDF repositories and reasoners for query execution [7]. It would be important to replicate the empirical study by involving a higher number of subjects, including real business process analysts. We are also interested in investigating the effort required to perform the semantic annotation of business process models.

## References

1. Baader, F., Calvanese, D., McGuinness, D.L., Nardi, D., Patel-Schneider, P.F.: The Description Logic Handbook: Theory, Implementation and Applications, 2nd edn. Cambridge University Press, New York, NY, USA (2010)
2. Cohen, J.: Statistical power analysis. *Curr. Dir. Psychol. Sci.* **1**(3), 98–101 (1992). <https://doi.org/10.1111/1467-8721.ep10768783>
3. Di Francescomarino, C., Ghidini, C., Rospocher, M., Serafini, L., Tonella, P.: Reasoning on semantically annotated processes. In: Bouguettaya, A., Krueger, I., Margaria, T. (eds.) *Service-Oriented Computing – ICSOC 2008*, pp. 132–146 (2008)
4. Di Francescomarino, C., Ghidini, C., Rospocher, M., Serafini, L., Tonella, P.: Semantically-aided business process modeling. In: Bernstein, A., Karger, D.R., Heath, T., Feigenbaum, L., Maynard, D., Motta, E., Thirunarayan, K. (eds.) *8th International Semantic Web Conference (ISWC 2009)*, LNCS, vol. 5823/2009, pp. 114–129 (2009)
5. Di Francescomarino, C., Tonella, P.: Crosscutting concern documentation by visual query of business processes. In: Ardagna, D., Mecella, M., Yang, J. (eds.) *Business Process Management Workshops*, pp. 18–31 (2009)
6. Filman, R.E., Friedman, D.P.: Aspect-oriented programming is quantification and obliviousness. *Tech. rep.* (2000)
7. Franz Inc.: AllegroGraph RDFStore Web 3.0's database. <https://www.franz.com/agraph/allegrograph/>. Accessed: January 2019
8. Group, R.D.A.W.: SPARQL query language for RDF. W3C recommendation, W3C (2008). <https://www.w3.org/TR/2008/REC-rdf-sparql-query-20080115/>
9. Group, S.W.: Sparql new features and rationale. Web page (2009). <https://www.w3.org/TR/2010/WD-sparql11-query-20100126/>
10. Havey, M.: SOA Cookbook. From technologies to solutions. Packt Pub. (2008). <https://books.google.it/books?id=GNOC1GWPsIoC>
11. Hepp, M., Leymann, F., Domingue, J., Wahler, A., Fensel, D.: Semantic business process management: a vision towards using semantic Web services for business process management. In: *IEEE International Conference on e-Business Engineering (ICEBE'05)*, pp. 535–540 (2005). <https://doi.org/10.1109/ICEBE.2005.110>
12. OMG: Owl 2: Web ontology language. <https://www.w3.org/TR/owl2-overview/> (2004)
13. Polyvyanyy, A., Ouyang, C., Barros, A., van der Aalst, W.M.P.: Process querying: Enabling business intelligence through query-based process analytics. *Decis. Support Syst.* **100**, 41–56 (2017). <https://doi.org/10.1016/j.dss.2017.04.011>
14. Prud'hommeaux, E., Seaborne, A.: SPARQL Query Language for RDF. W3C Recommendation (2008). <https://www.w3.org/TR/rdf-sparql-query/>



15. Rospocher, M., Ghidini, C., Serafini, L.: An ontology for the business process modelling notation. In: *Formal Ontology in Information Systems - Proc. of the Eighth International Conference, FOIS 2014, September, 22–25, 2014, Rio de Janeiro, Brazil*, pp. 133–146 (2014)
16. White, S.A., Miers, D.: *BPMN Modeling and Reference Guide. Understanding and Using BPMN*. Future Strategies Inc., Lighthouse Pt, FL (2008)
17. Wohlin, C., Runeson, P., Höst, M., Ohlsson, M.C., Regnell, B., Wesslén, A.: *Experimentation in Software Engineering: An Introduction*. Kluwer Academic Publishers (2000)

# Retrieving, Abstracting, and Changing Business Process Models with PQL



Klaus Kammerer, Rüdiger Pryss, and Manfred Reichert

**Abstract** Due to the increasing adoption of process-aware information systems (PAISs) in enterprises, extensive process model repositories have emerged. In turn, this has raised the need for properly querying, viewing, and evolving process models. In order to enable context-specific views on the latter as well as on related process data, a PAIS should provide sophisticated techniques for abstracting large process models. Moreover, to cope with the complexity of process model changes, domain experts should be supported in evolving process models based on appropriate model abstractions. This chapter presents the PQL language for querying, abstracting, and changing process models. Due to the generic approach taken, the definition of process model abstractions and changes on any graph-based process model notation become possible. Overall, PQL provides a key contribution to take process model repositories to the next level.

## 1 Introduction

Process-aware information systems (PAISs) provide support for business processes at the operational level [28]. In particular, a PAIS separates process logic from application code relying on explicit *process models*. This enables a *separation of concerns*, which is a well-established principle in Computer Science to increase maintainability and to reduce costs of change [38]. During the recent years, the increasing adoption of PAISs has resulted in large *process model repositories* (i.e., process model collections) [36]. In many application environments, a process model may comprise dozens or hundreds of activities [3, 4, 36]. Furthermore, process

---

K. Kammerer (✉) · M. Reichert  
Institute of Databases and Information Systems, Ulm University, Ulm, Germany  
<http://www.uni-ulm.de/dbis>  
e-mail: [klaus.kammerer@uni-ulm.de](mailto:klaus.kammerer@uni-ulm.de); [manfred.reichert@uni-ulm.de](mailto:manfred.reichert@uni-ulm.de)

R. Pryss  
Institute of Clinical Epidemiology and Biometry, University of Würzburg, Würzburg, Germany  
e-mail: [ruediger.pryss@uni-wuerzburg.de](mailto:ruediger.pryss@uni-wuerzburg.de)

models may refer to business objects, organizational units, user roles, and other kinds of resources [10]. Due to this high complexity, any PAIS should support the various stakeholders with personalized *views* on the processes they are involved in [33]. For example, managers rather prefer an abstract overview on a business process, whereas process participants need a detailed view of the process parts they are involved in.

Several approaches for creating *process views* and *process model abstractions*, respectively, have been proposed in literature [5, 7, 16, 23, 24, 29, 32, 34]. Current proposals, however, focus on fundamental model abstraction techniques (e.g., to aggregate or reduce process model elements) in order to enable context-specific (e.g., personalized) process views. These approaches neither allow specifying abstractions independently from a particular process model (e.g., to create views on different processes a particular user is involved in) nor defining them in a more descriptive way. The latter means that it should be possible to express *what* information shall be covered by a particular process view rather than to specify *how* the view shall be created. With contemporary approaches, for each relevant process model the required abstractions need to be created manually, e.g., by specifying a sequence of aggregation / reduction operations. Consequently, the abstractions need to be specified separately for each process model, which causes high efforts in practice (cf. Fig. 1a). One approach to remedy this drawback is to decrease the number of operations required for abstracting process models by composing elementary operations to high-level ones [29]. Still, the application of respective operations is specific to a particular process model.

In existing approaches (see [28] for an overview), process changes refer to specific process model elements (e.g., nodes and edges) rather than to generic process properties (e.g., expressed in terms of process attributes). In consequence, it would be difficult to express, for example, that a specific user role shall be replaced by another one in all process models stored in the repository [36], i.e., multi-model changes would have to be separately applied by the user to each process model.

In many domains (e.g., database management), the use of descriptive languages is common when facing large datasets. For example, *SQL* has been used to create, query, and update data in relational databases [11]. With *PQL (Process Query Language)*, this chapter introduces a descriptive language for creating, querying and updating process models in process repositories.

PQL not only allows querying process models, but also defining process views (i.e., abstracting process models) in a declarative way. Moreover, updates on process views may be specified with PQL and then be propagated in a controlled way to the process model the view was derived from (cf. Fig. 1b). As an advantage of such a declarative approach, PQL expressions can be automatically applied to multiple process models (i.e., model sets) if required. For example, process participants may use PQL to define personalized *process views* on their process, e.g., by abstracting or hiding process information not relevant for them. Additionally, process model collections can be easily updated using declarative specifications. For example, a change of the same or similar process elements within multiple process models (and process views respectively) may be triggered by one and the same PQL change

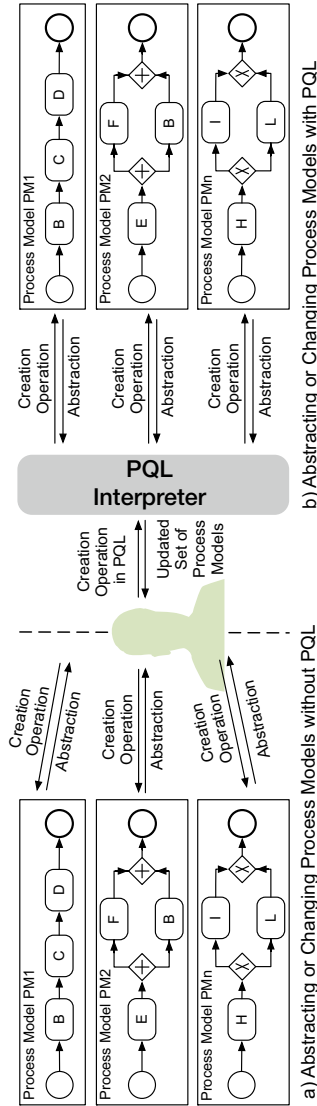


Fig. 1 Using PQL to abstract or change process models

description (e.g., changing all activities dealing with quality assurance in all variants of a particular business process [1]).

This chapter extends our previous work on PQL [14]. First, we introduce characteristic use cases for PQL in Sect. 2. In Sect. 3, we give detailed insights into process model abstractions and provide additional examples. Section 4 shows how to query, change, and abstract process model collections with PQL. Additionally, we discuss how PQL can be used to evolve process models with PQL updates of related process views. Section 5 gives insights into the architecture of a PAIS implementing PQL and Sect. 6 positions PQL in respect to the Process Querying Framework [25]. The chapter concludes with a summary in Sect. 7.

## 2 Use Cases

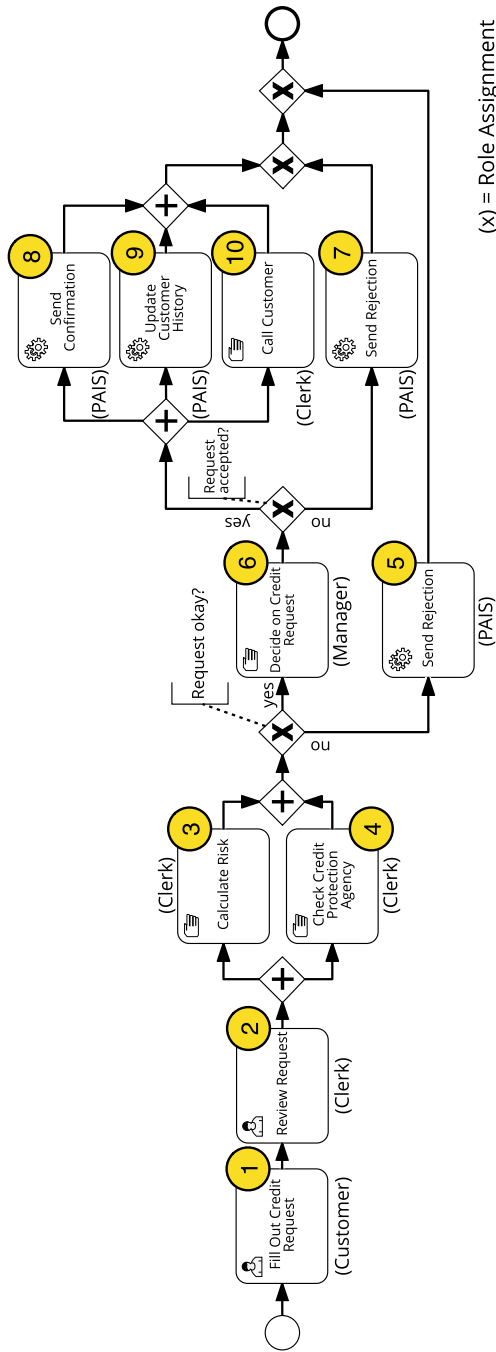
In the following, different use cases for managing process models are illustrated along a credit request process (cf. Fig. 2). The latter involves human tasks referring to three user roles (i.e., *customer*, *clerk*, and *manager*) as well as automatic tasks executed by the PAIS. For the sake of clarity and comprehensibility, the dataflow between the process activities is not considered in this example.

The process is started by the customer filling out a credit request form (Step ①). Then, the clerk reviews the credit request (Step ②), calculates the risk (Step ③), and checks the creditworthiness of the customer with the credit protection agency (Step ④). After completing these tasks, the clerk decides whether to reject the request (Step ⑤) or to forward it to his manager who finally decides on whether or not to grant the credit request (Step ⑥). If the manager rejects the request, a respective e-mail is sent to the customer (Step ⑦). Otherwise, a confirmation e-mail is sent (Step ⑧) and the customer relationship management (CRM) database is updated accordingly (Step ⑨). Finally, the clerk calls the customer in connection with after sales (Step ⑩), before completing the process.

Note that several variants of process models for different types of credit requests (i.e., *low-medium-high* volume and risk) may exist in a credit company.

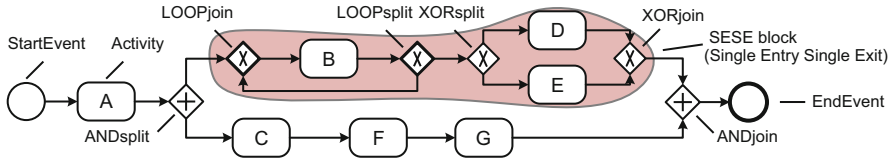
Regarding the management of process models, different use cases need to be supported. First, assume that the appropriate process model has to be selected out of a set of process models (Use Case *UC1*: Selection of Process Models). Furthermore, an evolution of the process model may become necessary, for example, when inserting an automatic task after task “Fill Out Credit Request” to retrieve customer data out of a database, if the customer has already been registered (Use Case *UC2*: Updates on Selected Process Models). In general, updates may be applied to a set of process models. Finally, the ability to delete or move tasks is crucial.

To increase the readability of process models [20] as well as to set a focus, *customized views* on process models are required. For example, a clerk does not want to see automated tasks, but only those he interacts with. Therefore, he should be able to apply abstractions on process models, e.g., by hiding technical tasks (Use



(X) = Role Assignment

Fig. 2 Credit request process (expressed in terms of BPMN 2.0)



**Fig. 3** Example of a process model captured in BPMN

Case *UC3: Reduction of Tasks*). As a complete reduction of all other tasks might affect model readability, it should be possible to further consolidate the tasks not executed by the clerk (Use Case *UC4: Aggregation of Tasks*).

Assume that the clerk wants to call a customer after sending out the confirmation. For this purpose, he changes the process model and moves task “Call Customer” after task “Send Confirmation”. In general, it is desirable to apply respective changes on an abstracted process view instead of a complex process model. When changing a process view, all process models associated with the view have to be updated accordingly, as the change becomes relevant for all participants (Use Case *UC5: Change Operations on Process Model Abstractions*).

PQL enables all these use cases based on concepts and technologies introduced in the following sections.

### 3 Fundamentals of Process Model Abstractions

This section provides basic terminology as well as fundamentals on process model abstractions needed for understanding the chapter. Section 3.1 defines basic notions, whereas Sect. 3.2 introduces various change operations for process models. In turn, Sect. 3.3 describes how to create and formally represent *process model abstractions*. In this context, we illustrate how elementary model abstraction operations may be composed to high-level (i.e., user-friendly) ones. Finally, Sect. 3.4 discusses how process model changes can be accomplished based on abstracted process models (i.e., process views).

#### 3.1 Process Model

A process model comprises *process elements*, i.e., *process nodes*, as well as the *control flow* between them (cf. Fig. 3). The latter is expressed in terms of *gateways* and *control flow edges* (cf. Definition 3.1). Note that the data perspective of business processes is excluded in this chapter to set a focus (see [17] for details on dataflow abstractions).

**Definition 3.1 (Process Model)** A *process model* is defined as a tuple  $P = (N, NT, CE, EC, ET, attr, val)$ , where:

- $N$  is a set of process nodes (i.e., activities, gateways, and start/end nodes).
- $NT : N \rightarrow NodeType$ , with  $NodeType = \{StartEvent, EndEvent, Activity, ANDsplit, ANDjoin, XORsplit, XORjoin, LOOPsplit, LOOPjoin\}$ , being a function with  $NT(n)$  returning the type of node  $n \in N$ . The nodes in  $N$  are divided into disjoint sets either comprising activities  $A$  ( $NT(n) = Activity, n \in A$ ) or structural nodes  $S$  ( $NT(n) \neq Activity, n \in S$ ), i.e.,  $N = A \cup S$ .
- $CE \subseteq N \times N$  is a set of precedence relations (i.e., *control edges*):  
 $e = (n_{src}, n_{dest}) \in CE$  with  $n_{src} \neq n_{dest}$ .
- $EC : CE \rightarrow Conds \cup \{TRUE\}$  assigns to each control edge either a branching condition or *TRUE*, with the latter meaning that the branching condition of the respective control edge always evaluates to true.
- $ET : CE \rightarrow EdgeType$ , with  $EdgeType = \{ET\_Control, ET\_Sync, ET\_Loop\}$ .  $ET(e)$  assigns a type to edge  $e \in CE$ .
- $attr : N \cup CE \rightarrow \mathfrak{P}^1(Attr)$  assigns to each process element  $pe \in N \cup CE$  its corresponding attribute set  $attr(pe) \in \mathfrak{P}(Attr)$  (with  $Attr$  denoting the set of all known attributes).
- $val : (N \cup CE) \times Attr \rightarrow valueDomain(Attr)$  assigns to any attribute  $x \in Attr$  of a process element  $pe \in N \cup CE$  its value:<sup>2</sup>.

$$val(pe, x) = \begin{cases} value(x), & x \in attr(pe) \\ null, & x \notin attr(pe) \end{cases}$$

□

Definition 3.1 can be used for representing process models and corresponding process model abstractions. In particular, Definition 3.1 is not restricted to a specific activity-oriented modeling language, but may be applied to any graph-based process modeling language. This chapter uses a subset of BPMN elements as modeling notation and further assumes that process models are *well-structured* according to the ADEPT meta model [27], i.e., sequences, parallel branchings, alternative branchings, and loops are specified as blocks with well-defined start and end nodes having the same gateway type. These blocks—also known as Single-Entry-Single-Exit (SESE) blocks (cf. Definition 3.2)—may be arbitrarily nested but must not overlap. Furthermore, ADEPT distinguishes between different edge types. *Control flow edges* define the temporal order of activity executions and *loop edges* describe re-entries in a process control flow. Accordingly, process models can be considered acyclic when excluding loop edges during analysis. To increase expressiveness, *synchronization edges* allow for a *cross-block* synchronization of parallel activities (similar to the links known from WS-BPEL). In Fig. 3, for example, activity  $E$  must not be enabled before  $G$  is completed, if a synchronization edge from activity  $G$

---

<sup>1</sup> Power set.

<sup>2</sup>  $value(x)$  denotes the value of process attribute  $x$ .



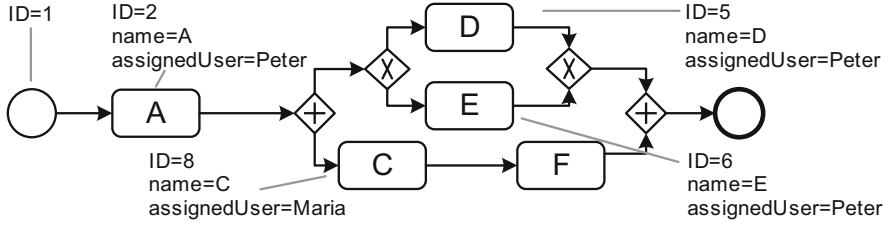


Fig. 4 Example of a process model with attributes captured in BPMN

Table 1 Examples of change operations

Operation	Description
$InsertNode(P, n_{pred}, n_{succ}, n)$	Node $n$ is inserted between nodes $n_{pred}$ and $n_{succ}$ in process model $P$ . Control edges between $n_{pred}$ and $n$ as well as between $n$ and $n_{succ}$ are inserted to ensure the existence of a control flow between these nodes.
$DeleteNodes(P, N^*)$	A set of nodes $N^*$ is removed from process model $P$ .
$MoveNode(P, n, n_{pred}, n_{succ})$	Node $n$ is moved from its current position to the one between $n_{pred}$ and $n_{succ}$ ; control edges are adjusted accordingly.

to  $E$  exists. Additionally, process elements may be associated with attributes, e.g., activities have attributes like  $ID$ ,  $name$ , and  $assignedRole$  (cf. Fig. 4).

**Definition 3.2 (SESE Block [12])** Let  $P = (N, NT, CE, EC, ET, attr, val)$  be a process model and  $X \subseteq N$  be a subset of activities (i.e.,  $NT(n) = Activity \forall n \in X$ ). Then, subgraph  $P' = (X, NT', CE', EC', ET', attr', val')$  is called a SESE (Single Entry Single Exit) block of  $P$  iff  $P'$  is connected and has exactly one incoming and one outgoing edge connecting it with  $P$ . Further,  $(n_s, n_e) \equiv MinimalSESE(P, X)$  denotes start and end nodes of the minimal SESE comprising all activities from  $X \subseteq N$ .  $\square$

How to determine SESE blocks is described in [12]. As we presume well-structured process models, a minimal SESE can be always determined [23].

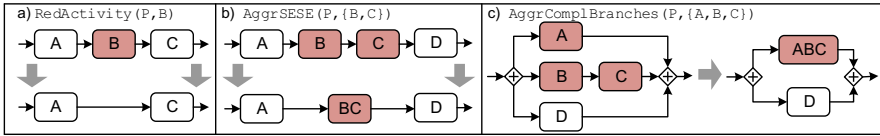
### 3.2 Changing Process Models

Table 1 shows elementary change operations on process models. These may be encapsulated as *high-level change operations*, e.g., to insert a complete process fragment through the application of a set of *InsertNode* operations [37].

Change operation  $InsertNode(P, n_{pred}, n_{succ}, n)$  inserts node  $n$  between nodes  $n_{pred}$  and  $n_{succ}$  in process model  $P$ . For this purpose, the control edge between  $n_{pred}$  and  $n_{succ}$  is removed and two control edges between  $n_{pred}$  and  $n$  as well as between  $n$  and  $n_{succ}$  are added to guarantee for connected nodes. Change

**Table 2** Examples of elementary abstraction operations

Operation	Description
$RedActivity(P, n)$	Activity $n$ and its incoming and outgoing edges are removed from $P$ , and a new edge linking the predecessor of $n$ with its successor is inserted (cf. Fig. 5a).
$AggrSESE(P, N^*)$	All nodes of the SESE block defined by $N^*$ are removed in $P$ and an abstract activity is re-inserted instead (cf. Fig. 5b).
$AggrComplBranches(P, N^*)$	Complete branches of an XOR/AND branching are aggregated to a branch with one abstracted node in $P$ . $N^*$ must contain the activities of all branches (i.e., activities between split and corresponding join gateway) that shall be replaced by a single branch consisting of one aggregated node (cf. Fig. 5c).

**Fig. 5** Examples of elementary abstraction operations

operation  $DeleteNodes(P, N')$ , in turn, removes all nodes from  $N'$  in process model  $P = (N, NT, CE, EC, ET, attr, val)$  and adjusts the control edges accordingly, which results in process model  $P' = (N', NT', CE, EC, ET, attr, val)$ , in particular,  $\forall n' \in N' \subseteq N : e_1 = (n_{pred}, n') \in CE \wedge e_2 = (n', n_{succ}) \in CE \Rightarrow e_1, e_2 \notin CE' \wedge (n_{pred}, n_{succ}) \in CE' \wedge n' \notin N'$ . Change operation  $MoveNode(P, n, n_{pred}, n_{succ})$  moves node  $n$  from its current position to the one between nodes  $n_{pred}$  and  $n_{succ}$  in process model  $P$  and can be composed of operation  $DeleteNodes(P, n)$  and  $InsertNode(P, n_{pred}, n_{succ}, n)$ .

Due to lack of space, we omit a discussion of other change operations and refer interested readers to [30] instead.

### 3.3 Process Model Abstractions

In order to abstract a given process model (i.e., to create a *process model abstraction* or *process view*), its schema needs to be transformed accordingly. For this purpose, a set of *elementary operations* with well-defined semantics are provided (cf. Table 2), which may be further combined to realize *high-level abstraction operations* (e.g., to display all activities a particular actor is involved in as well as their precedence relations) [23, 29].

At the elementary level, two categories of operations are provided: *reduction* and *aggregation*. An *elementary reduction operation* hides an activity of a process model, e.g.,  $RedActivity(P, n)$  removes activity  $n$  and its incoming/out-

going edges, and re-inserts a new edge linking the predecessor of  $n$  with its successor in process model  $P$  (cf. Fig. 5a). An *elementary aggregation operation*, in turn, takes a set of activities as input and combines them to an abstracted node. For example,  $AggrSESE(P, N')$  removes all nodes of the SESE block induced by node set  $N'$  and re-inserts an abstract activity instead (cf. Fig. 5b). In turn,  $AggrComplBranches(P, N')$  aggregates multiple branches of an XOR/AND branching to a single branch with one abstracted node (cf. Fig. 5c). In general, practically relevant abstractions of a process model can be created through the consecutive application of elementary operations on the respective process model (cf. Definition 3.3) [5, 29]. Note that there exist other elementary operations, which refer to process perspectives other than control flow (e.g., dataflow, see [15]).

**Definition 3.3 (Process Model Abstraction)** Let  $P = (N, NT, CE, EC, ET, attr, val)$  be a process model. A process model abstraction (also denoted as process view)  $V(P)$  is described through a creation pair  $CS_{V(P)} = (P, Op)$  with  $Op = \langle Op_1, \dots, Op_n \rangle$  being a sequence of elementary operations applied on  $P$ , where  $Op_i \in \mathcal{O} \equiv \{RedActivity(P, n), AggrSESE(P, N'), AggrComplBranches(P, N')\}$ ,  $i \in [1..n]$ .  $\square$

A node  $n$  of the abstracted process model  $V(P)$  either directly corresponds to a node  $n \in N$  of the original process model  $P$  or it abstracts a set of nodes from  $P$ .  $PMNode(V(P), n)$  reflects this correspondence by returning either  $\{n\}$  or node set  $N_n$  aggregated in  $V(P)$ , depending on creation pair  $CS_{V(P)}$ . For example, consider  $V(P)$  with  $CS_{V(P)} = (P, AggrComplBranches(P, (A, B, C)))$  (cf. Fig. 5c). Then,  $PMNode(V(P), D) = \{D\}$  and  $PMNode(V(P), ABC) = \{A, B, C\}$  hold.

After abstracting a process model, unnecessarily complex control flow structures might result due to the generic nature of the abstraction operations applied. For example, certain branches of a parallel branching might become “empty” (i.e., nullified) or a parallel branching might only have one branch left after applying reductions. In such cases, gateways no longer needed may be removed to obtain an easier to comprehend schema of the abstracted model. For this purpose, well-defined refactoring operations are provided. In particular, refactorings do not alter the control flow dependencies of the activities and, hence, the behavioral semantics of the refactored process model is preserved [36].

To abstract multiple perspectives of a process model (e.g., control and dataflow) several elementary operations need to be co-applied [29]. For example,  $AggrSESE$  and  $AggrComplBranch$  as well as  $AggrDataElements$  [17] may be combined to the high-level operation  $AggregateControlAndDataFlow$  (cf. Fig. 6).

In general, the nodes to be abstracted (e.g., all activities performed by a specific role) should be easy to select. Current abstraction approaches, however, require the explicit specification of the respective nodes in relation to a particular process model (e.g., activity  $A$  in process model  $PI$ ). Thus, a declarative specification of the nodes to be abstracted (e.g., to select all activities a particular user is involved in) would significantly increase the usability of any approach dealing with process model abstractions.

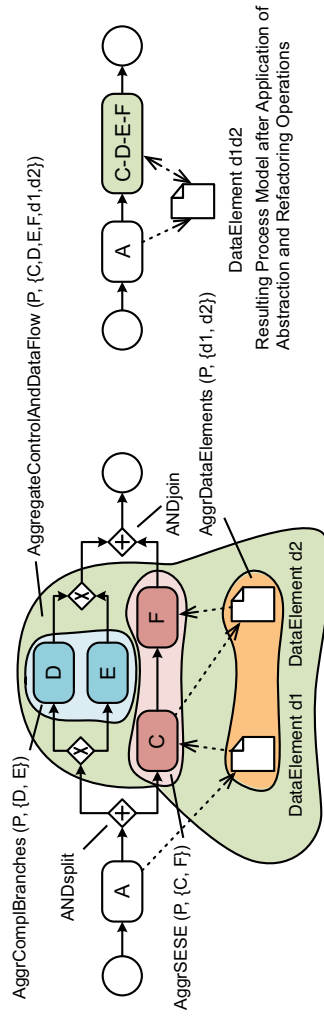


Fig. 6 Composition of elementary abstraction operations

### 3.4 Updating Process Models Based on Model Abstractions

Process model abstractions not only enable context-specific process views, but may be also used as basis for updating large process models. In the following, we illustrate how updates of an abstracted large process model can be specified and propagated to the original model. We restrict our considerations to selected operations changing the control flow, whereas update operations related to other process perspectives are not considered (see [17] for details).

When allowing users to change a process model by adapting a related process view (i.e., model abstraction), it needs to be ensured that the view update can be automatically propagated to the original process model. For this purpose, well-defined view update operations are provided, whose pre- / post-conditions ensure that only those changes may be propagated to the original process model that do not introduce any syntactical or semantical errors (see [18] for an overview of view update operations). Note that the propagation of view updates to an original process model is not straightforward. In certain cases, ambiguities occur when propagating view updates to the original process model, e.g., it might be impossible to determine a unique position for inserting a new activity in the original process model.

Consider the example from Fig. 7: when inserting activity X between activity A and aggregated activity CD in process view  $V(P)$ , several positions become possible for inserting X in the original process model, i.e., there exist ambiguities in how to transform the process view change into a corresponding process model change. For example, X may be inserted before or after B. Note that such ambiguities are due to the model abstractions applied (i.e., reduction of B in the example).

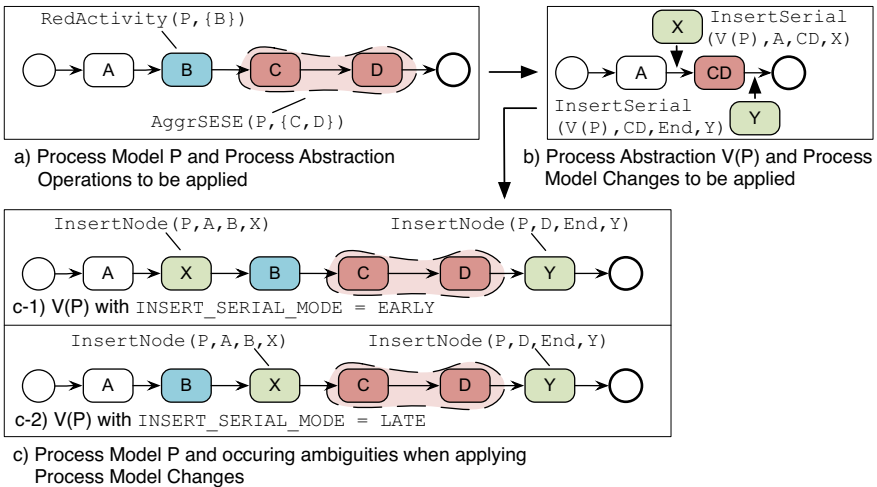


Fig. 7 Emerging ambiguities when propagating changes on process abstractions

When propagating view updates to an original process model, it should be not required from users to manually resolve ambiguities as this is a time-consuming and error-prone task. Note that the original process model might be unknown to a user. Therefore, we provide *parameterizable* view update operations [18], which may then be configured in a way allowing for the automated resolution of ambiguities according to a well-defined policy. For example, view update operation *InsertSerial* in Fig. 7 has parameter  $InsertSerialMode = \{EARLY, LATE, PARALLEL\}$ , which indicates the position at which an activity, which was added to a process view, shall be inserted into the original process model in case of ambiguities. In general, a process model abstraction may comprise a parameter set (cf. Definition 3.4) in addition to the creation pair introduced in Section 3.3. The parameter values are then used to define standard parameters for every parameterizable view update operation (see [18] for details).

**Definition 3.4 (Parameter Set)** Let  $P = (N, NT, CE, EC, ET, attr, val)$  be a process model. A process model abstraction (i.e., view)  $V(P)$  is described through a creation pair  $CS_{V(P)} = (P, Op)$  and parameter set  $PS_{V(P)} = (P, p)$  with  $p = \langle p_1, \dots, p_n \rangle$  being a set of parameters defined for  $P$ :  $p_i \subseteq \mathcal{P} \equiv \{InsertSerialMode, InsertBlockMode, InsertBranchMode, DeleteActivityMode, DeleteBlockMode\}$ ,  $i \in [1..n]$ .  $\square$

Definition 3.5 illustrates the parametrizable view update operation *InsertSerial* to indicate how a process view change can be transformed into a corresponding process model change, taking the chosen parameterizations into account.

We introduce the following auxiliary functions used in the following:

$last(P, \{x_0, \dots, x_n\}) = x_n$ , where  $x_0, \dots, x_n \in N$  and  $(x_0, x_1), \dots, (x_{n-1}, x_n) \in CE$ ,  
 $first(P, \{x_0, \dots, x_n\}) = x_0$ , where  $x_0, \dots, x_n \in N$  and  $(x_0, x_1), \dots, (x_{n-1}, x_n) \in CE$ ,  
 $succ(P, n) = n'$ , where  $n, n' \in N$  and  $(n, n') \in CE$ ,  
with  $P = (N, NT, CE, EC, ET, attr, val)$ .

**Definition 3.5 (InsertSerial)** Let  $V(P)$  be a process model abstraction of  $P$  (i.e., a view),  $n'_1 = last(PMNode(V(P), n_1))$ ,  $n'_2 = first(PMNode(V(P), n_2))$ , and  $f = (succ(P, n'_1) \equiv n'_2)$ . Let further  $p$  be an *InsertSerialMode* parameter, and  $n_{new}$  be the activity to be inserted. Then:

$$InsertSerial(V(P), n_1, n_2, n_{new}) = \begin{cases} InsertNode(P, n'_1, n'_2, n_{new}) & f \\ InsertNode(P, n_1, succ(P, n'_1), n_{new}) & \neg f \wedge (p = EARLY) \\ InsertNode(P, pred(P, n'_2), n'_2, n_{new}) & \neg f \wedge (p = LATE) \\ Ops_{parallel} & \neg f \wedge (p = PARALLEL) \end{cases}$$

with  $(n_{split}, n_{join}) = MinimalSESE(P, n'_1, n'_2)$ ,  $Ops_{parallel} = \{InsertNode(P, pred(P, n_{split}), n_{split}, ANDsplit), InsertNode(P, n_{join}, succ(P, n_{join}), ANDjoin), InsertEdge(P, ANDsplit, ANDjoin, ET_Control), InsertNode(P, ANDsplit, ANDjoin, n_{new}), InsertEdge(P, n_1, n_{new}, ET_SoftSync), InsertEdge(P, n_{new}, n_2, ET_SoftSync)\}$ .  $\square$

## 4 The PQL Language

Operations for creating process views refer to the process model they shall be applied to. Thus, their effects cannot be described independently from this model, which causes high efforts if a particular abstraction shall be reused, i.e., be applied to multiple process models. To remedy this drawback, we introduce the process query language PQL, which allows specifying abstractions independently from a specific process model. Furthermore, PQL allows expressing changes that shall be applied to a collection of process models.

### 4.1 Overview

PQL allows describing process model abstractions as well as process model changes in a declarative way. Such declarative descriptions, in turn, may be applied to a single process model or to a collection of process models showing some common properties. In the following, we denote declarative descriptions of any abstraction or change of a collection of process models as *PQL requests*. In general, a PQL request consists of two sections: the *selection section* specifies the criteria for selecting the process models concerned by the PQL request, whereas the *modification section* defines the abstractions and changes to be applied to the selected process models.

Figure 8 illustrates the processing of a PQL request. First of all, an authorized user sends a *PQL request* to the *PQL interpreter* (Step ①). Then, all process models that match the predicates specified in the selection section of the PQL request are selected from the process repository (Step ②). If applicable, the changes specified in the modification section of the PQL request are then applied to all selected process models (Step ③). Subsequently, the abstractions set out by the modification section are applied to the selected process models as well (Step ④). Finally, all selected, changed, and abstracted process models are presented to the user (Step ⑤). Note that Steps 3 and 4 are optional depending on the respective definition of the modification section.

The specification of PQL requests is based on the *Cypher Query Language* [21], which we *adjust* to the specific requirements of process model repositories. More precisely, Cypher is a declarative graph query language known from the Neo4J graph database. In particular, Cypher allows querying and changing the graphs from a graph database [31]. Furthermore, it has been designed with the goals of efficiency, expressiveness, and human-readability. Thus, Cypher is well suited as basis for PQL. A first example of a PQL request expressed with Cypher is depicted in Listing 1.

```
1 MATCH a1:ACTIVITY-[:ET_Control]->a2:ACTIVITY-[:ET_Control]->a3:ACTIVITY
2 WHERE not (a1-[:ET_Control]->a3)
3 RETURN a3
```

**Listing 1** Example PQL request

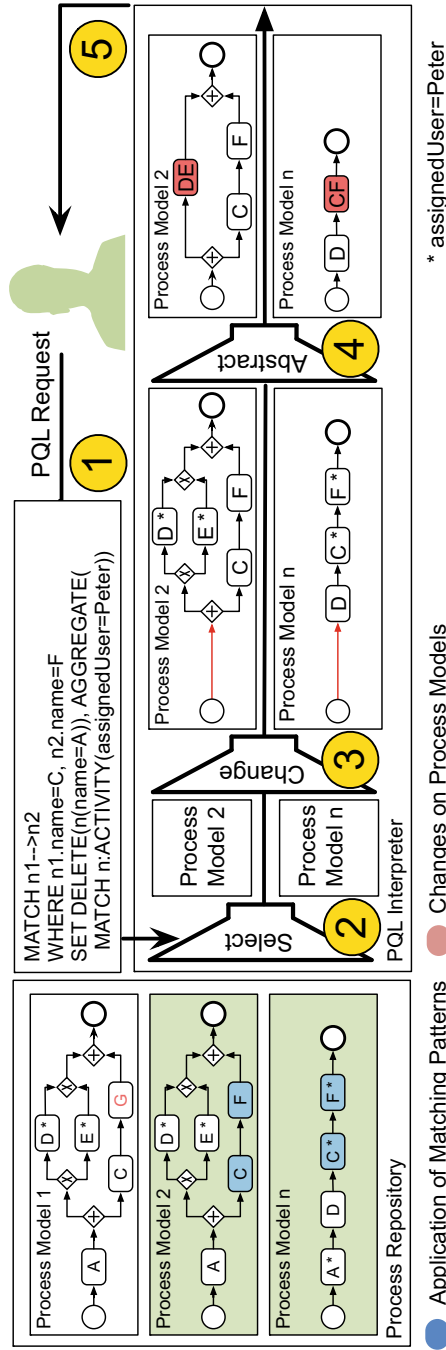


Fig. 8 Processing a PQL request



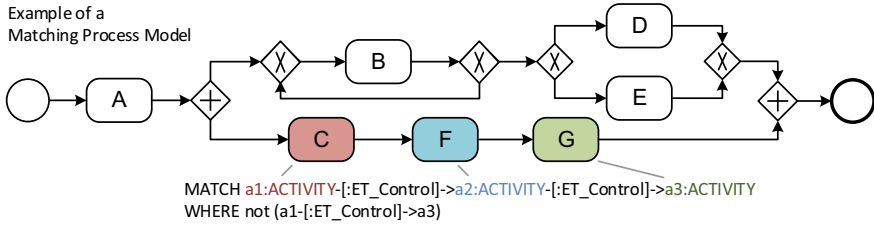


Fig. 9 PQL request determining a sequence of three activities

Line 1 refers to the selection of all process models that contain a path (i.e., a sequence of edges with type *ET\_Control*) linking activities *a1*, *a2*, and *a3*. Note that *a1*, *a2*, and *a3* constitute variables. To be more precise, the PQL request searches for process models comprising any sequence consisting of three activities (cf. Line 1), which must not be direct predecessors of *a1* (cf. Line 2), and returns only direct successors of *a2* (cf. Line 3). An application on the process model depicted in Fig. 3, for example, returns activity *G* as the only possible match (cf. Fig. 9).

Listing 2 presents the general syntax of a PQL request in BNF grammar notation [2]. Other relevant PQL syntax elements will be introduced step-by-step.

```
PQLrequest ::= match where? set?
```

Listing 2 BNF for a PQL request

## 4.2 Selecting Process Models and Process Elements

PQL enables a predicate-based selection of process models and process elements. First, a search predicate describes the structural properties of the process models to be queried, e.g., to select all process models comprising two sequential nodes *n1* and *n2* (cf. Step ② in Fig. 8). Second, process models and elements are selected through predicate-based rules on process element attributes. Usually, the latter are defined for each process model maintained in the repository, e.g., a user role designated to execute certain activities [29]. In general, a *predicate* allows assigning properties (i.e., attributes) to process elements and may be used to compare attributes of process models and elements. In this context, *comparison operators* for numerical values (i.e.,  $\neq$ ,  $<$ ,  $\leq$ ,  $=$ ,  $\geq$ ,  $>$ ) can be used. For example, for a string value we can check its equality with a fixed value or calculate its edit distance to another string value (to determine their similarity) [35]. Two or more predicates may be concatenated using Boolean operations (i.e., *AND*, *OR*, and *NOT*).

As aforementioned, PQL provides *structural* and *attributitional matching patterns* to determine whether a specific process fragment is present in a particular process model. To be more precise, *structural matching patterns* consider the control flow of a process model, i.e., they define the fragments that need to be present in selected

process models. In turn, *attributional matching patterns* allow selecting process models and elements based on process element attributes.

**Structural matching patterns** define constraints on the process fragments to be matched against the process models stored in the repository. In PQL, structural matching patterns are tagged with keyword `MATCH` (Line 1 in Listing 3), followed by a matching pattern characterizing the respective process fragment (Line 3).

```

1 match      ::= "MATCH" match_op ((" match_op)+)?
2 match_op   ::= match_pat | "VIEW" any_val
3 match_pat  ::= (PQL_PATHID "=")? (MATCH_FUNCTION (" path ") | path)
4
5 path       ::= node ((edge) node)+)?
6
7 node       ::= PQL_NODEID (":" NODETYPE)? ("(" NODEID ")")?
8
9 edge       ::= cond_edge | uncond_edge
10 uncond_edge ::= ("-" | "->")
11 cond_edge  ::= ((" edge_attrib "-" ) | (" edge_attrib "->"))
12 edge_attrib ::= "[" PQL_EDGEID? (":"
13              ((EDGETYPE ("|" EDGETYPE)* )? | edge_quant)?
14              "]"
15 edge_quant ::= "*" (EXACT_QUANTIFIER |
16                 (MIN_QUANTIFIER ".." MAX_QUANTIFIER)?)?

```

**Listing 3** BNF for structural matching in a PQL request

Structural matching patterns are further categorized into dedicated and abstract ones. While *dedicated patterns* (Lines 9–14 in Listing 3) allow describing SESE blocks of a process model, *abstract patterns* (Lines 15+16) offer an additional edge attribute defining control flow adjacencies between nodes, i.e., the proximity of a pair of nodes. For example, in order to select all succeeding nodes of activity *A* in Fig. 4 abstract structural patterns are required. Table 3 summarizes basic PQL structural matching patterns.

**Attributional matching patterns** allow filtering the process fragments selected through structural matching. For this purpose, predicates referring to process element attributes are defined (cf. Listing 4). Attributional matching is indicated with keyword `WHERE` that may follow a `MATCH` keyword (cf. Table 4). Note that attributional matching patterns refer to process elements pre-selected by a structural matching pattern. For example, node variable *a*, which is selected with pattern `MATCH a`, can be filtered with an attributional matching pattern to fit nodes with attribute *ID* = 5 as follows:

```
MATCH a : ACTIVITY(*) WHERE a.ID = 5
```

```

1 where      ::= "WHERE" predicate ((BOOL_OPERATOR predicate)+)?
2 predicate  ::= comparison_pred | regex_pred
3
4 comparison_pred ::= PROPERTY_ID COMPARISON_OPERATOR any_val
5 regex_pred  ::= PROPERTY_ID "-" REGEX_EXPRESSION

```

**Listing 4** BNF for attributional matching in a PQL request

**Table 3** Examples of structural matching patterns

Pattern	Description	Type
MATCH a->b	Existence of an edge of any type between nodes <i>a</i> and <i>b</i> .	dedicated
MATCH a(2) - [:E_TYPE] ->b	A process fragment whose nodes <i>a</i> and <i>b</i> are connected by an edge of type <i>E_TYPE</i> ; furthermore, <i>a</i> has attribute ID with value 2	dedicated
MATCH a(2) - [:] ->b	A process fragment whose nodes <i>a</i> and <i>b</i> are connected by a control flow edge; pattern “[:]” acts as shortcut.	dedicated
MATCH a- [*1..5] ->b	A process fragment with nodes <i>a</i> and <i>b</i> that do not necessarily succeed directly, but are separated by at least one and at most five nodes.	abstract
MATCH a- [*] ->b	An arbitrary number of nodes between nodes <i>a</i> and <i>b</i> .	abstract
MATCH p = minSESE(a-[:*3]->c)	A minimum SESE block with a maximum of three control edges between nodes <i>a</i> and <i>c</i> .	abstract

**Table 4** Examples of attributional matching patterns

Pattern	Description
MATCH (a) WHERE (a.NAME="Review Request")	Select all nodes with name “Review Request”.
MATCH (a) WHERE HAS (a.attrib)	Select all nodes for which an attribute with name <i>attrib</i> is assigned.
MATCH (a) WHERE a:ACTIVITY	Select all nodes with node type <i>ACTIVITY</i> .
MATCH (a-[*]->b) WHERE a:ACTIVITY(1), b(2)	Select (1) activity <i>a</i> with ID=1 and node type <i>ACTIVITY</i> and (2) node <i>b</i> with ID=2.

Attributional matching patterns may be combined with structural ones. For this purpose, the abbreviated form shown in Listing 5 can be used, i.e., PQL request `MATCH (a) WHERE (a.NAME="Review Request")` may be abbreviated as `MATCH a(NAME="Review Request")`.

```

1 node ::= PQL_NODEID (":" NODETYPE)?
2       ("(" (NODEID | predicate ((BOOL_OPERATOR predicate)+)?) ")")?

```

**Listing 5** BNF for combined matching in a PQL request

Figure 10 illustrates the application of a matching pattern to the process model from Fig. 4. Figure 10a matches a sequence of nodes  $n_1$ ,  $n_2$ , and  $n_3$ , with node  $n_1$  having attribute  $ID = 2$ , node  $n_2$  being an arbitrary node, and node  $n_3$  having type *ACTIVITY*. Figure 10b matches for a node  $n_1$  with  $n_1.ID = 8$  and arbitrary nodes

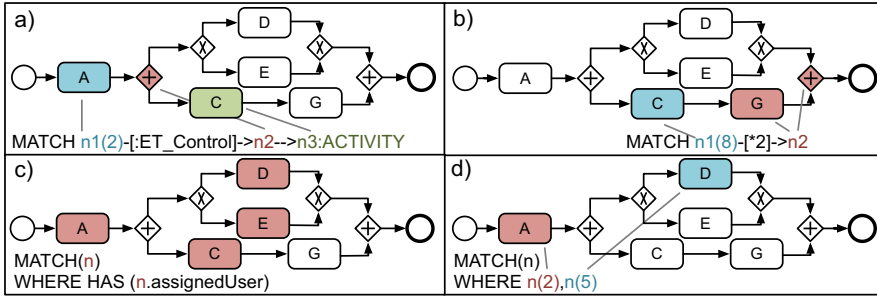


Fig. 10 Overview of PQL matching patterns

succeeding  $n_1$  with a maximum distance of 2 to  $n_1$  (i.e., nodes  $G$  and  $ANDjoin$  in the example). In turn, Fig. 10c matches all nodes with assigned attribute *assignedUser*. Finally, Fig. 10d matches nodes whose *ID* is either 2 or 5 (i.e., nodes  $A$  and  $D$ ).

### 4.3 Changing Process Models

In contemporary process model repositories, usually, common changes to multiple process models need to be applied separately to each model. This not only causes high efforts for process engineers, but constitutes an error-prone task as well. To remedy this drawback, PQL allows changing all process models defined in the *selection section* of a PQL request in one go, i.e., by one and the same *change transaction*. Structural matching patterns may be used to select the process models to be changed. PQL change operations are listed in Sect. 3.2.

Consider the credit request process from Fig. 2. Assume that after filling out the credit request, another activity, loading customer data from a database, shall be inserted before reviewing the request. Listing 6 shows how to insert a node with type *ACTIVITY* and name ‘Load Customer Data’ between nodes *FillOutCreditRequest* and *ReviewRequest*. Note that the insertion will be applied to all process models containing these nodes (Line 2). Model changes and abstractions can be defined in one PQL request, with the changes being applied first.

```

1 MATCH a-[:]->b
2 WHERE a.NAME="Fill Out Credit Request", b.NAME="Review Request"
3 SET INSERTNODE(a, b, ACTIVITY, "Load Customer Data")

```

Listing 6 PQL request to insert a node

## 4.4 Abstracting Process Models

This section illustrates how to define the abstractions that shall be applied to the process models referred by the selection section of a PQL request. Based on the matching patterns, PQL allows defining abstractions independently of a specific process model. As opposed to elementary abstraction operations (cf. Sect. 3.3), two high-level abstraction operations are introduced: `AGGREGATE` and `REDUCE`. Both allow abstracting a set of arbitrary process elements (including data elements [17]). Thereby, the process elements to be abstracted are categorized into process element sets according to their type (e.g., node type). If an aggregation shall be applied to a collection of process nodes, a minimum SESE block comprising all nodes is determined and replaced by an abstracted node. In this way, the well-structuredness and soundness of the resulting process model can be ensured.

PQL supports two ways of managing process abstractions. First, *persistent process abstractions* are stored persistently. This materialization is similar to materialized views in relational databases. Second, *inline process abstractions* are generated on-the-fly when processing the PQL request, without persisting query results.

We first introduce inline process abstractions. In PQL, abstractions of process models are tagged with the keyword `SET`. In turn, keywords `AGGREGATE` and `REDUCE` indicate the elements to be aggregated or reduced (cf. Listing 7, Fig. 10).

```

1 set          ::= "SET" operation ((" operation")+)?
2 operation    ::= abstraction | change_operation | view_operation
3 abstraction  ::= ("AGGREGATE" | "REDUCE")+ "(" PQLrequest ")"
4 view_operation ::= "CREATE VIEW" any_val (abstraction)+
5              | "DELETE VIEW" any_val

```

**Listing 7** BNF for structural matching in a PQL request

The PQL request depicted in Listing 8 selects all process models that contain two nodes *a* and *b* with names *Review Request* and *Call Customer*. Process elements selected by the first `MATCH` are aggregated if their type corresponds to *ACTIVITY* and  $val(pe, assignedUser) = Clerk$  holds. When applying the PQL request to the credit request process from Fig. 2, all nodes with assigned user *Clerk* (cf. Fig. 11a) as well as automated activities (cf. Fig. 11b) are aggregated.

```

1 MATCH a:ACTIVITY(NAME="Review Request"),b:ACTIVITY(NAME="Call Customer")
2 SET   AGGREGATE(
3     MATCH n:ACTIVITY(*)
4     WHERE n.assignedUser="Clerk"
5     OR n.assignedUser="Automatic*");

```

**Listing 8** PQL request to aggregate nodes

Listing 9 depicts a PQL request removing parallel nodes *Calculate Risk* and *Check Credit Protection Agency* (cf. Fig. 2). First, all process models are selected, for which an `ANDsplit` node is succeeded by node *a* with name *Calculate Risk* and node *b* with name *Check Credit Request Agency* (Lines 1+2). Then, nodes *a* and *b* are reduced (cf. Fig. 12a), i.e., the nested PQL request (Line 4) uses the same variables *a* and *b* as the parent PQL request, e.g., nested variable *b* refers to node

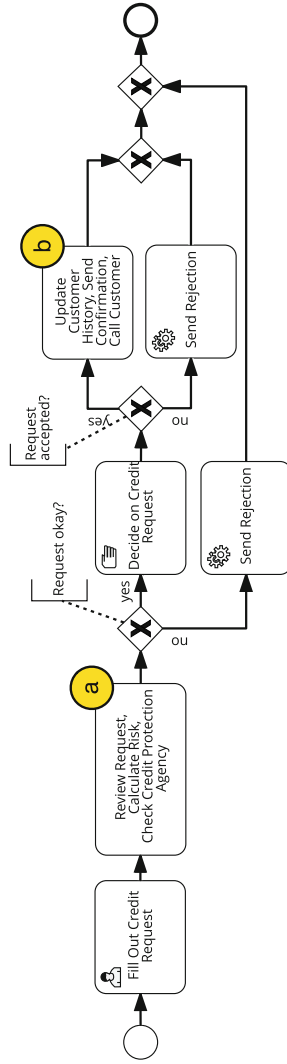


Fig. 11 Credit request process with aggregated nodes

*Check Credit Request Agency* defined in Lines 1+2. Note that AND gateways are reduced due to the application of refactoring operations (cf. Sect. 3.3).

```

1 MATCH n:ANDsplit-[:] ->a:ACTIVITY(NAME="Calculate Risk"),
2 n:ANDsplit-[:] ->b:ACTIVITY(NAME="Check Credit Request Agency")
3 SET REDUCE(
4     MATCH a,b);

```

**Listing 9** PQL request to reduce nodes

## 4.5 Handling Process Views with PQL

The specifications based on matching patterns and process abstractions are managed either by the requesting entity or server side. In PQL, a *process view* references related process models from a repository. This view, in turn, may define abstractions to the selected process models. Accordingly, a process view comprises one or multiple matching patterns (cf. Sect. 4.2), a parameter set (cf. Definition 3.4) and, optionally, a set of process abstraction operations to be applied to the selected process models (cf. Sect. 3.4).

As opposed to inline matching patterns and process abstractions, view creation operations should not be transmitted for every PQL request, but be managed server side. This centric approach allows for the reuse of the view by multiple entities, similar to SQL views. Moreover, process views may either be materialized (i.e., explicitly stored on the server side) or be virtually handled. In the latter case, the process view is generated upon a PQL request by first retrieving the process models according to the matching patterns and then applying the stored abstraction operations (i.e., aggregations and reductions) to them.

### 4.5.1 Creating, Updating, and Deleting Process Views

Listing 10 shows a PQL request for **creating a process view**. According to the structural matching pattern (Line 1) the view is created on every process model containing an activity with name *FillOutCreditRequest*. Moreover, the name of the view (i.e., “ManagerView”) is specified (Line 2), together with the parameter set applied for the view, i.e., parameter *InsertSerialMode* (Line 3). Default values are defined for every parameter, if the latter is not set upon view creation (cf. Sect. 3.4). Finally, abstraction operations *aggregate* and *reduce* are applied on all matching (i.e., selected) process models. For example, if the PQL request is applied on the credit request process from Fig. 2, the resulting process view solely comprises aggregated tasks related to the clerk, whereas all service tasks (e.g., task “Send Rejection”) are removed.

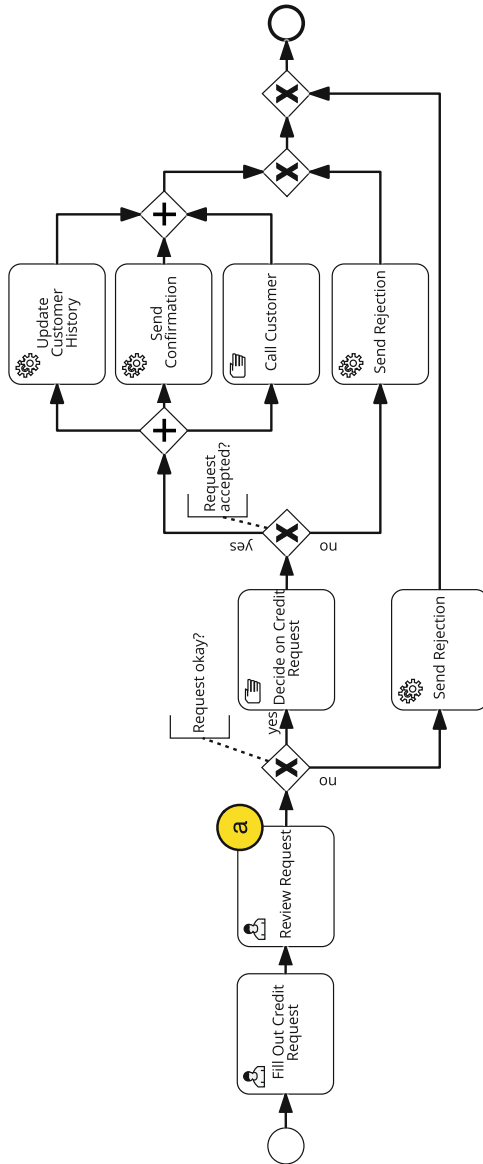


Fig. 12 Credit request process with reduced nodes



```

1 MATCH a:ACTIVITY(NAME="Fill Out Credit Request")
2 SET CREATE VIEW ManagerView (
3     INSERT_SERIAL_MODE=EARLY,
4     AGGREGATE (
5         MATCH n:ACTIVITY(*)
6         WHERE n.assignedUser="Clerk"),
7     REDUCE (
8         MATCH n:ACTIVITY(*)
9         WHERE n.NAME="Automatic*"
10    )
11 )

```

**Listing 10** PQL request to create a process view

Note that a process view can realize process abstractions as illustrated in Listing 10. However, it may be also solely built based on matching patterns (cf. Listing 11) as well, e.g., when selecting a collection of process models without changing or abstracting them.

```

1 MATCH a:ACTIVITY(role="Clerk")
2 SET CREATE VIEW ClerkView

```

**Listing 11** PQL request to create a process view with one matching pattern

In order to **retrieve a process view**, its name can be set as matching value, e.g., “ClerkView” in Listing 12.

```

1 MATCH VIEW ClerkView

```

**Listing 12** PQL request to retrieve a process view

If the definition of a **process view shall be updated**, e.g., to change parameters or abstraction operations, first of all, the view needs to be matched (Line 1 in Listing 13), and then be overwritten by all parameters and operations specified by the PQL request. In the below example, view “ClerkView” is overwritten with parameter `INSERT_BLOCK_MODE` as well as an abstraction operation aggregating all activities assigned to users with role “Manager”.

```

1 MATCH VIEW ClerkView
2 SET INSERT_BLOCK_MODE=EARLY_EARLY,
3     AGGREGATE (
4         MATCH n:ACTIVITY(*)
5         WHERE n.assignedUser="Manager")

```

**Listing 13** PQL request to update a process view

Listing 14 shows a PQL request to **delete a process view** with name “ClerkView”.

```

1 MATCH VIEW ClerkView
2 SET DELETE VIEW

```

**Listing 14** PQL request to delete a process view

## 4.5.2 Changing Abstracted Process Models

Process views not only enable context-specific process visualizations through model abstractions, but also provide the basis (i.e., interface) for changing large process

models based on simpler model abstractions. When changing a process view and, subsequently, the original process model, the correctness of the resulting model needs to be ensured and potential ambiguities be properly handled. More precisely, the view changes need to be transformed into process model changes, i.e., a sequence of change operations (cf. Sect. 3.4). In this context, potential ambiguities are resolved through the use of the pre-specified configuration parameters.

If a process abstraction, created with an *inline* PQL request, shall be changed, the PQL request must comprise the abstraction operations, configuration parameters, and change operations. As example consider Fig. 7. The insertion of X between A and aggregated activity CD in terms of an inline PQL request is expressed by Listing 15. Note that variables ‘a’ and ‘cd’ of the matching pattern, which refer to A and CD respectively, may be used by the insert operation.

```

1 MATCH a:ACTIVITY(NAME="A") - [:] ->cd:ACTIVITY(NAME="CD")
2 SET AGGREGATE (
3     MATCH n:ACTIVITY(*)
4     WHERE n.NAME="C" OR n.NAME="D")
5     ,REDUCE (
6     MATCH n:ACTIVITY(*)
7     WHERE n.NAME="B")
8     ,INSERTNODE(a,cd, ACTIVITY, "X")

```

**Listing 15** PQL request to change an inline process abstraction

Changes on a persistent process view look similar. Listing 16 shows two PQL requests to (a) create a process view aggregating activities C and D and (b) to reduce (i.e., hide) activity B (cf. Fig. 7). As opposed to inline PQL abstractions, the abstraction operations are defined once in the respective process view (Lines 1–8), i.e., there is no need to specify them for every PQL request (Lines 10–11).

```

1 MATCH n
2 SET CREATE VIEW newView (
3     AGGREGATE (
4     MATCH n:ACTIVITY(*)
5     WHERE n.NAME="C" OR n.NAME="D")
6     ,REDUCE (
7     MATCH n:ACTIVITY(*)
8     WHERE n.NAME='B') )
9
10 MATCH newView, a:ACTIVITY(NAME="A") - [:] ->cd:ACTIVITY(NAME="CD")
11 SET INSERTNODE(a,cd, ACTIVITY, "X")

```

**Listing 16** PQL request to a) create a process view b) change this process view

## 5 Implementation

To demonstrate the applicability of PQL we developed *Clavii BPM Platform*<sup>3</sup> [13], whose software architecture enables predicate-based specifications of process

<sup>3</sup> <http://www.clavii.com/>.

abstractions as well as process changes with PQL [14].<sup>4</sup> In this section, the PQL processing pipeline and its components are described.

## 5.1 Software Architecture

Figure 13 gives an overview of Clavii BPM platform, which has been implemented as Java EE application using the Activiti process engine [26]. Clavii comprises two core components: *Clavii Controller* and *Clavii Web Interface*. The latter is based on the Google GWT Web framework and interacts with the former through remote procedure calls [19]. *Clavii Controller* implements the logic of Clavii BPM platform, providing various engines for visualizing, changing, executing, and monitoring processes. Additional components we implemented include *TaskManager*, which executes service tasks as well as script tasks, and *ProcessFilterManager* that allows filtering process models.

Using Clavii Web Interface, all Clavii functions can be accessed through a Web-based application. Clavii fosters rapid prototyping as the creation of process models follows the correctness-by-construction principle [8]. Moreover, (partial) process models may be already executed during creation time, i.e., process elements can only be modeled syntactically correct and, for example, missing process data elements or decisions for branches necessary for correct execution are requested upon execution.

PQL is based on several components including *ProcessModelManager* (i.e., management of process models) and *ProcessFilterManager* (i.e., management of process views and process abstractions) (cf. Fig. 13).

Figure 14a illustrates the creation of a process model abstraction in Clavii. Drop-down menu ① shows a selection of pre-specified PQL requests that are directly applicable to a process model. In turn, Fig. 14b depicts a screenshot of Clavii's configuration window, where the stored PQL request "Technical Tasks" ② can be altered. The displayed PQL request aggregates all nodes that neither correspond to a service nor a script task ③.

## 5.2 Processing Pipeline

Figure 15 shows the PQL processing pipeline that is based on the data state model [6]. The latter describes *stages* and *data transformation steps* required to visualize data. Stages show a status of processed data and are illustrated as *diamonds*, whereas

---

<sup>4</sup> A screencast demonstrating the modeling and execution capabilities of Clavii may be retrieved via the following link: <http://screencast.clavii.com>.

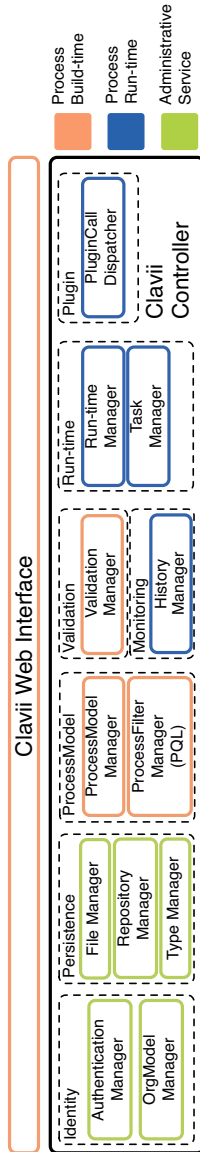
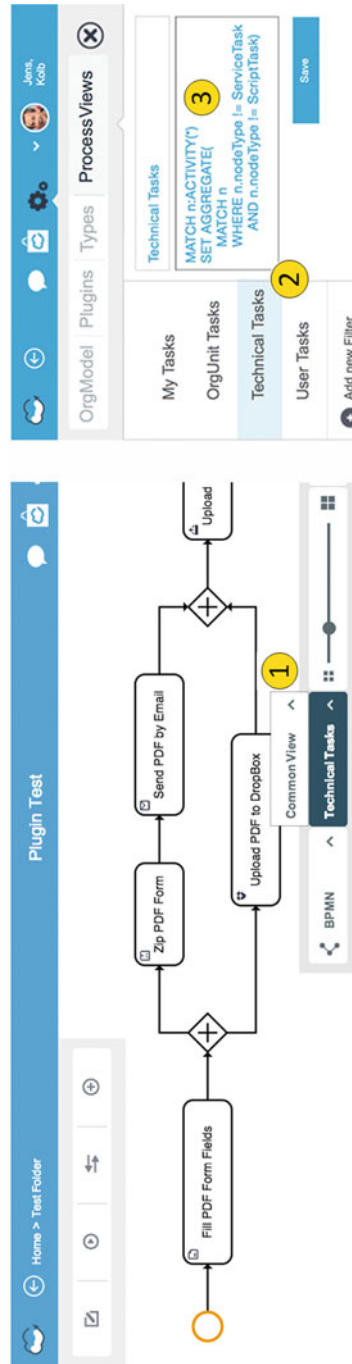


Fig. 13 Clavii architecture

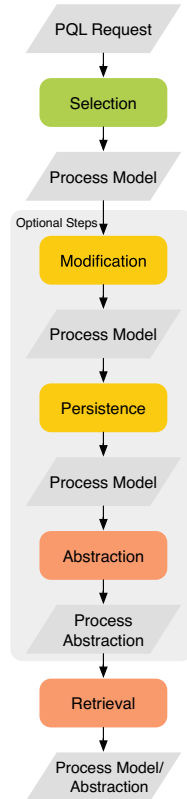


a) Process Model Abstraction

Fig. 14 Process model abstractions in the Clavii BPM platform

b) Stored PQL Request Preserving Technical Tasks

**Fig. 15** PQL processing pipeline



transformation steps describe transformations of processed data and are illustrated as *rectangles*. Steps act as transitions between stages.

Every PQL request passes this pipeline, which then processes the PQL actions (e.g., model change operations) orchestrating the involved software components. The processing pipeline comprises five steps, i.e., *selection*, *modification*, *persistence*, *abstraction*, and *retrieval*, whereby the behavior of each step can be configured (e.g., to indicate that certain steps are optional).

The *selection* step transforms a PQL request into an intermediate representation, which is then interpreted by the PQL software architecture. For this purpose, the *interpreter* checks the syntax of the PQL request against the PQL grammar, and then creates a parse tree as intermediate representation comprising parsed process fragments, dependency descriptions, and references to process models stored in the repository. A *dependency component* manages dependencies between process fragments and models, and evaluates the PQL change operations to be executed. Dependencies and change operations are defined in a PQL request and may contain matching patterns to select process models, change operations, and abstraction operations. In order to select process models, the dependency component accesses the process repository. In general, selecting process models based on graph matching requires

significant CPU and memory resources. Clavii, therefore, stores copies of the process models in a Neo4j graph database to benefit from its specific optimizations.

If a PQL request specifies process model changes, the *modification* step will execute the change operations on the selected process models. In this context, a dependency component is required to determine correct control flow positions for the changes specified by the PQL request. Subsequently, the *modification component* applies the changes to the selected process models. It manages the structure, considers the expressiveness level and constraints of process models, and ensures that all change operations are applied correctly.

Step *persistence* stores process model updates in the repository. The *persistence connector component* is used for this task. After selecting a process model and, optionally, performing changes, additional abstractions may be applied in the *abstraction* step. Finally, step *retrieval* puts process models, and process views, into an exchange container format.

### 5.3 PQL Lexer and Parser

To convert a PQL request into a set of methods to be invoked by the various components, e.g., ProcessModelManager performing change operations, *lexer* and *parser* are required. A *lexer* (i.e., a lexical analyzer) is a software that creates *tokens* out of a sequence of characters (i.e., a *string*) according to specified rules. The text must obey a specific *syntax* to correctly create the tokens, which are grouped strings with special meaning. A parser, in turn, converts the tokens into a semantic model (i.e., *intermediate representation*).

As *parser generator*, Clavii uses ANTLR (ANother Tool for Language Recognition) [22], which enables the dynamic generation of lexers and parsers based on a *grammar*, instead of developing them from scratch. ANTLR is written in Java and generates recursive descent parsers, i.e., a parsable string is decomposed into a parse tree and parsing is executed from the root element to the leaves (tokens) of the parse tree. Generally, an ANTLR grammar consists of 4 abstract language patterns: *sequence*, *choice*, *token dependence*, and *nested phrase* [22]. A sequence (of characters) is a token, e.g., “GET” or “POST” in the HTTP-Protocol [9]. Sequences are grouped by *rules*. In turn, `method: 'POST'` describes a *phrase* consisting of a rule *method* and an assigned token *POST*. The rule has to be executed when the token occurs in the parsed string. Furthermore, a rule may include *choices* between alternative phrases. Using phrase `method: 'POST' | 'PUT'`, for example, rule *method* has to be executed when one of the two tokens is present. Tokens may have dependent tokens. This occurs, for example, if a grammar requires that both the opening and the closing bracket must be present in a sequence. This dependency can be expressed by phrase `methodList: '(' (method) + ')'`, where *method* constitutes another rule, which must occur between two tokens '(' and ')'. Phrase `'(method)+'` expresses that rule *method* occurs at least once. Finally, rules may refer to themselves, which we denote as *nested phrases*, e.g., phrase

`expr: 'a' ('expr+') | INT` defines a nested phrase allowing for recursive definitions. Finally, sequence `a(5)` or sequence `a(a(5))` are valid expressions matching rule `expr`.

Using an ANTLR parser, every parsable string is converted into a parse tree. The latter constitutes a tree for which every node represents a previously parsed rule defined by a grammar. Figure 16 shows an example of the parse tree for the PQL request `MATCH a:ACTIVITY(id=2133) SET REDUCE(node.type != userTask)`—tokens are highlighted in bold face. All non-capitalized nodes of the parse tree (except leaves) represent *rules*. By contrast, capitalized nodes correspond to *tokens*.

## 6 PQL and the Process Querying Framework

This section aligns PQL with different parts provided by the Process Querying Framework [25].

### 6.1 Part 1: Model, Simulate, Record, and Correlate

Part 1 of the Process Querying Framework is responsible for acquiring and constructing behavior models as well as for formalizing and designing process queries.

PQL is based on methods to describe patterns of a process graph. Consequently, PQL queries refer to process graphs, i.e., process models represented as graph structure. On the one hand, PQL is not limited to a specific graph-based process modeling language. On the other, it cannot be applied to other behavioral models like event logs, simulation models, and correlation models. PQL queries are based on the Cypher graph query language.

Using PQL, we are able to support the selection, change, and deletion of process models (i.e., CRUD operations) from a repository. Correlation properties for selecting process models are based on both structural and attributional matching patterns. Structural matching patterns refer to the control flow of a process model, whereas attributional matching patterns define the properties of a node or edge within a process graph. Nodes may represent elements (e.g., activities) of a process model. Attributional matching patterns, in turn, refer to process element attributes that describe the various perspectives of a process model. Concerning the organizational perspective, for example, PQL enables queries for searching all activities a particular user is assigned to.

Although, PQL is described informally, its fundamental functions (e.g., creation of process views) have been formalized.



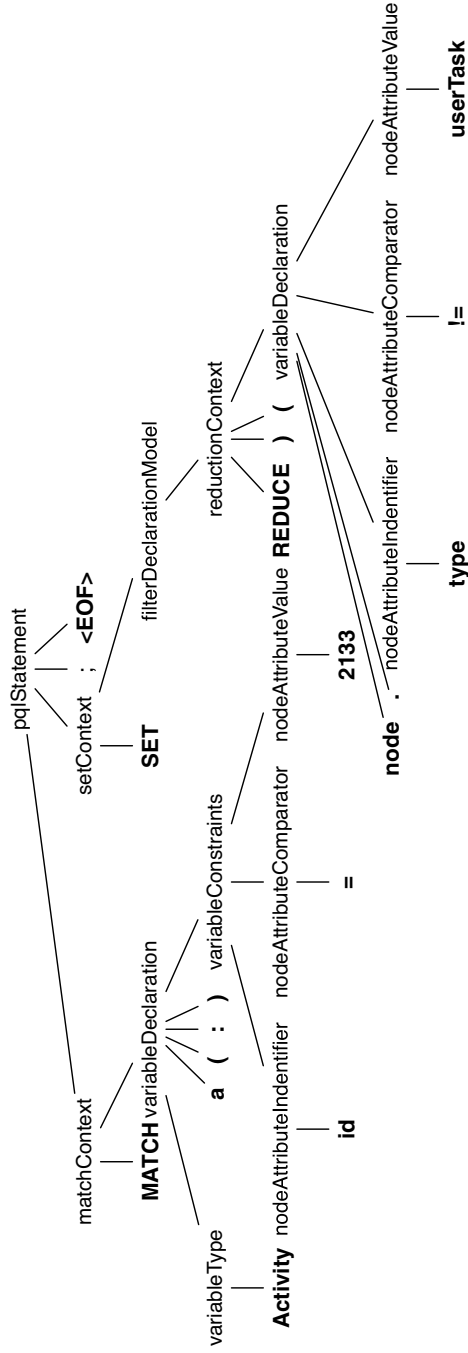


Fig. 16 Example of a PQL parse tree

## **6.2 Part 2: Prepare**

Part 2 of the framework shall ensure that process repositories are ready for being efficiently queried. This includes components for constructing dedicated data structures that can speed up the processing of the queries. The Process Querying Framework suggests two methods for preparing queries: indexing and caching. Indexing uses dedicated data structures to efficiently retrieve data records based on defined attributes, whereas caching stores data in a cache to speed up future repetitions of the requests. Preparation is currently not in the focus of the PQL framework. However, as we store process models as graphs in a highly efficient graph database (cf. Sect. 5), we can utilize the optimizations and preparations provided by the graph database (e.g., indices on attribute names, efficient operations on stored process graphs).

## **6.3 Part 3: Execute**

Part 3 of the framework deals with the actual execution of process queries. In this context, a filtering component takes a process repository and process query as input, and produces a filtered process repository as output. Behavioral models, i.e., process models, irrelevant for the process query, are filtered out. Moreover, the optimization component of the Process Querying Framework is responsible for query optimization, taking the same input as the filtering component and producing an efficient execution plan. We did not develop explicit techniques enabling such logical optimizations, e.g., to convert a PQL query into an equivalent, but more efficiently processible PQL query. However, as shown along the presented processing pipeline, PQL requests are processed according to a well-defined procedure (cf. Sect. 5.2).

The Process Querying component takes as inputs a filtered process repository, execution plan, index, process statistics and cache, and then applies a process querying method to produce a process repository implementing the query. We showed how to convert PQL requests to an intermediate representation enabling us to feed Clavii process querying components, i.e., *ProcessModelManager* and *ProcessFilterManager* (cf. Sect. 5.3). PQL is not only able to read and filter (i.e., retrieve) process models, but also to abstract and/or update them. Note that PQL is the only process querying language supporting updates on process models, either directly or indirectly by changing related process views.

## **6.4 Part 4: Interpret**

According to Part 4 of the framework, users shall be supported while designing process queries and during the interpretation of query results. For example, if

errors occur during the execution of a query, e.g., due to insufficient resources or structural/semantical errors of the query, appropriate feedback to the user or the requesting component shall be provided.

In Clavii, syntax errors of a PQL request are represented as Java Exceptions (PQLSyntaxError), which are handled by the Clavii system. An error message will be presented in the user interface, if a PQL syntax contains errors or querying a PQL request takes longer than a pre-specified time threshold.

However, PQL does not provide concepts for the intent of explaining queries to a user.

## 7 Conclusion

We introduced the *Process Query Language (PQL)*, which enables users to automatically select, abstract, and change process models in large process model collections. Due to its generic approach, the definition of process model abstractions and changes on any graph-based process notation becomes possible. For this purpose, *structural* and *attributional matching patterns* are used, which declaratively select process elements either based on the control flow structure of a process model or element attributes. *PQL* has been implemented in a proof-of-concept prototype demonstrating its applicability. Altogether, process querying languages like PQL will become a key part of process repositories to enable a professional management of abstractions and changes on large process model sets.

**Reprint** Figures 1, 3, 4, 5, 8, 9, 10, and 14 are reprinted with permission from K. Kammerer, J. Kolb, and M. Reichert. *PQL—A descriptive language for querying, abstracting and changing process models*. Enterprise, Business-Process and Information Systems Modeling. Springer, 2015. pp. 135–150 (“© Springer International Publishing Switzerland 2015”).

## References

1. Ayora, C., Torres, V., Weber, B., Reichert, M., Pelechano, V.: VIVACE: a framework for the systematic evaluation of variability support in process-aware information systems. *Inf. Software Tech.* **57**, 248–276 (2015)
2. Backus, J.: Can programming Be liberated from the Von Neumann style? A functional style and its algebra of programs. *Commun. ACM* **21**(8), 613–641 (1978)
3. Bobrik, R., Bauer, T., Reichert, M.: Proviado - personalized and configurable visualizations of business processes. In: *Proceedings of the 7th International Conference on E-Commerce and Web Technologies*, pp. 61–71 (2006)
4. Bobrik, R., Reichert, M., Bauer, T.: Requirements for the visualization of system-spanning business processes. In: *Proceedings of the 16th International Workshop on Database and Expert Systems Applications*, pp. 948–954 (2005)

5. Bobrik, R., Reichert, M., Bauer, T.: View-based process visualization. In: Proceedings of the 5th International Conference on Business Process Management (BPM'07), pp. 88–95 (2007)
6. Chi, E.H.: A taxonomy of visualization techniques using the data state reference model. In: IEEE Symp on Inf Visualization, pp. 69–75 (2000)
7. Chiu, D.K., Cheung, S., Till, S., Karlapalem, K., Li, Q., Kafeza, E.: Workflow view driven cross-organizational interoperability in a Web service environment. *Inf. Technol. Manage.* **5**(3/4), 221–250 (2004)
8. Dadam, P., Reichert, M.: The ADEPT project: a decade of research and development for robust and flexible process support - challenges and achievements. *Comp. Sci. - Res. Develop.* **23**(2), 81–97 (2009)
9. Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., Berners-Lee, T.: RFC 2616-HTTP/1.1, the hypertext transfer protocol (1999)
10. Hipp, M., Mutschler, B., Michelberger, B., Reichert, M.: Navigating in process model repositories and enterprise process information. In: Proceedings of the 8th International IEEE Conference on Research Challenges in Information Science (RCIS'14), pp. 1–12. IEEE Computer Society, New York (2014)
11. Information Technology – Database Languages – SQL – Part 11: Information and Definition Schemas (SQL/Schemata). Norm ISO 9075:2011 (2011)
12. Johnson, R., Pearson, D., Pingali, K.: Finding regions fast: single entry single exit and control regions in linear time. In: Proceedings of the ACM SIGPLAN'93 (1993)
13. Kammerer, K., Kolb, J., Andrews, K., Bueringer, S., Meyer, B., Reichert, M.: User-centric process modeling and enactment: the Clavii BPM platform. In: Proceedings of the BPM Demo Session 2015 (BPMD 2015), no. 1418 in CEUR Workshop Proc, pp. 135–139. CEUR-WS.org (2015)
14. Kammerer, K., Kolb, J., Reichert, M.: PQL - a descriptive language for querying, abstracting, and changing process models. In: Proceedings of the 17th International Working Conference on Business Process Modeling, Development, and Support (BPMDS'15), no. 214 in LNBP, pp. 135–150. Springer, New York (2015)
15. Kolb, J.: Abstraction, Visualization, and Evolution of Process Models. Ph.D. thesis, Ulm University (2015)
16. Kolb, J., Reichert, M.: A flexible approach for abstracting and personalizing large business process models. *ACM Appl. Comp. Rev.* **13**(1), 6–17 (2013)
17. Kolb, J., Reichert, M.: Data flow abstractions and adaptations through updatable process views. In: 28th ACM Symp Applied Computing (SAC'13), pp. 1447–1453 (2013)
18. Kolb, J., Kammerer, K., Reichert, M.: Updatable process views for user-centered adaption of large process models. In: Proceedings of the 10th International Conference on Service Oriented Computing (ICSOC'12), pp. 484–498 (2012)
19. Marinacci, J.: Building Mobile Applications with Java: Using the Google Web Toolkit and PhoneGap. O'Reilly, Sebastopol (2012)
20. Mendling, J., Reijers, H.A., van der Aalst, W.M.P.: Seven process modeling guidelines (7PMG). *Inf. Software Technol.* **52**(2), 127–136 (2010)
21. Panzarino, O.: Learning Cypher. Packt Publishing, Birmingham (2014)
22. Parr, T.: The Definitive ANTLR 4 Reference. Pragmatic Bookshelf, Raleigh (2013)
23. Polyvyanyy, A., Smirnov, S., Weske, M.: The triconnected abstraction of process models. In: Proceedings of the 7th International Conference on Business Process Management (BPM'09), pp. 229–244. Springer, New York (2009)
24. Polyvyanyy, A., Weidlich, M., Weske, M.: Isotactics as a foundation for alignment and abstraction of behavioral models. In: Business Process Management, pp. 335–351. Springer, New York (2012)
25. Polyvyanyy, A., Ouyang, C., Barros, A., van der Aalst, W.M.P.: Process querying: enabling business intelligence through query-based process analytics. *Dec. Supp. Syst.* **100**, 41–56 (2017)
26. Rademakers, T.: *Activiti in Action: Executable Business Processes in BPMN 2.0*. Manning Publications, Shelter Island (2012)

27. Reichert, M., Dadam, P.: ADEPTflex - supporting dynamic changes of workflows without losing control. *J. Intell. Inf. Sys.* **10**(2), 93–129 (1998)
28. Reichert, M., Weber, B.: *Enabling Flexibility in Process-Aware Information Systems - Challenges, Methods, Technologies*. Springer, New York (2012)
29. Reichert, M., Kolb, J., Bobrik, R., Bauer, T.: Enabling personalized visualization of large business processes through parameterizable views. In: *Proceedings of the 27th ACM Symposium On Applied Computing (SAC'12)*, pp. 1653–1660. ACM Press, New York (2012)
30. Rinderle-Ma, S., Reichert, M., Weber, B.: On the formal semantics of change patterns in process-aware information systems. In: *Proceedings of the 27th International Conference on Conceptual Modeling (ER'08)*. LNCS, vol. 5231, pp. 279–293. Springer, New York (2008)
31. Robinson, I., Webber, J., Eifrem, E.: *Graph Databases*. O'Reilly, Sebastopol (2013)
32. Smirnov, S., Reijers, H.A., Weske, M., Nugteren, T.: Business process model abstraction: a definition, catalog, and survey. *Distrib. Parallel Databases* **30**(1), 63–99 (2012)
33. Streit, A., Pham, B., Brown, R.: Visualization support for managing large business process specifications. In: *BPM'05*, pp. 205–219 (2005)
34. Tran, H.: *View-Based and Model-Driven Approach for Process-Driven, Service-Oriented Architectures*. TU Wien, PhD thesis (2009)
35. Wagner, R.A., Fischer, M.J.: The string-to-string correction problem. *J. ACM* **21**(1), 168–173 (1974)
36. Weber, B., Reichert, M.: Refactoring process models in large process repositories. In: *Proceedings of the 20th International Conference on Advanced Information Systems Engineering (CAiSE'08)*, pp. 124–139 (2008)
37. Weber, B., Reichert, M., Rinderle, S.: Change patterns and change support features - enhancing flexibility in process-aware information systems. *Data Knowl. Eng.* **66**(3), 438–466 (2008)
38. Weber, B., Sadiq, S., Reichert, M.: Beyond rigidity - dynamic process lifecycle support: a survey on dynamic changes in process-aware information systems. *Comp. Sci. - Res. Develop.* **23**(2), 47–65 (2009)

# QuBPAL: Querying Business Process Knowledge



Maurizio Proietti, Francesco Taglino, and Fabrizio Smith

**Abstract** We present a query language, called QuBPAL, for retrieving knowledge from repositories of business processes represented in the BPAL (*Business Process Abstract Language*) framework. BPAL combines in a single modeling framework the procedural and the ontological aspects of business processes. This is done by providing a uniform, logic-based representation of both the workflow, with its associated procedural semantics, and the domain knowledge that captures the meaning of the entities participating in the process. This uniform representation is achieved by using Logic Programming (LP) as an intermediate language, to which we map BPMN models and OWL/RDF definitions. QuBPAL queries allow combining structural, behavioral, and domain-related knowledge and hence enable reasoning about the process from all these perspectives.

## 1 Introduction

Business Process (BP) modeling is the core of various methodologies and tools that support the automation of Business Process Management [30]. Indeed, a BP model represents the knowledge about a process in machine accessible form and may be used to simulate, analyze, and enact a process. In particular, many aspects of a BP, including performance and logical correctness, can be analyzed by querying the model by means of suitable query processing methods [21].

Most BP modeling languages and frameworks focus on the representation and analysis of the workflow graph, that is, they focus on how the BP activities are orchestrated from a procedural point of view, so as to verify whether the behavior of the process satisfies suitable properties (e.g., soundness [32]).

---

M. Proietti (✉) · F. Taglino  
CNR-IASI, Rome, Italy  
e-mail: [maurizio.proietti@iasi.cnr.it](mailto:maurizio.proietti@iasi.cnr.it); [francesco.taglino@iasi.cnr.it](mailto:francesco.taglino@iasi.cnr.it)

F. Smith  
United Technologies Research Center (UTRC), Rome, Italy  
e-mail: [fabrizio.smith@utrc.utc.com](mailto:fabrizio.smith@utrc.utc.com)

However, the workflow structure is not the only relevant aspect of a process that needs to be modeled. Indeed, the knowledge about the business domain where the process is carried out is often as important as its procedural behavior, and in many BP modeling frameworks, this type of knowledge is represented in an informal and implicit way, typically expressed through natural language comments and labels attached to the models. The lack of a formal representation of the domain knowledge within BP models is a strong limitation on the use of those models for the effective support of automated process management.

In order to overcome this limitation, various approaches have advocated the application to BP modeling of techniques based on computational ontologies [8, 11, 13, 33], which have also been shown to be fruitful in the related area of Web Services [4, 9]. These approaches have led to the concept of *Semantic Business Process*, where process-related and domain-specific ontologies provide formal, machine processable definitions for the basic entities involved in a process, such as activities, actors, data items, and the relations between them.

The approach followed in this chapter combines in a single modeling framework, called *Business Process Abstract Language* (BPAL), the procedural and the ontological aspects of a BP [26]. This is done by providing a logic-based representation of both the workflow, with its associated procedural semantics, and the domain knowledge that captures the meaning of the entities participating in the process. This representation is achieved by using Logic Programming (LP) [15] as an intermediate language, to which we map de-facto standards for BP modeling and ontology definition. Indeed, we do *not* define a new BP modeling language, but we provide a translation of the Business Process Model and Notation (BPMN) [17] and of the Ontology Web Language (OWL) [12] to Prolog.

The LP translation produces a *Business Process Knowledge Base* (BPKB) from a BPMN model where process elements are annotated with concepts of an OWL ontology. Then, we can reason about a BP, both from the procedural and ontological point of views, by querying the BPKB. To this aim, we define a *query language*, called QuBPAL, which answers queries that combine Prolog predicates representing properties of the process behavior, and OWL  $< \textit{subject}, \textit{property}, \textit{object} >$  triples representing domain-related knowledge.

In this chapter, we present a brief overview of the BPAL framework and of the associated QuBPAL query language. The chapter is organized as follows. In Sect. 2, we present the BPAL framework [7, 26], which provides a logical representation of the workflow structure and the procedural semantics. BPAL copes with a significant fragment of the BPMN 2.0 specification. Then, we describe how BP elements are annotated with semantic concepts and properties defined using an OWL 2 RL ontology [16]. In Sect. 3, we introduce the QuBPAL query language. We present its syntax, which is a variant of SPARQL, a widely accepted query language for the Semantic Web [23], and its semantics, defined in terms of a translation to LP queries. In Sect. 4, we present the use cases supported by the BPAL framework and the associated QuBPAL query language. In Sect. 5, we describe the implementation of BPAL and we present its graphical interface, which is intended to ease the interaction with the user, and the mapping to XSB Prolog queries [29]. In Sect. 6, we

put BPAL in relation with the Process Querying Framework [21]. Finally, in Sect. 7, we discuss future developments and perspectives.

## 2 Business Process Knowledge Base

In this section, we explain how we formally represent (repositories of) business processes by means of the notion of a *Business Process Schema* (BPS) [25]. A BPS, its meta-model, its procedural (or *behavioral*) semantics, and its ontology-based semantics, are specified by sets of rules, for which we adopt the standard notation and semantics of logic programming (see, for instance, [15]). In particular, a *term* is either a variable, or a constant, or a function applied to terms. Variable names have uppercase initials, while constant and function names have lowercase initials. An *atom* is a formula of the form  $p(t_1, \dots, t_m)$ , where  $t_1, \dots, t_m$  are terms. A *rule* is of the form  $A \leftarrow L_1 \wedge \dots \wedge L_n$ , where  $A$  is an *atom* and  $L_1, \dots, L_n$  are *literals*, i.e., atoms or negated atoms. If  $n = 0$  we call the rule a *fact*. A rule (or an atom, or a literal) is *ground* if no variables occur in it. A *logic program* is a finite set of rules.

The rest of the section is structured in the following subsections. In Sect. 2.1, we show how a BPS is specified. Then, in Sect. 2.2, we present a formal definition of the behavioral semantics of a BPS. Finally, in Sect. 2.3, we describe how BPAL enables the semantic annotation of a BPS, i.e., the association of the elements of a BPS with the concepts of an ontology that captures the knowledge about the domain where the process is executed.

### 2.1 Business Process Schemas

We show how a BPS is specified with the help of an example. Let us consider the BP *po* depicted in Fig. 1, where the handling of a purchase *order* is represented using BPMN. The process starts with the *ordering* activity, which is performed by the sales department (*sales\_dpt*). A purchase order document is created (*create\_order*) and checked (*check\_order*). If the order is approved, the *approve\_order* activity is executed, while if it is rejected the *cancel\_order* activity is performed. Furthermore, if the order needs to be completed, the *modify\_order* activity is executed, and the order is checked again. A label on an arc identifies the condition to traverse that arc. For instance, *c1*, *c2*, and *c3* identify the conditions that the order is approved, rejected, and modified, respectively.

If the *ordering* activity times out (this is signaled by the *exc* event), or at the end of the *ordering* activity the purchase order results to be rejected (i.e., condition *c2* holds), the rejection of the order is notified to the customer (*notify\_rejection*). Otherwise, if the order has been approved (i.e., condition *c1* holds), the *supplier* verifies the availability of the items specified in the order (*check\_inventory*), and produces a list of items (*parts\_list*) containing also information about their



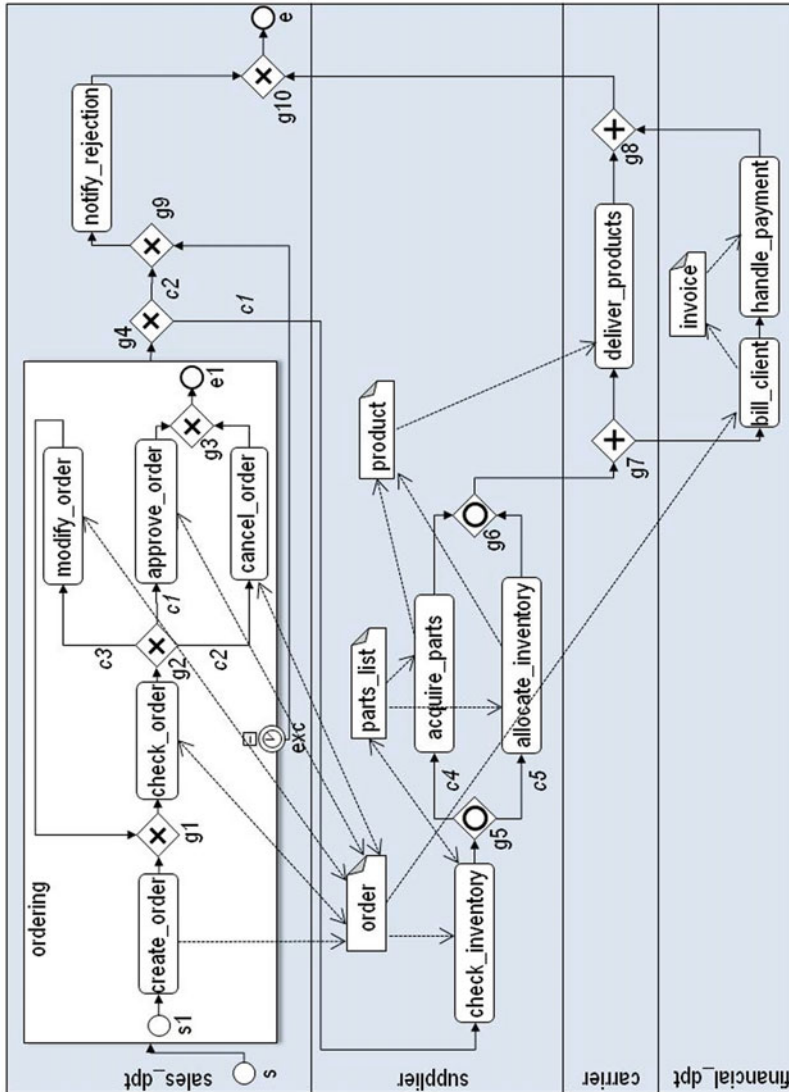


Fig. 1 The purchasing order process *po*

availability in the inventory. If some items are currently unavailable (condition  $c_4$ ), the *acquire\_parts* activity is executed. If some items are available (condition  $c_5$ ), the *allocate\_inventory* activity is executed. Once all the items are at disposal of the *supplier*, the requested *product* is available and delivered by the *carrier* (*deliver\_products*). At the same time, the financial department (*financial\_dpt*) creates the invoice (*bill\_client*) and manages the payment (*handle\_payment*).

A BP (e.g., *po*) consists of a set of *flow elements* (e.g., activities, events, and gateways) and *relations* between them (e.g., flows). A BP is associated with a unique *start event* and a unique *end event*, which are flow elements that represent the entry point and the exit point, respectively, of the process. An *activity* is a unit of work performed within the process. A *task* is an atomic activity (e.g., *approve\_order*), i.e., no further decomposable, while a *compound activity* is associated with a process that provides the definition of its internal structure (e.g., *ordering*). An *intermediate event* is an event that occurs between the start and the end of the process (e.g., the *time-out exception*, namely *exc*, attached to the *ordering* activity). The order of execution of the flow elements is specified by the *sequence flow* relation (corresponding to solid arrows). The *branching/merging* of the control flow is specified using three types of *gateways*: *exclusive* (XOR, e.g.,  $g_1$  and  $g_2$ ), *inclusive* (OR, e.g.,  $g_5$  and  $g_6$ ), and *parallel* (AND, e.g.,  $g_7$  and  $g_8$ ). The *item flow* relation (corresponding to a dotted arrow) specifies that a flow element gets as *input* or produces as *output* a particular *item*, i.e., a data object. For instance, *approve\_order* has *order* as input, and *bill\_client* has *invoice* as output. A *participant* is a role associated with a *lane* (e.g., *sales\_dpt*) or a *pool* (a collection of lanes, not shown in our example).

Other entities usually employed to model processes, such as *messages*, are not presented here for lack of space. However, BPAL can represent most constructs based on the BPMN specification [17].

A BPS is a formal representation of a BP, which is specified in BPAL by a set of ground facts of the form  $p(c_1, \dots, c_n)$ , where  $c_1, \dots, c_n$  are constants denoting BP elements (e.g., flow elements, items, and participants), and  $p$  is a predicate symbol taken from the BPAL vocabulary. BPAL also includes a set of *rules* that represent *meta-model* properties of a BPS, defining a number of structural properties that a BPS should satisfy as a directed graph, where edges correspond to sequence and item flow relations. Two categories of structural properties should be satisfied by a *well-formed* (i.e., syntactically correct) BPS: (i) *local* properties related to its elementary components (e.g., every task must have at most one ingoing and at most one outgoing sequence flow), and (ii) *global* properties related to the overall structure of the BPS (e.g., every flow element must lie on a path from the *start* to the *end* event).

In Table 1, we report a partial list of BPAL predicates, which are sufficient to get an idea of how a BPS is specified.

Table 2 shows the representation of the *ordering* compound activity, which is a sub-process of *po*, by means of the BPAL predicates. The rest of the BPS is omitted for reasons of space.

**Table 1** BPS and meta-model predicates

Predicate	Meaning
$bp(p, s, e)$	$p$ is a business process (or sub-process) with start event $s$ and end event $e$
$flow\_el(x)$	$x$ is either an event or an activity or a gateway
$event(x)$	$x$ is either a start event (i.e., $start\_ev(x)$ ) or an intermediate event (i.e., $int\_ev(x)$ ) or an end event (i.e., $end\_ev(x)$ )
$activity(x)$	$x$ is either an atomic task (i.e., $task(x)$ ) or a compound activity (i.e., $comp\_act(x, s, e)$ , for some $s$ and $e$ )
$gateway(x)$	$x$ is either a parallel branch (i.e., $par\_branch(x)$ ), or an exclusive branch (i.e., $exc\_branch(x)$ ), or an inclusive branch (i.e., $inc\_branch(x)$ ), or a parallel merge (i.e., $par\_merge(x)$ ), or an exclusive merge (i.e., $exc\_merge(x)$ ), or an inclusive merge (i.e., $inc\_merge(x)$ )
$item(x)$	$x$ is a data object
$participant(x)$	$x$ is a role associated with a lane or a pool
$seq(x, y, p)$	there is a sequence flow from $x$ to $y$ in process $p$
$it\_flow(a, d, p)$	activity $a$ has data item $d$ as input (i.e., $input(a, d, p)$ ) or output (i.e., $output(a, d, p)$ )
$assigned(a, r, p)$	activity $a$ is assigned to participant $r$ in process $p$
$exception(e, a, p)$	exception $e$ is associated with activity $a$ in process $p$
$occurs(x, p)$	the flow element $x$ belongs to process $p$
$c\_seq(cond, x, y, p)$	there is a sequence flow from $x$ to $y$ in process $p$ with associated condition $cond$
$reachable(x, y, p)$	there is a path of sequence flows from $x$ to $y$ in $p$ , or in a compound activity of $p$
$wf\_sub\_process(p, s, e)$	$s$ and $e$ are the boundary flow elements of a single-entry-single-exit (SESE) region [20] of process $p$
$str\_sub\_proc(p, s, e)$	$s$ and $e$ are the boundary elements of a structured [19] sub-process of $p$

## 2.2 Behavioral Semantics

This section presents a formal definition of the behavioral semantics, or *enactment*, of a BPS, by following an approach inspired by the *Fluent Calculus*, a well-known calculus for action and change (see [31] for an introduction). In the Fluent Calculus, the *state* of the world is represented as a collection of *fluents*, i.e., terms representing atomic properties that hold at a given instant of time.

An action, also represented as a term, may cause a change of state, i.e., an update of the collection of fluents associated with it. Finally, a *plan* is a sequence of actions that leads from the initial to the final state.

A fluent is an expression of the form  $f(a_1, \dots, a_n)$ , where  $f$  is a fluent symbol and  $a_1, \dots, a_n$  are constants or variables. In order to model the behavior of a

**Table 2** A BPS representing the *ordering* sub-process from Fig. 1

Activity	Gateway	Event
<i>comp_act</i> ( <i>ordering</i> , <i>s1</i> , <i>e1</i> )	<i>exc_merge</i> ( <i>g1</i> )	<i>start_ev</i> ( <i>s1</i> )
<i>task</i> ( <i>create_order</i> )	<i>exc_branch</i> ( <i>g2</i> )	<i>end_ev</i> ( <i>e1</i> )
<i>task</i> ( <i>check_order</i> )	<i>exc_merge</i> ( <i>g3</i> )	<i>int_ev</i> ( <i>exc</i> )
<i>task</i> ( <i>approve_order</i> )		
<i>task</i> ( <i>cancel_order</i> )		
<i>task</i> ( <i>modify_order</i> )		
Sequence flow	Item	Item flow
<i>seq</i> ( <i>s1</i> , <i>create_order</i> , <i>ordering</i> )	<i>item</i> ( <i>order</i> )	<i>output</i> ( <i>create_order</i> , <i>order</i> , <i>ordering</i> )
<i>seq</i> ( <i>create_order</i> , <i>g1</i> , <i>ordering</i> )		<i>input</i> ( <i>check_order</i> , <i>order</i> , <i>ordering</i> )
<i>seq</i> ( <i>g1</i> , <i>check_order</i> , <i>ordering</i> )		<i>output</i> ( <i>check_order</i> , <i>order</i> , <i>ordering</i> )
<i>seq</i> ( <i>check_order</i> , <i>g2</i> , <i>ordering</i> )		<i>input</i> ( <i>approve_order</i> , <i>order</i> , <i>ordering</i> )
<i>c_seq</i> ( <i>c1</i> , <i>g2</i> , <i>approve_order</i> , <i>ordering</i> )		<i>output</i> ( <i>approve_order</i> , <i>order</i> , <i>ordering</i> )
<i>c_seq</i> ( <i>c2</i> , <i>g2</i> , <i>cancel_order</i> , <i>ordering</i> )		<i>input</i> ( <i>cancel_order</i> , <i>order</i> , <i>ordering</i> )
<i>c_seq</i> ( <i>c3</i> , <i>g2</i> , <i>modify_order</i> , <i>ordering</i> )		<i>output</i> ( <i>cancel_order</i> , <i>order</i> , <i>ordering</i> )
<i>seq</i> ( <i>modify_order</i> , <i>g1</i> , <i>ordering</i> )		<i>input</i> ( <i>modify_order</i> , <i>order</i> , <i>ordering</i> )
<i>seq</i> ( <i>approve_order</i> , <i>g3</i> , <i>ordering</i> )		<i>output</i> ( <i>modify_order</i> , <i>order</i> , <i>ordering</i> )
<i>seq</i> ( <i>cancel_order</i> , <i>g3</i> , <i>ordering</i> )		
<i>seq</i> ( <i>g3</i> , <i>e1</i> , <i>ordering</i> )		

BPS, we represent states as *finite sets* of ground fluents. We take a closed-world interpretation of states, that is, we assume that a fluent  $F$  holds in a state  $S$  iff  $F \in S$ . This set-based representation of states relies on the assumption that the BPS is *safe*, i.e., during its enactment there are no concurrent executions of the same flow element [32]. This assumption enforces the set of states reachable by a given BPS to be finite. The safeness assumption can be relaxed by representing the states as *multisets* of fluents, but then the finiteness of the state space is no longer guaranteed.

A *fluent expression* is built inductively from fluents, the binary function symbol *and*, and the unary function symbol *not*. The satisfaction relation assigns a truth value to a fluent expression with respect to a state. This relation is encoded by a predicate  $holds(F, S)$ , which holds if the fluent expression  $F$  is true in state  $S$ :

- $holds(F, S) \leftarrow F = true;$
- $holds(F, S) \leftarrow F \in S;$
- $holds(not(F), S) \leftarrow \neg holds(F, S);$
- $holds(and(F_1, F_2), S) \leftarrow holds(F_1, S) \wedge holds(F_2, S).$

We consider the following two kinds of fluents:  $cf(E_1, E_2, P)$ , which means that the flow element  $E_1$  has been executed and the flow element  $E_2$  is waiting for execution, during the enactment of process  $P$  (*cf* stands for *control flow*);  $en(A, P)$ , which means that the activity  $A$  is being executed during the enactment of the process  $P$  (*en* stands for *enacting*). To clarify our terminology note that, when a flow element  $E_2$  is waiting for execution,  $E_2$  might not be enabled to execute, because other conditions need to be satisfied, such as those depending on the synchronization with other flow elements (for instance, in the presence of merge gateways).

We assume that the execution of an activity has a beginning and a completion, while the other flow elements execute instantaneously. Thus, we consider two kinds of actions: *begin*( $A$ ), which starts the execution of an activity  $A$ , and *complete*( $E$ ), which represents the completion of the execution of a flow element  $E$  (possibly, an activity). The change of state determined by the execution of a flow element is formalized by a relation  $result(S_1, A, S_2)$ , which holds if the result of performing action  $A$  in state  $S_1$  leads to state  $S_2$ . For defining the relation  $result(S_1, A, S_2)$ , we use the following auxiliary predicates: (i)  $update(S_1, T, U, S_2)$ , which holds if  $S_2 = (S_1 \setminus T) \cup U$ , where  $S_1, T, U$ , and  $S_2$  are sets of fluents, and (ii)  $setof(F, C, S)$ , which holds if  $S$  is the set of ground instances of fluent  $F$  such that condition  $C$  holds. The relation  $r(S_1, S_2)$  holds if state  $S_2$  is a *successor* of state  $S_1$ , that is, there is a *begin* or *complete* action  $A$  that can be executed in state  $S_1$  leading, in one step, to state  $S_2$ :

$$r(S_1, S_2) \leftarrow result(S_1, A, S_2).$$

We say that a state  $S_2$  is *reachable* from a state  $S_1$  if there is a finite sequence of actions (of length  $\geq 0$ ) from  $S_1$  to  $S_2$ , that is,  $reachable\_state(S_1, S_2)$  holds, where the relation  $reachable\_state$  is the reflexive-transitive closure of  $r$ .

In order to get an idea of the fluent-based formalization of the behavioral semantics of a BPS, we show the case of task execution. The beginning of the execution of an atomic task  $A$  in process  $P$  is modeled by adding the  $en(A, P)$  fluent to the state:

$$(T1) \quad result(S_1, begin(A), S_2) \leftarrow task(A) \wedge holds(cf(X, A, P), S_1) \wedge update(S_1, \{cf(X, A, P)\}, \{en(A, P)\}, S_2)$$

At the completion of  $A$ , the  $en(A, P)$  fluent is removed and the control flow moves on to the unique successor of  $A$ :

$$(T2) \quad result(S_1, complete(A), S_2) \leftarrow task(A) \wedge holds(en(A, P), S_1) \wedge seq(A, Y, P) \wedge update(S_1, \{en(A, P)\}, \{cf(A, Y, P)\}, S_2)$$

The BPAL framework provides a fluent-based formalization of the behavioral semantics of a BPS by focusing on a core of the BPMN language, including events, compound activities, parallel, inclusive, and exclusive gateways. More details can be found in [26].

The execution of a process is modeled as an *execution trace*, i.e., a sequence of actions of the form  $[act_1(a_1), \dots, act_n(a_n)]$ , where, for  $i = 1, \dots, n$ ,  $act_i$  is either *begin* or *complete*. Thus, a trace is a formal representation of a process log. The predicate  $trace(s_1, t, s_2)$  defined in terms of the predicate  $result$  holds if  $t$  is a sequence of actions that lead from state  $s_1$  to state  $s_2$ . The predicate  $c\_trace(t, p)$  holds if  $t$  is a *correct trace* of a BPS  $p$ , that is,  $t$  is a trace that leads from the initial state (containing the fluent  $cf(start, s, p)$ , where  $s$  is the start event of  $p$ ) to the final state of  $p$  (containing the fluent  $cf(e, end, p)$ , where  $e$  is the end event of  $p$ ).

In order to provide a general verification mechanism for behavioral properties, the BPAL framework implements a model checking methodology based on the

temporal logic CTL (*Computation Tree Logic*, see [5] for a comprehensive overview, and [10, 14] for its application in the area of BP verification).

The validity of a CTL temporal formula  $\varphi$  in a state  $s$  is defined by the predicate  $holds(\varphi, s)$ . For instance, we have the following:

- $holds(ex(\varphi), s)$ :  $\varphi$  holds at the successor state of  $s$ ;
- $holds(eu(\varphi_1, \varphi_2), s)$ : there exists a sequence of states starting from  $s$ , where  $\varphi_1$  holds until  $\varphi_2$  holds;
- $holds(ag(\varphi), s)$ :  $\varphi$  holds at every states reachable from  $s$ ;
- $holds(af(\varphi), s)$ : for every sequence of states starting from  $s$ ,  $\varphi$  eventually holds.

Then, we can define behavioral properties like *precedence* and *response* using CTL formulas as follows:

- $prec(\varphi_1, \varphi_2, p)$ :  $\varphi_1$  precedes  $\varphi_2$ , i.e., every state where  $\varphi_2$  holds is always preceded by one where  $\varphi_1$  holds. This predicate is defined by the formula  $holds(not(eu(not(\varphi_2), and(\varphi_1, not(\varphi_2))))), initial(p))$ , where  $initial(p)$  is the term denoting the initial state of  $p$ ;
- $resp(\varphi_1, \varphi_2, p)$ :  $\varphi_2$  responds to  $\varphi_1$ , i.e., every state where  $\varphi_1$  holds is eventually followed by one where  $\varphi_2$  holds. This predicate is defined by the formula  $holds(ag(imp(\varphi_1, af(\varphi_2))), initial(p))$ , where  $imp$  denotes logical implication.

### 2.3 Semantic Annotations

The BPAL framework enables an explicit representation of the domain knowledge regarding the entities involved in a BP, i.e., the business environment in which the process is carried out. This is done by allowing *semantic annotations* to enrich the procedural knowledge specified by a BPS with domain knowledge expressed in terms of a given *business reference ontology* (BRO). In Table 3, an excerpt of a

**Table 3** Business reference ontology excerpt

$Purchase\_Order \sqsubseteq AccountingDocument$	$AvailablePL \sqsubseteq PartList$
$SubmittedPO \sqsubseteq Purchase\_Order$	$UnavailablePL \sqsubseteq PartList$
$PassedPO \sqsubseteq Purchase\_Order$	$UnavailablePL \sqcap AvailablePL \sqsubseteq \perp$
$RejectedPO \sqsubseteq Purchase\_Order$	$RejectedPO \sqcap PassedPO \sqsubseteq \perp$
$IncompletePO \sqsubseteq Purchase\_Order$	$PassedPO \sqcap IncompletePO \sqsubseteq \perp$
$DeliveredPO \sqsubseteq Purchase\_Order$	$RejectedPO \sqcap IncompletePO \sqsubseteq \perp$
$Invoice \sqsubseteq AccountingDocument$	$Product \sqsubseteq Value\_Object$
$EmittedInvoice \sqsubseteq Invoice$	$Refusal \sqsubseteq Communication$
$PaidInvoice \sqsubseteq Invoice$	$Carrier \sqsubseteq TransportationCompany$
$DeliveredPO \sqcap \exists related.PaidInvoice \sqsubseteq FulfilledPO$	$Supply \sqsubseteq Logistics$
$CorporateCustomer \sqsubseteq BusinessPartner$	$Transportation \sqsubseteq Logistics$
$recipient \sqsubseteq related$	$contents \sqsubseteq related$

BRO is reported. We use standard notation from Description Logics [3]:  $\sqsubseteq$  denotes concept inclusion,  $\perp$  denotes the empty concept (interpreted as the empty set of individuals),  $\sqcap$  denotes concept intersection, and  $\exists$  denotes existential quantification. Table 3 shows some axioms of the taxonomy (e.g., *Purchase\_Order* is a kind of *AccountingDocument*), and also some constraints between concepts, such as their disjointness (e.g., any individual which is an instance of *RejectedPO* cannot be an instance of *PassedPO*).

Annotations provide two kinds of ontology-based information: (i) formal definitions of the basic entities involved in a process (e.g., activities, actors, items), which specify their meaning in an unambiguous way (*terminological* annotations), and (ii) specifications of preconditions and effects of the enactment of tasks, as well as conditions associated with sequence flows (*functional* annotations).

In Sect. 2.3.1 we briefly recall the fragment of an ontology language used in BPAL for representing the semantics of a business domain. Then, in Sects. 2.3.2 and 2.3.3, we show how to specify terminological and functional annotations, respectively.

### 2.3.1 Rule-Based Ontologies

A BRO is intended to capture the semantics of a business domain in terms of the relevant vocabulary plus a set of axioms, i.e., the *Terminological Box* (TBox), which defines the intended meaning of the vocabulary terms. To represent the semantic annotations and the behavioral semantics of a BPS in a uniform way, we formalize ontologies as sets of rules. In particular, we consider a fragment of OWL falling within the OWL 2 RL profile [16], which is an upward-compatible extension of RDF and RDFS whose semantics is defined via a set of Horn rules (that is, rules without negated atoms in their body), called OWL 2 RL/RDF rules. OWL 2 RL ontologies are modeled by means of the ternary predicate  $t(s, p, o)$  representing an OWL statement with subject  $s$ , predicate  $p$ , and object  $o$ . For instance, the assertion  $t(a, rdfs:subClassOf, b)$  represents the inclusion axiom  $a \sqsubseteq b$ . Reasoning on triples is supported by OWL 2 RL/RDF rules of the form  $t(s, p, o) \leftarrow t(s_1, p_1, o_1) \wedge \dots \wedge t(s_n, p_n, o_n)$ . For instance, the rule  $t(A, rdfs:subClassOf, B) \leftarrow t(A, rdfs:subClassOf, C) \wedge t(C, rdfs:subClassOf, B)$  defines the transitive closure of the subsumption relation.

### 2.3.2 Terminological Annotations

A *terminological annotation* associates elements of a BPS with concepts of the BRO, in order to describe the former in terms of a suitable conceptualization of the underlying business domain provided by the latter. This association is specified by a set of OWL assertions of the form  $BpsEl : \exists termRef.Concept$ , where:

**Table 4** Terminological annotation examples

BPS element	Terminological annotation
<i>order</i>	$\exists termRef.Purchase\_Order$
<i>invoice</i>	$\exists termRef.Invoice$
<i>sales_dpt</i>	$\exists termRef.Sales\_Department$
<i>carrier</i>	$\exists termRef.Carrier$
<i>deliver_products</i>	$\exists termRef.(Transportation \sqcap \exists related.Product)$
<i>notify_rejection</i>	$\exists termRef.(Refusal \sqcap \exists content.Purchase\_Order \sqcap \exists recipient.CorporateCustomer)$

- *BpsEl* is an element of a BPS;
- *Concept* is either i) a named concept defined in the ontology, (for instance, *Purchase\_Order*), or ii) a complex concept, defined by a class expression (for instance,  $Refusal \sqcap \exists content.Purchase\_Order$ );
- *termRef* is an OWL property name.

*Example 1* In Table 4, we list examples of annotations of the *po* process (Fig. 1) using concepts of the BRO shown in Table 3. The *order* and *invoice* items are annotated with the *Purchase\_Order* and the *Invoice* concepts, respectively, while the *sales\_dpt* and *carrier* participants with the *Sales\_Department* and the *Carrier* concepts, respectively. The *deliver\_products* task is defined as a *Transportation* related to a *Product* (e.g., the products which are the *contents* of the delivered packages). Finally, the *notify\_rejection* task is annotated as a *Refusal*, which is a *Communication* (e.g., through an e-mail message) of a *Purchase\_Order* to a *CorporateCustomer*.

### 2.3.3 Functional Annotations

By using the ontology vocabulary and axioms, we define semantic annotations for modeling the behavior of individual process elements in terms of *preconditions* under which a flow element can be executed and *effects* on the state of the world after its execution. Preconditions and effects can be used, for instance, to model input/output relations of activities with data items, which is the standard way of representing information handling in BPMN diagrams. Fluents can represent the *status* of a data item affected by the execution of an activity at a given time during the execution of the process. A precondition specifies the status a data item must possess when an activity is enabled to start, and an effect specifies the status of a data item after having completed the activity. Moreover, in order to select the successors of decision points, we can associate fluent expressions with the arcs outgoing from inclusive or exclusive branch gateways.

Functional annotations are formulated by means of the following three relations:

- $pre(A, C, P)$ , which specifies that fluent expression *C*, called *enabling condition*, must hold to execute element *A* in process *P*;



- $eff(A, E^-, E^+, P)$ , which specifies that fluents  $E^-$ , called *negative effects*, do not hold after the execution of element  $A$ , and that fluents  $E^+$ , called *positive effects*, hold after the execution of  $A$  in process  $P$ . We assume that  $E^-$  and  $E^+$  are disjoint sets;
- $c\_seq(C, B, Y, P)$ , which specifies a *conditional sequence flow*:  $C$  is a condition associated with the exclusive or inclusive branch gateway  $B$ , i.e., a fluent expressions that must hold in order to enable flow element  $Y$ , successor of  $B$  in process  $P$ .

In the presence of functional annotations, the enactment of a BPS is modeled as follows. Given a state  $S_1$ , a flow element  $A$  can be enacted if  $A$  is waiting for execution according to the control flow semantics, and its enabling condition  $C$  is satisfied, i.e.,  $holds(C, S_1)$  is true. Moreover, given an annotation  $eff(A, E^-, E^+, P)$ , when  $A$  is completed in a given state  $S_1$ , then a new state  $S_2$  is obtained by taking out from  $S_1$  the set  $E^-$  of fluents and then adding the set  $E^+$  of fluents. We need to check that effects satisfy a *consistency* condition which guarantees that: (i) no contradiction can be derived from the fluents of  $S_2$  by using the state independent axioms of the reference ontology, and (ii) no fluent belonging to  $E^-$  holds in  $S_2$ .

The state update is formalized by extending the *result* relation so as to take into account the *pre* and *eff* relations. We only consider the case of task execution. The other cases are similar and are omitted.

$$\begin{aligned}
 result(S_1, begin(A), S_2) &\leftarrow task(A) \wedge holds(cf(X, A, P), S_1) \wedge pre(A, C, P) \wedge \\
 &holds(C, S_1) \wedge update(S_1, \{cf(X, A, P)\}, \{en(A, P)\}, S_2) \\
 result(S_1, complete(A), S_2) &\leftarrow task(A) \wedge holds(en(A, P), S_1) \wedge \\
 &eff(A, E^-, E^+, P) \wedge seq(A, Y, P) \wedge \\
 &update(S_1, \{en(A, P)\} \cup E^-, \{cf(A, Y, P)\} \cup E^+, S_2)
 \end{aligned}$$

The enabling conditions and the negative and positive effects occurring in functional annotations are fluent expressions built from fluents of the form  $t_f(s, p, o)$ , corresponding to the OWL statement  $t(s, p, o)$ , where we adopt the usual *rdf*, *rdfs*, and *owl* prefixes for names in the OWL vocabulary, and the *bro* prefix for names relative to the business reference ontology of our running example. We assume that the fluents appearing in functional annotations are either of the form  $t_f(a, rdf : type, c)$ , corresponding to the unary atom  $c(a)$ , or of the form  $t_f(a, p, b)$ , corresponding to the binary atom  $p(a, b)$ , where  $a$  and  $b$  are *individuals*, while  $c$  and  $p$  are concepts and properties, respectively, defined in the BRO. Thus, fluents correspond to assertions about individuals that represent the *Assertion Box* (ABox) of the ontology. Hence, the ABox may change during process enactment due to the effects specified by the functional annotations, while the ontology definitions and axioms, i.e., the *TBox* of the ontology, do not change.

The semantics of inclusive and exclusive branches is extended to evaluate the associated conditions. We show the case of exclusive branches. The other is similar.

$$\begin{aligned}
 result(S_1, complete(B), S_2) &\leftarrow exc\_branch(B) \wedge holds(cf(A, B, P), S_1) \wedge \\
 &c\_seq(G, B, C, P) \wedge holds(G, S_1) \wedge update(S_1, \{cf(A, B, P)\}, \{cf(B, C, P)\}, S_2)
 \end{aligned}$$

**Table 5** Functional annotations for the *po* process

Flow element	Enabling condition	Effects
create_order		$E^+$ : $t_f(o, rdf : type, bro : SubmittedPO)$
check_order	$t_f(o, rdf : type, bro : SubmittedPO)$	$E^+$ : $t_f(o, rdf : type, bro : PassedPO)$ $E^-$ : $t_f(o, rdf : ype, bro : SubmittedPO)$
check_order	$t_f(o, rdf : type, bro : SubmittedPO)$	$E^+$ : $t_f(o, rdf : type, bro : RejectedPO)$ $E^-$ : $t_f(o, rdf : ype, bro : SubmittedPO)$
check_order	$t_f(o, rdf : type, bro : SubmittedPO)$	$E^+$ : $t_f(o, rdf : type, bro : IncompletePO)$ $E^-$ : $t_f(o, rdf : ype, bro : SubmittedPO)$
modify_order	$t_f(o, rdf : type, bro : IncompletePO)$	$E^+$ : $t_f(o, rdf : type, bro : SubmittedPO)$ $E^-$ : $t_f(o, rdf : ype, bro : IncompletePO)$
check_inventory	$t_f(o, rdf : type, bro : PassedPO)$	$E^+$ : $t_f(a, rdf : type, bro : AvailablePL)$
check_inventory	$t_f(o, rdf : type, bro : PassedPO)$	$E^+$ : $t_f(u, rdf : type, bro : UnavailablePL)$
check_inventory	$t_f(o, rdf : type, bro : PassedPO)$	$E^+$ : $t_f(a, rdf : type, bro : AvailablePL)$ , $t_f(u, rdf : type, bro : UnavailablePL)$
acquire_parts	$t_f(u, rdf : type, bro : UnavailablePL)$	$E^+$ : $t_f(p, rdf : type, bro : Product)$
allocate_inventory	$t_f(a, rdf : type, bro : AvailablePL)$	$E^+$ : $t_f(p, rdf : type, bro : Product)$
bill_client	$t_f(o, rdf : type, bro : PassedPO)$	$E^+$ : $t_f(i, rdf : type, bro : EmittedInvoice)$ , $t_f(o, bro : related, i)$
handle_payment	$t_f(i, rdf : type, bro : EmittedInvoice)$	$E^+$ : $t_f(i, rdf : type, bro : PaidInvoice)$ $E^-$ : $t_f(i, rdf : type, bro : EmittedInvoice)$
deliver_products	$t_f(o, rdf : type, bro : PassedPO)$ , $t_f(p, rdf : type, bro : Product)$	$E^+$ : $t_f(o, rdf : type, bro : DeliveredPO)$

**Table 6** Conditions associated with exclusive and inclusive branch gateways

Gateway	Target	Condition
g2	approve_order	c1: $t_f(o, rdf : type, bro : PassedPO)$
g2	cancel_order	c2: $t_f(o, rdf : type, bro : RejectedPO)$
g2	modify_order	c3: $t_f(o, rdf : type, bro : IncompletePO)$
g4	check_inventory	c1: $t_f(o, rdf : type, bro : PassedPO)$
g4	g9	c2: $t_f(o, rdf : type, bro : RejectedPO)$
g5	acquire_parts	c4: $t_f(a, rdf : type, bro : UnavailablePL)$
g5	allocate_inventory	c5: $t_f(u, rdf : type, bro : AvailablePL)$

Let us now present an example of specification of functional annotations. In particular, our example shows nondeterministic effects, that is, a case where a flow element *A* is associated with more than one pair ( $E^-$ ,  $E^+$ ) of negative and positive effects.

*Example 2* Tables 5 and 6 specify the functional annotations for the *po* process of Fig. 1. These annotations determine the behavior of the process.

The positive effect  $t_f(o, rdf : type, bro : SubmittedPO)$  of the *create\_order* activity (see Table 5) states that, when the activity is executed, an order *o* is created and

$o$  is a *SubmittedPO*. The result of the subsequent *check\_order* task is that order  $o$  is no longer a *SubmittedPO* (see the negative effect of *check\_order* in Table 5) and  $o$  is either approved (i.e.,  $o$  is a *PassedPO*), or rejected (i.e.,  $o$  is a *RejectedPO*), or it is incomplete (i.e.,  $o$  is an *IncompletePO*). The *approve\_order*, *cancel\_order*, and *modify\_order* tasks are executed if the order  $o$  is either a *PassedPO*, *RejectedPO*, or *IncompletePO*, respectively. The selection of those tasks is the result of associating conditions  $c1$ ,  $c2$ , and  $c3$ , with the arcs outgoing from  $g2$ . Conditions  $c1$  and  $c2$  associated with the arcs outgoing from  $g4$  enforce that if the order is approved then the *check\_inventory* task is executed, and if the order is rejected then the *notify\_rejection* task is executed. The effect of the *check\_inventory* task is the production of either a list of parts that are available in the inventory ( $a$  is an *AvailablePL*) or a list of parts that are not available in the inventory ( $u$  is an *UnavailablePL*), or both. The fluents expressing the effects of *check\_inventory* are also associated with the sequence flows outgoing from the inclusive branch  $g5$ , and enable either the *allocate\_inventory* task or the *acquire\_parts* task, or both. The effect of *allocate\_inventory* and/or *acquire\_parts* is that a *Product*  $p$  is made available. Finally, the product is delivered by the *deliver\_products* activity, which has a *DeliveredPO* effect, and in parallel, an invoice is emitted (an *EmittedInvoice* is an effect of *bill\_client*) and the payment is managed (a *PaidInvoice* is an effect of *handle\_payment*).

In order to evaluate a statement of the form  $holds(t_f(s, p, o), x)$ , where  $t_f(s, p, o)$  is a fluent and  $x$  is a state, the definition of the *holds* predicate given previously must be extended to take into account the axioms belonging to the BRO. Indeed, we want that a fluent of the form  $t_f(s, p, o)$  be true in state  $X$  not only if it belongs to  $X$ , but also if it can be inferred from the fluents in  $X$  and the axioms of the ontology. This is done by adding extra rules, derived from the OWL 2 RL/RDF rules, which infer  $holds(t_f(s, p, o), x)$  atoms from the fluents holding in a state and the axioms of the ontology. For instance, for concept subsumption, we add the rule:

$$holds(t_f(S, rdf : type, C), X) \leftarrow holds(t_f(S, rdf : type, B), X) \wedge t(B, rdfs : subclassOf, C).$$

### 3 Querying the Business Process Knowledge Base

A repository of BPs is represented by a *Business Process Knowledge Base* (BPKB), i.e., a collection of BPSs together with the facts and rules formalizing the meta-model, the behavioral semantics, the business reference ontology, and the semantic annotations. A BPKB represents a rich knowledge from various perspectives: (1) the structural perspective, modeled by the business process schema, (2) the ontology-based domain knowledge, modeled by the semantic annotations, and (3) the procedural perspective, modeled by the rule-based behavioral semantics. To query and reason about a BPKB with respect to all the above mentioned perspectives, we

have defined the QuBPAL query language, through which the various retrieval and reasoning services are delivered.

QuBPAL queries are SELECT-FROM-WHERE statements, whose syntax is inspired by SPARQL, a query language for RDF data stores developed by the World Wide Web Consortium, which is widely accepted in the Semantic Web community [23]. QuBPAL can be considered as an extension of SPARQL, in the sense that SPARQL queries are expressed in terms of RDF triples, while QuBPAL queries can be made out of conjunctions of  $t(s, p, o)$  triples together with any other predicate defined in the BPKB. QuBPAL queries are translated into Logic Programming (LP) queries, and then evaluated by using the underlying Prolog inference engine.

In Sect. 3.1, we present the syntax of the QuBPAL query language. Then, in Sect. 3.2, we describe the translation into LP queries that defines the semantics of QuBPAL queries. Finally, in Sect. 3.3, we illustrate the use of the language through some example queries.

### 3.1 Syntax

The syntax of a QuBPAL query is shown in Table 7.

The symbol  $\epsilon$  denotes the empty sequence. Identifiers prefixed by question marks (i.e.,  $?bpId$  and  $?x$ ) denote variables. The SELECT statement defines the output of query evaluation, which is specified using the following constructs:

- an optional *process identifier*, denoted by  $<?bpId >$ ;
- a (possibly empty) sequence  $?x^*$  of variables occurring in the WHERE statement.

When the SELECT statement has the empty sequence of variables, the query returns either *true* or *false*. The FROM statement indicates the process(es) from which data are to be retrieved. If it is omitted, the whole repository is considered; otherwise one can specify a non-empty sequence of process identifiers  $<bpId >^+$ .

The WHERE statement specifies an expression denoting a property which the set of data returned by the query must satisfy. This expression is a sentence built from:

- the set of the predicates defined in the BPKB;
- the logical connectives AND, OR, NOT, and the equality predicate =, with the standard logic semantics;
- another QuBPAL query, enclosed in curly brackets, thereby allowing nested queries, which may return values for the variables occurring in their SELECT statement.

**Table 7** QuBPAL syntax

SELECT ( $\epsilon$   $<?bpId >$ ) $?x^*$
( $\epsilon$   FROM $<bpId >^+$ )
WHERE <i>expression</i>

The arguments of the predicates appearing in the WHERE statement of a query are:

- variables, which can be semantically typed (e.g.,  $?x :: \textit{Concept}$ ); at most one process variable occurs in the statement;
- individual constants;
- complex terms, such as fluent expressions and lists.

## Predicates of the WHERE Statement

The predicates used in the WHERE statement of a QuBPAL query allow the definition of complex properties combining structural, ontological, and behavioral properties. Below, we list some of these predicates.

*BPS Predicates.* Any predicate used to specify a BPS and its meta-model properties, and in particular the ones of Table 1 (see Sect. 2.1).

*Semantic Annotation Predicates.* Any predicate specifying a semantic (terminological or functional) annotation (see Sects. 2.3.2 and 2.3.3). In particular, in order to query terminological annotations, we introduce the predicate  $\textit{sigma}(x, c)$ , meaning that  $x$  is annotated with some concept  $a$ , via the OWL property  $\textit{termRef}$ , and  $a$  is a subclass of  $c$ . Instances of the  $\textit{sigma}(x, c)$  predicate are automatically generated by semantically typed variables. For instance, the semantic typing  $?x :: \textit{Concept}$  is translated into  $\textit{sigma}(X, \textit{'Concept'})$ . Functional annotations can be queried using the following predicates:

- $\textit{enabling\_cond}(e, f, p)$ : fluent  $f$  is an enabling condition of element  $e$  in process  $p$ ;
- $\textit{pos\_eff}(e, f, p)$ : fluent  $f$  is a positive effect of element  $e$  in process  $p$ ;
- $\textit{neg\_eff}(e, f, p)$ : fluent  $f$  is a negative effect of element  $e$  in process  $p$ .

*Triple Predicates.* Any ternary predicate of the form  $t(s, p, o)$  representing an OWL 2 RL/RDF statement with subject  $s$ , predicate  $p$  and object  $o$  (see Sect. 2.3.1).

*Behavioral Predicates.* Any predicate that defines a property of the behavioral semantics of a process, such as the predicates  $\textit{holds}(f, s)$ ,  $\textit{reachable\_state}(s_1, s_2)$ ,  $\textit{c\_trace}(t, p)$ ,  $\textit{prec}(f_1, f_2, p)$ , and  $\textit{resp}(f_1, f_2, p)$  defined in Sect. 2.2. Other behavioral predicates are the following:

- $\textit{all\_traces}(t, p, n)$ :  $t$  is a trace from the initial state of  $p$  of length at most  $n$ ;
- $\textit{deadlock}(s, p)$ :  $s$  is a deadlock state, that is, a non-final state of  $p$  with no successor;
- $\textit{dead\_act}(a, p)$ :  $a$  is a dead activity, that is, an activity that cannot be executed in any enactment of  $p$ ;
- $\textit{imp\_comp}(s, p)$ :  $s$  is a final state of  $p$  where improper completion holds, i.e., some activity is being executed;
- $\textit{opt\_com}(p)$ : process  $p$  satisfies the *option to complete* property, i.e., from any reachable state, it is possible to complete the process;

- *inconsistent(s, p)*: *s* is a state of *p* that violates an integrity constraint defined in the ontology (i.e., *holds(false, s)*);
- *non\_exec(a, p)*: *a* is an activity that can be reached by the control flow in a state of *p* where its enabling condition is not satisfied;
- *always\_in\_final(c, p)*: every possible execution of *p* eventually completes with a final state satisfying the concept *c*;
- *act\_prec(a<sub>1</sub>, a<sub>2</sub>, p)*: in all possible executions of *p*, the enactment of activity *a<sub>1</sub>* precedes the enactment of activity *a<sub>2</sub>*, that is, the predicate *prec(en(a<sub>1</sub>, p), en(a<sub>2</sub>, p), p)* holds;
- *act\_resp(a<sub>1</sub>, a<sub>2</sub>, p)*: in all possible executions of *p*, the enactment of activity *a<sub>1</sub>* is eventually followed by the enactment of activity *a<sub>2</sub>*, that is, the predicate *resp(en(a<sub>1</sub>, p), en(a<sub>2</sub>, p), p)* holds.

### 3.2 Semantics

The semantics of a QuBPAL query is formally defined by its translation into an LP query, which is similar to the translation of SQL into Datalog [1]. Now, we describe this translation for a subset of the legal queries. The extension to the general case is straightforward.

A SELECT-FROM-WHERE statement *St* corresponds to a predicate *q*, defined by a set of clauses  $\{q(V_1, \dots, V_m) \leftarrow body_1, \dots, q(V_1, \dots, V_m) \leftarrow body_n\}$ . *V<sub>1</sub>, ..., V<sub>m</sub>* correspond to the variables occurring in the SELECT statement of *St*. (Each LP variable *V<sub>i</sub>*, for *i* = 1, ..., *m*, corresponds to a QuBPAL variable *?v<sub>i</sub>*.) Each clause corresponds to a process (or sub-process) identifier appearing in the FROM statement of *St*, and the variables occurring in the *process selector* of *St* are bound to the constants specified in the FROM statement, as shown in Table 8.

The connective AND is directly translated to the logical conjunction “∧”. The connective OR generates one clause for each disjunct (see Table 9).

Nested queries are simply replaced by an atom that represents the head of the clauses obtained by translating the nested query, as shown in Table 10, where we assume that *q<sub>nest</sub>* has only one clause.

Suppose that the expression in the WHERE statement contains a negated predicate NOT *pred*[ $\vec{?x}$ ,  $\vec{?y}$ ], where  $\vec{?y}$  is the tuple of variables occurring in predicate

**Table 8** FROM statement translation pattern. Variable *P* is omitted from the arguments of *q* if  $\langle ?p \rangle$  does not appear in the SELECT statement

<b>SELECT</b> $\langle ?p \rangle ?x_1 \dots ?x_{m-1}$ <b>FROM</b> $\langle p_1 \rangle \dots \langle p_n \rangle$ <b>WHERE</b> <i>pred</i>
$q(P, X_1, \dots, X_{m-1}) \leftarrow P = p_1 \wedge bp(P, S, E) \wedge pred$
...
$q(P, X_1, \dots, X_{m-1}) \leftarrow P = p_n \wedge bp(P, S, E) \wedge pred$

**Table 9** Conjunctive and disjunctive queries translation pattern

---

<b>SELECT ... FROM ... WHERE</b> $pred_1$ AND ( $pred_2$ OR $pred_3$ )
$q(\dots) \leftarrow \dots \wedge pred_1 \wedge q_{or}(\dots)$
$q_{or}(\dots) \leftarrow pred_2$
$q_{or}(\dots) \leftarrow pred_3$

---

**Table 10** Nested queries translation pattern

---

<b>SELECT ... FROM ... WHERE</b> $pred_1$ AND { <b>SELECT</b> $\langle ?p \rangle ?x$ ... <b>FROM</b> ... <b>WHERE</b> $pred_2$ }
$q(\dots) \leftarrow \dots \wedge pred_1 \wedge q_{nest}(P, X, \dots)$
$q_{nest}(P, X, \dots) \leftarrow \dots \wedge pred_2$

---

**Table 11** Negated queries translation pattern

---

<b>SELECT ... FROM ... WHERE</b> $pred_1[\vec{?x}]$ AND NOT $pred_2[\vec{?x}, \vec{?y}]$
$q(\vec{X}, \dots) \leftarrow \dots \wedge pred_1[\vec{X}] \wedge \neg q_{neg}(\vec{X})$
$q_{neg}(\vec{X}) \leftarrow pred_2[\vec{X}, \vec{Y}]$

---

$pred[\vec{?x}, \vec{?y}]$  and not elsewhere in the statement. If  $pred[\vec{?x}, \vec{?y}]$  is an atomic predicate and  $\vec{?y}$  is the empty tuple, then NOT  $pred[\vec{?x}]$  is translated to the negative literal  $\neg pred[\vec{X}]$ . Otherwise, if  $pred[\vec{?x}, \vec{?y}]$  is a non-atomic predicate, or  $\vec{?y}$  is not empty, we replace the negated predicate by a literal  $\neg q_{neg}(\vec{X})$ , and we introduce the clause  $q_{neg}(\vec{X}) \leftarrow pred[\vec{X}, \vec{Y}]$  (see, e.g., Table 11).

It has been shown [28] that given any QuBPAL query  $Q$  not containing trace predicates (such as  $c\_trace$  or  $all\_traces$ ), if  $Q$  can be translated into a *non-floundering* LP query (i.e., an LP query that does not call a non-ground negative subgoal [15]), then  $Q$  terminates in polynomial time with respect to  $|Q| \times |\mathcal{S}|$  (using Prolog with *tabling* [29]), where  $|Q|$  is the size of the query and  $|\mathcal{S}|$  is the cardinality of the set of reachable states. (Note that  $\mathcal{S}$  is a finite set, because a state is a finite set of fluents and each fluent is a ground term  $f(c_1, \dots, c_n)$ , where  $f, c_1, \dots, c_n$  are taken from a finite set of symbols.) Thus, querying BPKBs with respect to behavioral predicates is practically feasible only when the number of activities that can be executed in parallel during process enactments, and hence the set of reachable states, is not too high.

### 3.3 Query Examples

In this section, we present some examples of queries over a BPKB. We provide a natural language description of the query, its formulation in QuBPAL, and the

corresponding translation into LP clauses. Then, we present the answer to the query with respect to the BP  $po$  depicted in Fig. 1.

*Example 3.1* Retrieve all the *Transportation* activities assigned to a *TransportationCompany*.

### QuBPAL query

```
SELECT ?a
FROM <po>
WHERE activity(?a::bro:Transportation) AND
       assigned(?a, ?c::bro:TransportationCompany, ?p)
```

### LP translation

$$q(A) \leftarrow P = po \wedge activity(A) \wedge sigma(A, 'bro : Transportation') \wedge assigned(A, C, P) \wedge sigma(C, 'bro : TransportationCompany').$$

where  $A, P$  are the LP variables that translate the QuBPAL variables  $?a, ?p$ , respectively, and  $po$  is the LP constant that denotes the process identifier occurring in the FROM statement.

The answer to this query is the *deliver\_products* activity from Fig. 1. Indeed, this activity is annotated with a concept (see Example 1) which is a subclass of the *Transportation* concept (see Table 3), and it is assigned to the *carrier* participant, which is annotated with the *Carrier* concept (see Example 1), a specialization of the *TransportationCompany* concept (see Table 3). Note that variables  $?c$  and  $?p$  do not appear in the SELECT statement, and hence no value is returned for them in the answer.

*Example 3.2* The following query retrieves all pairs  $(a1, a2)$  of activities such that  $a1$  generates a *SubmittedPO* (see Table 5 for the functional annotations of the activities),  $a2$  is a *Communication* activity (see Example 1 for the terminological annotations), and  $a2$  is reachable from  $a1$  via a sequence flow path of the  $po$  process.

### QuBPAL query

```
SELECT ?a1 ?a2
FROM <po>
WHERE reachable(?a1, ?a2, ?p) AND
       activity(?a1) AND
       pos_eff(?a1, ?f1, ?p) AND
       ?f1=tf(?o, rdf:type, bro:SubmittedPO) AND
       activity(?a2::bro:Communication)
```

### LP translation

$$q(A1, A2) \leftarrow P = po \wedge reachable(A1, A2, P) \wedge activity(A1) \wedge pos\_eff(A1, F1, P) \wedge F1 = tf(O, 'rdf : type', 'bro : SubmittedPO') \wedge activity(A2) \wedge sigma(A2, 'bro : Communication').$$



The answer to this query consists of two pairs of activities, namely (*create\_order*, *notify\_rejection*) and (*modify\_order*, *notify\_rejection*). Indeed, the *create\_order* and *modify\_order* tasks are the only activities that generate a *SubmittedPO*, and the *notify\_rejection* task, which is annotated by a *Refusal* concept, is the only *Communication* activity in the *po* process (note that *Refusal* is a specialization of *Communication* in the ontology of Table 3). Moreover, there is a sequence flow path from *create\_order* to *notify\_rejection*, and one from *modify\_order* to *notify\_rejection*. Indeed, in the definition of the *reachable* predicate, we assume that the successor *g4* of the compound activity *ordering* is also a successor of the *end* event *e1* of that activity.

*Example 3.3* The following query searches for all the tasks that take as input a *Purchase\_Order* and precede the *deliver\_products* task in any possible execution of the process *po*.

### QuBPAL query

```
SELECT ?t
FROM <po>
WHERE task(?t) AND
      input(?t, ?i::bro:Purchase_Order, ?p) AND
      act_prec(?t, deliver_products, ?p)
```

### LP translation

$$q(T) \leftarrow P = po \wedge bp(P, S, E) \wedge task(T) \wedge input(T, I, P) \wedge \sigma(I, 'bro : Purchase\_Order') \wedge act\_prec(T, deliver\_products, P).$$

This query returns the tasks *check\_order*, *approve\_order*, and *check\_inventory*, which are the only tasks that take as input a *Purchase\_Order* and must be completed in all executions of the BP *po* that reach the *deliver\_products* task. Indeed, in order for *deliver\_products* to be executed, the fluent  $t_f(o, rdf:type, bro:PassedPO)$ , which is one of the possible effects of *check\_order*, must be true. This fluent must hold when *approve\_order* is executed (see condition *c1* in Fig. 1). Obviously, *check\_inventory* must also be executed, as it lies on all paths from the start event to *deliver\_products*. In contrast, the *cancel\_order* and *modify\_order* activities might not be executed, in particular, when the effect produced by the *check\_order* activity is not  $t_f(o, rdf:type, bro:PassedPO)$ .

The queries shown in Examples 3.1–3.3 are solved by the BPAL reasoner in at most 30 ms each on an Intel Core i7-7560U, equipped with 2.40 GHz×4 CPU and 16 GB RAM. This time refers to a BPKB representing the *po* process and includes: (i) the time spent to translate the query from the SELECT-FROM-WHERE format to the LP representation and (ii) the time spent for the visualization of the answer in the BPAL Graphical User Interface (see Sect. 5 for more details on the implementation of the BPAL framework).

More realistic experiments have been conducted in a pilot of the European Project BIVÉE<sup>1</sup> and in the context of a collaboration between the Italian National Research Council (CNR) and SOGEI (ICT Company of the Italian Ministry of Finance). The former is related to the modeling of production processes in manufacturing oriented networked enterprises, while the latter regards the procedural modeling of legislative decrees in the tax domain. The experiments revealed good efficiency for very sophisticated queries over small and medium sized BPs (about one hundred of activities and several thousands of reachable states) [27].

## 4 Use Cases

The end-user tools provided by the BPAL platform (i.e., the software platform that implements the BPAL framework) allow the semantic annotation of existing BP models as well as the modeling of BPs from scratch. Furthermore, reasoning capabilities over the BPKB are made available through the QuBPAL querying mechanism.

Figure 2 summarizes the use cases supported by the BPAL platform. They are briefly described in the rest of the section. A non-directed arc connecting a user and a use case means that the user can perform that use case. A directed dotted arc, labeled as *include*, from a use case to another use case means that the execution of the former requires the execution of the latter. Finally, a solid line ending with a triangle from a use case to another means that the former is a specialization of the latter.

*Model BP* The platform provides functionalities to create (*Create BP* use case), edit (*Edit BP* use case) and view (*View BP* use case) business processes in terms of the BPMN notation. The choice of providing a support for BPMN is motivated by the wide adoption and acceptance of BPMN that is witnessed by the huge number of modeling tools that implementing it. However, while BPAL has been strongly inspired by BPMN, its constructs are common to a large variety of process and workflow languages, e.g., BPEL [2] and UML activity diagrams [18]. Hence, a relevant non-functional requirement that has been considered during the implementation is the extensibility of the platform to provide support for importing BPs designed using other notations/formats into a BPAL repository.

*Import BP* Besides the possibility of creating a business process from scratch, the BPAL platform offers functionalities to import a BPMN file. In particular, the import is allowed for those BPMN serializations which are XML files containing only the structural description of the process, leaving out all the graphical details; this format, supported, e.g., by Intalio Process Modeler,<sup>2</sup> has been also used in several research

---

<sup>1</sup> BIVÉE: Business Innovation and Virtual Enterprise Environment (FoF-ICT-2011.7.3-285746).

<sup>2</sup> <http://www.intalio.com/process-designer>.

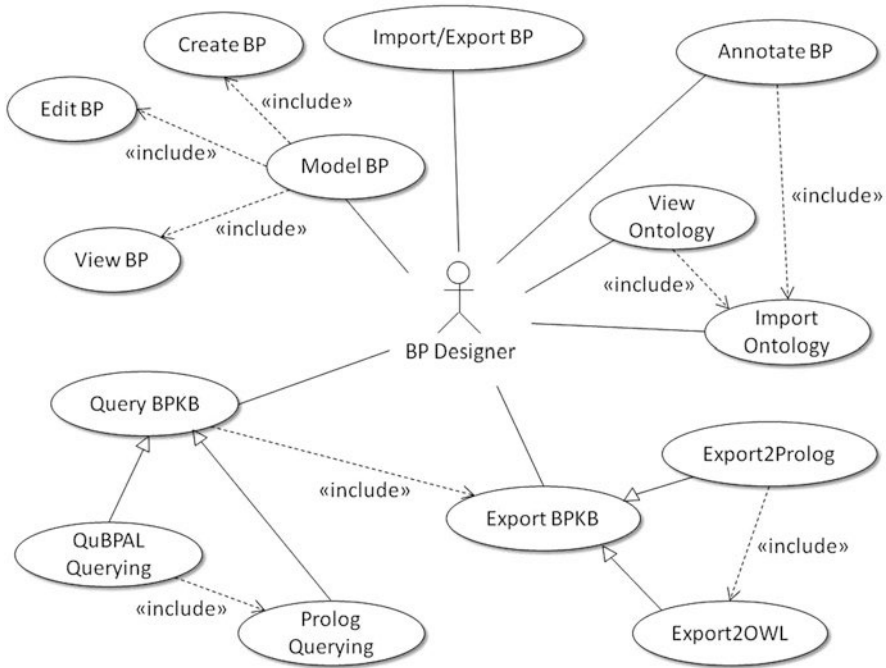


Fig. 2 Use cases supported by the BPAL platform

projects since it is adopted by the Eclipse SOA Tools Platform,<sup>3</sup> an open-source BPMN editor.

**Annotate BP.** The support provided for the semantic annotation includes: (i) associating BPS elements with OWL expressions built in terms of a reference ontology through the *sigma* predicate (*terminological annotation*), and (ii) associating flow elements of a BPS with preconditions and effects in terms of ontology-based expressions (*functional annotation*).

**Import Ontology.** OWL/RDF ontologies can be loaded in the platform in order to define terminological and functional annotations.

**View Ontology.** An ontology imported in the platform can be shown in a hierarchical view to support the annotation and querying activities.

**Export BPKB.** The platform supports the generation of formal logic-based representations of a business process. In particular, it allows the export to OWL (*Export2OWL* use case), and to Prolog (*Export2Prolog* use case).

**Query BPKB.** The platform enables reasoning over a business process knowledge base by posing queries either in the QuBPAL language (*QuBPAL Querying* use case) or in Prolog syntax (*Prolog Querying* use case).

<sup>3</sup> <http://www.eclipse.org/bpmn/>.

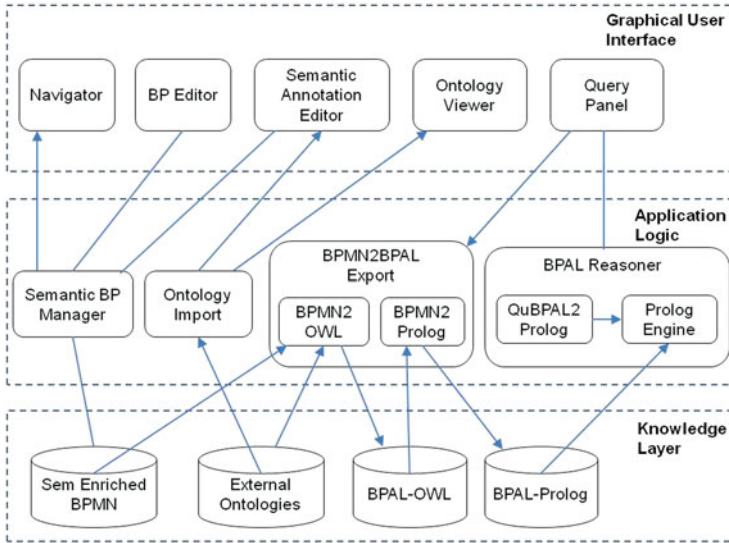


Fig. 3 The BPAL platform macro architecture

## 5 Implementation

The BPAL platform<sup>4</sup> is a prototypical implementation of the framework discussed so far, and it is implemented as an Eclipse Plug-in.<sup>5</sup> Below, we outline the architectural aspects and the main implementation choices.

The BPAL platform is organized as a three-tier (i.e., Graphical User Interface, Application Logic, and Knowledge Layer) application. Figure 3 sketches these three layers. Boxes with rounded corners represent main functional components, while cylinders represent knowledge repositories. Furthermore, arrows represents dataflow. In particular, directed arrows emphasize the main flow direction, while non-directed arrows indicate that the flow can occur in both directions.

Next, we define each of these tiers.

### 5.1 Graphical User Interface

This component provides the graphical user interface to define a BPKB and to interact with the BPAL Reasoner. A screen-shot of the main components of the GUI is depicted in Fig. 4.

<sup>4</sup> <http://saks-wiki.iasi.cnr.it/xwiki/bin/view/Tools/BPAL/public>.

<sup>5</sup> <http://www.eclipse.org/>.

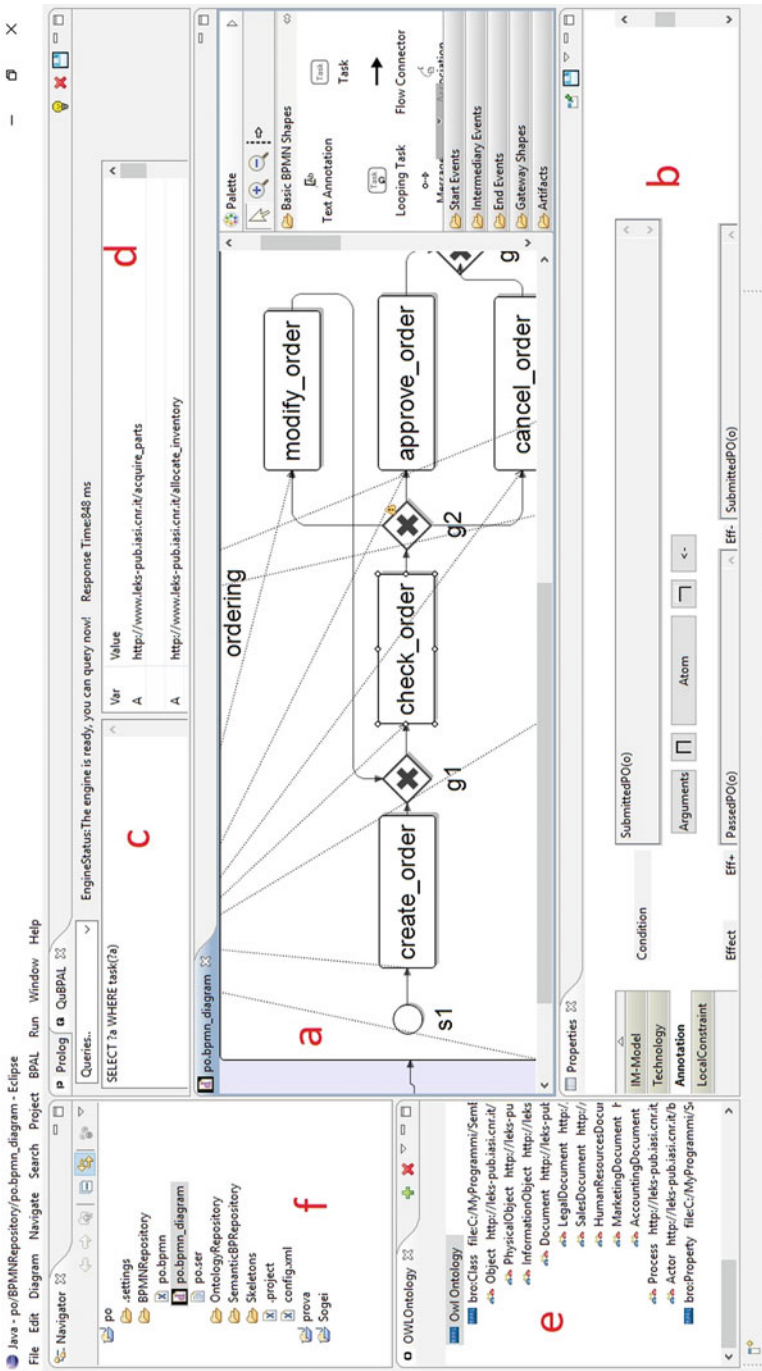


Fig. 4 The graphical user interface of the BPAL platform

- The *BP Editor* (Fig. 4a) is based on the STP BPMN Modeler,<sup>6</sup> which comprises a set of tools to model business process diagrams using BPMN.
- The *Semantic Annotation Editor* (Fig. 4b) allows the annotation of process elements with respect to a reference ontology.
- The *Query Panel* provides a prompt to access the BPAL reasoner through the querying mechanism. In particular, in Fig. 4c and d, the QuBPAL tab panel is shown, with query definition and query results on the left and right side, respectively. As an alternative, a Prolog panel, from where one can make any Prolog query, can be opened through the Prolog tab.
- The *Ontology Viewer* (Fig. 4e) allows the visualization of OWL ontologies. This panel is particularly helpful for BPS annotation and query definition.
- The *Navigator* (Fig. 4f), provides a tree view of the resources available in the workspace, such as BP schemas and ontologies.

## 5.2 Application Logic

This layer includes all the components that implement the functionalities of the platform that correspond to the use cases presented in Sect. 4. The main components are:

- *Semantic BP Manager*, which is in charge of storing all the updates of the BPKB.
- *Ontology Import*, which allows an OWL ontology to be loaded in the platform in order to define terminological and functional annotations.
- *BPMN2BPAL Export*, which translates semantically annotated BPMN processes into BPAL. Two BPAL serializations are allowed: OWL/RDF and Prolog, generated by the *BPMN2OWL* and the *BPMN2Prolog* sub-modules, respectively. In particular, while the export to OWL starts directly from the *.bpmn* file, the export to Prolog takes as input the OWL representation.
- *BPAL Reasoner*, which enables the resolution of queries over the Prolog representation. If a query is expressed in QuBPAL, the *QuBPAL2Prolog* sub-module translates the query to Prolog. A Prolog query is taken as input by the *Prolog Engine* sub-module, which is in charge of compiling the queries and collecting the results of the evaluations. In the current implementation, the core of the *Prolog Engine* is XSB,<sup>7</sup> a Logic Programming and deductive database system. The OWL representation can also be directly queried by giving it as input to an OWL reasoner or by loading it into a triple store<sup>8</sup> and making SPARQL queries [23].

---

<sup>6</sup> [https://wiki.eclipse.org/index.php?title=SOA/BPMN\\_Modeler](https://wiki.eclipse.org/index.php?title=SOA/BPMN_Modeler).

<sup>7</sup> <http://xsb.sourceforge.net/>.

<sup>8</sup> A triple store is a database suitable for managing OWL/RDF ontologies.

### 5.3 Knowledge Layer

This layer represents the knowledge repository of the BPAL platform, and it is organized as follows:

- *Semantically Enriched BPMN*, which contains the serialization of the business processes (.bpmn files), and their terminological and functional annotations (.ser files).
- *External Ontologies*, which contains the OWL ontologies that can be used to define terminological and functional annotations.
- *BPAL-OWL*, which contains the BPAL ontology that defines the constructs of the BPAL meta-model, and the OWL representation of the semantically enriched processes (OWL export).
- *BPAL-Prolog*, which contains the Prolog code for the BPKB rules, and the Prolog export of the processes and ontologies that are needed to resolve queries both on the structure and the execution of business processes.

## 6 Framework

The aim of this section is to put the BPAL platform in relation with the Process Querying Framework (PQF) proposed in [21], by identifying the PQF components which are addressed, and to what extent, by the BPAL platform. PQF is a reference abstract architecture whose components are intended to be selectively replaced in order to devise new process querying methods. PQF distinguishes between *active components* and *passive components*. The former identify functionalities performed by the querying methods, while the latter identify input and output objects. Furthermore, the framework organizes the components into four logical groups. In the following, the four groups are briefly outlined, and the functionalities of the BPAL platform that correspond to the PQF components of each group are recalled.

*Model, Simulate, Record, and Correlate* This group collects the components responsible for the acquisition and construction of process models, as well as for the design and formalization of process queries.

The BPAL platform provides functionalities for *modeling* business processes by using BPMN. In particular, the platform allows the user to graphically design, or import, BPMN processes models. Furthermore, process models can be semantically enriched by associating process components with terminological and functional annotations in the form of logic-based expressions built in terms of a reference ontology.

At present, the BPAL framework does not offer direct support to recording event logs, simulating, and correlating processes. However, some of the BPAL functionalities could be useful for those tasks, as we briefly discuss below.

- *Recording* event logs can be realized by representing logs as lists of events. Then, we can use predicates over traces, such as *c\_trace* and *all\_traces*, to reason about event logs (e.g., to check *conformance* with respect to a given process model).
- *Simulating* non-deterministic executions of processes can be achieved by querying the BPKB using trace predicates, like the above mentioned *c\_trace* and *all\_traces*. Indeed, by leveraging the features of Prolog execution, these predicates can be used not only to check that a given trace is a legal enactment of a process model, but also to generate traces that satisfy the properties specified by the model.
- *Correlating* different processes can take advantage of both the behavioral and ontology-based predicates provided by BPAL. By using behavioral predicates one can prove or discover properties relating process executions. Moreover, by annotating processes using the same reference ontology one can realize terminological alignments between process components, possibly belonging to different processes.

Concerning the *definition* and *formalization* of queries, the BPAL platform accepts queries both in Prolog and in QuBPAL, a user friendly SELECT-FROM-WHERE language. For queries in the QuBPAL syntax, a translation to Prolog is performed.

*Prepare* This group collects components responsible for making process repositories ready to resolve queries in an efficient manner. With respect to that, the BPAL platform supports *caching* facilities by leveraging on the *tabling* mechanisms of the XSB system. In fact, tabling avoids redundant subcomputations, and by doing so guarantees the termination of evaluation also for recursive predicates such as *reachable\_state* (see the end of Sect. 3.2 for a brief discussion of the termination of QuBPAL queries).

*Execute* This group includes components responsible for executing queries over process repositories. In the BPAL platform, the component that is in charge of resolving queries on business processes is the *BPAL Reasoner*, which is built upon the XSB system (see Sect. 5.2). *Filtering* is ensured by the FROM constraints specified in the QuBPAL query, which allows a user to restrict the processing of the query over a subset of all the processes in the repository.

*Interpret* This group collects the components in charge of processing query results, e.g., for presentation and explanation purposes. With respect to that, the BPAL platform can exploit the features of the underlying Logic Programming engine to explain and inspect the result of a query. For instance, since Prolog computes bindings for the free variables appearing in a query, BPAL can extract a trace that satisfies, or violates, a given property of the behavior of a process, on the basis of the result of a query concerning such a property.



## 7 Conclusions and Future Work

In this chapter, we presented QuBPAL, the query language of the BPAL platform, a framework for semantic BP modeling. The BPAL platform provides a graphical user interface to assist the user in the definition and interrogation of a BP Knowledge Base. We discussed how functionalities for modeling, semantically annotating, and querying BPs are made available by the tool, and how knowledge about the process can be retrieved and processed via QuBPAL queries. The main design choices have been oriented at guaranteeing both a sound foundation and a high level of practical usability. The soundness of the foundation is guaranteed by the logic-based approach underpinning the BPAL framework. The practical usability derives from the support of widely used and accepted standards and technologies. Indeed, we adopted BPMN as a graphical modeling notation, and its XML textual format to import and manipulate BP models, possibly designed through external BP management systems. For the domain knowledge representation, we use OWL/RDF, the current de-facto standard for ontology modeling and metadata exchange. The whole platform is packaged as an Eclipse plug-in, which extends the Eclipse STP BPMN Modeler, an established open-source BPMN editor. The querying and reasoning engine is built on top of a standard LP engine (XSB).

The results have been quite encouraging in practice. Indeed, the rule-based implementation of the OWL reasoner and the effective tabled evaluation mechanism of the XSB engine have demonstrated good response time and scalability, for small and medium-sized repositories of BP models [27].

In the literature, several approaches have been proposed for dealing with the three main perspectives of process knowledge: graph-matching for structural querying, model checking for behavioral verification, description logics for domain knowledge representation (see also various chapters in this book). Each of them proposes techniques that have been proven effective with respect to the specific aspect they address. The goal of the BPAL approach is to manage a BPKB which organizes and stores the conceptual knowledge about the three aforementioned perspectives, and it allows inference over this structure in a uniform and formal framework.

Several directions of research deserve further investigation.

On the technical level, we would like to incorporate query optimization techniques to enhance the reasoning approach. As it stands, the reasoner performs only simple optimizations based on the re-ordering of literals, and all the queries are evaluated with a purely goal-oriented, top-down approach, without any pre-processing of the knowledge base. We are confident that the query evaluation process can be strongly improved through more sophisticated query rewriting techniques, which have been largely investigated in the area of Logic Programming.

We also think that the BPAL framework could be used in other phases of the BP lifecycle, besides modeling and analysis at design time. In particular, the trace semantics of BPAL appears a suitable starting point to support: (i) querying at runtime, i.e., over a running instance of the process during its enactment, and (ii) querying a-posteriori, i.e., over the execution logs of completed enactments. Fur-

thermore, an approach based on using QuBPAL for querying BP repositories can be a valid support for process reuse and composition, as discussed in recent work [24].

Finally, we plan to extend the BPAL framework to model other aspects of process knowledge, such as constraints on data and time duration. Initial steps for the enhancement of our approach towards that direction have been already taken [6, 22].

**Acknowledgments** We would like to thank Artem Polyvyanyy for careful reading of drafts of this chapter and helpful suggestions for improvement. Maurizio Proietti is a member of the INdAM-GNCS National Research Group.

## References

1. Abiteboul, S., Hull, R., Vianu, V.: *Foundations of Databases*. Addison-Wesley, Boston (1995)
2. Andrews, T., Curbera, F., Dholakia, H., Goland, Y., Klein, J., Leymann, F., Liu, K., Roller, D., Smith, D., Thatte, S., Trickovic, I., Weerawarana, S.: BPEL4WS, business process execution language for Web services version 1.1. IBM (2003). <http://download.boulder.ibm.com/ibmdl/pub/software/dw/specs/ws-bpel/ws-bpel.pdf>
3. Baader, F., Calvanese, D., McGuinness, D.L., Nardi, D., Patel-Schneider, P.F. (eds.): *The Description Logic Handbook: Theory, Implementation, and Applications*. Cambridge University Press, Cambridge (2003)
4. Burstein, M., et al.: *OWL-S: Semantic Markup for Web Services*. W3C Member Submission (2004). <http://www.w3.org/Submission/OWL-S/>
5. Clarke, E.M., Grumberg, O., Peled, D.A.: *Model Checking*. The MIT Press, Cambridge (1999)
6. De Angelis, E., Fioravanti, F., Meo, M.C., Pettorossi, A., Proietti, M.: Verification of time-aware business processes using constrained Horn clauses. In: *Proceedings of the 26th International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR 2016)*. LNCS, vol. 10184 (2017)
7. De Nicola, A., Missikoff, M., Proietti, M., Smith, F.: An open platform for business process modeling and verification. In: *Database and Expert Systems Applications, 21st International Conference*. LNCS, vol. 6261, pp. 76–90. Springer, New York (2010)
8. Di Francescomarino, C., Ghidini, C., Rospocher, M., Serafini, L., Tonella, P.: Semantically-aided business process Modeling. In: *International Semantic Web Conference*. LNCS, vol. 5823, pp. 114–129. Springer, New York (2009)
9. Fensel, D., Lausen, H., Polleres, A., de Bruijn, J., Stollberg, M., Roman, D., Domingue, J.: *Enabling Semantic Web Services: The Web Service Modeling Ontology*. Springer, New York (2006)
10. Fu, X., Bultan, T., Su, J.: Analysis of interacting BPEL Web services. In: *Proceedings of the International Conference on World Wide Web*, pp. 621–630. ACM, New York (2004). <http://doi.acm.org/10.1145/988672.988756>
11. Hepp, M., Leymann, F., Domingue, J., Wahler, A., Fensel, D.: Semantic business process management: a vision towards using semantic Web services for business process management. In: *Proceedings of International Conference on e-Business Engineering*. IEEE Computer Society, Washington (2005)
12. Horrocks, I., Patel-Schneider, P.F., van Harmelen, F.: From SHIQ and RDF to OWL: the making of a Web Ontology Language. *Web Semant.: Sci. Serv. Agents World Wide Web* 1(1), 7–26 (2003)
13. Lin, Y.: *Semantic annotation for process models: facilitating process knowledge management via semantic interoperability*. Ph.D. thesis, Norwegian University of Science and Technology (2008)

14. Liu, Y., Müller, S., Xu, K.: A static compliance-checking framework for business process models. *IBM Syst. J.* **46**, 335–361 (2007)
15. Lloyd, J.W.: *Foundations of Logic Programming*. Springer Inc., New York (1987)
16. Motik, B., et al.: *OWL 2 Web Ontology Language Profiles* (Second Edition). W3C Recommendation (2012). <http://www.w3.org/TR/owl2-profiles/>
17. OMG: *Business Process Model and Notation* (2013). <http://www.omg.org/spec/BPMN/2.0.2>
18. Penker, M., Eriksson, H.E.: *Business Modeling With UML: Business Patterns at Work*. Wiley, New York (2000)
19. Polyvyanyy, A.: *Structuring process models* (2012). Ph.D. thesis, University of Potsdam, Germany
20. Polyvyanyy, A., Vanhatalo, J., Völzer, H.: Simplified computation and generalization of the refined process structure tree. In: Bravetti, M., Bultan, T. (eds.) *Proceedings 7th International Workshop Web Services and Formal Methods (WS-FM 2010)*. Lecture Notes in Computer Science, vol. 6551, pp. 25–41. Springer, New York (2011)
21. Polyvyanyy, A., Ouyang, C., Barros, A., van der Aalst, W.M.P.: Process querying: enabling business intelligence through query-based process analytics. *Dec. Support Syst.* **100**, 41–56 (2017)
22. Proietti, M., Smith, F.: Reasoning on data-aware business processes with constraint logic. In: *Proceedings of the 4th International Symposium on Data-driven Process Discovery and Analysis (SIMPDA 2014)*, Milan, Italy, November 19–21, 2014, CEUR Workshop Proceedings, vol. 1293, pp. 60–75 (2014)
23. Prud'hommeaux, E., Seaborne, A.: *SPARQL Query Language for RDF* (2008). W3C Recommendation. <http://www.w3.org/TR/2007/WD-rdf-sparql-query-20070326/>
24. Smith, F., Bianchini, D.: Selection, ranking and composition of semantically enriched business processes. *Comput. Ind.* **65**(9), 1253–1263 (2014)
25. Smith, F., Missikoff, M., Proietti, M.: Ontology-based querying of composite services. In: *Business System Management and Engineering*, pp. 159–180 (2010)
26. Smith, F., Proietti, M.: Rule-based behavioral reasoning on semantic business processes. In: *ICAART* (2), pp. 130–143. SciTePress, Setubal (2013)
27. Smith, F., Proietti, M.: BPAL: a tool for managing semantically enriched conceptual process models. In: Cunningham, P., Cunningham, M. (eds.) *Proceedings of eChallenges (e-2014)*. IIMC International Information Management Corporation, Dublin (2014)
28. Smith, F., Proietti, M.: Ontology-based representation and reasoning on process models: a logic programming approach. *CoRR* **abs/1410.1776** (2014). <http://arxiv.org/abs/1410.1776>
29. Swift, T., Warren, D.: XSB: extending the power of Prolog using tabling. In: *Theory and Practice of Logic Programming (TPLP)*, vol. 12(1–2), pp. 157–187. Cambridge University Press, Cambridge (2012)
30. ter Hofstede, A.M., van der Aalst, W.M.P., Adamns, M., Russell, N. (eds.): *Modern Business Process Automation: YAWL and its Support Environment*. Springer, New York (2010)
31. Thielscher, M.: Introduction to the fluent calculus. *Electron. Trans. Artif. Intell.* **2**, 179–192 (1998)
32. van der Aalst, W.M.P.: The application of Petri nets to workflow management. *J. Circ. Syst. Comput.* **8**(1), 21–66 (1998)
33. Weber, I., Hoffmann, J., Mendling, J.: Beyond soundness: on the verification of semantic business process models. *Distrib. Parallel Databases* **27**, 271–343 (2010)

# CRL and the Design-Time Compliance Management Framework



Amal Elgammal and Oktay Turetken

**Abstract** Following the crisis in 2008, the financial industry has faced growing numbers of laws and regulations globally. The number and complexity of these regulations are creating significant issues for governance, risk, and compliance management in almost all industrial sectors. This emergent *business need* calls for a structured and formal framework for managing business process compliance, which is sustainable throughout the complete business process lifecycle. A *preventive* focus is essential such that compliance is considered from the early stages of business process design, thus enforcing *compliance by design*. This chapter introduces the *Compliance Request Language (CRL)*, which is at the heart of a formal design-time compliance verification, analysis, and management framework and addresses the “Check Compliance” use case. Following a model-driven engineering approach, CRL is a graphical domain-specific language that is formally grounded and enables the abstract pattern-based specification of compliance requirements to alleviate the complexities of formal/mathematical languages. An integrated tool-suite has been developed as an instantiation artifact, and the various validation activities have been conducted to ensure the validity, efficacy, and applicability of the proposed language and framework.

## 1 Introduction

The global regulatory environment has grown in complexity and scope since the financial crisis in 2008. This is causing significant problems for organizations in all industrial sectors, as the complexity of hard and soft regulations is neither under-

---

A. Elgammal (✉)

Faculty of Computers and Information, Cairo University, Giza, Egypt

e-mail: [a.elgammal@fci-cu.edu.eg](mailto:a.elgammal@fci-cu.edu.eg)

O. Turetken

School of Industrial Engineering, Eindhoven University of Technology, Eindhoven, The Netherlands

e-mail: [o.turetken@tue.nl](mailto:o.turetken@tue.nl)

© Springer Nature Switzerland AG 2022

A. Polyvyanyy (ed.), *Process Querying Methods*,

[https://doi.org/10.1007/978-3-030-92875-9\\_10](https://doi.org/10.1007/978-3-030-92875-9_10)

stood nor appreciated [31]. Take, for example, the Dodd–Frank Wall Street Reform and Consumer Protection Act of 2010 that has an estimated 1,500 provisions and 398 rules, which are being drafted by relevant regulatory agencies—approximately 40% of these rules are in force since 2013. The US Bank Secrecy Act Anti-Money Laundering (AML) rules are equally complex and far-reaching, with a raft of major banks found not to be in compliance in 2012. As a consequence, Standard Chartered Bank, London, was fined a total of \$459 million by US regulators in December 2012. Worse still, HSBC Holdings Plc. paid a record \$1.92 billion in fines to US regulators for similar anti-money laundering offences. Similar examples are ample.

In a broader perspective, compliance is about unambiguously ensuring conformance to a set of prescribed and/or agreed upon rules [27]. These rules may originate from various sources, including laws and regulations, standards, public and internal policies, partner agreements, and jurisdictional provisions. To address this emergent *business need*, many organizations typically achieve compliance on a per-case basis resulting into myriad of ad-hoc solutions. In practice, these solutions are generally handcrafted for a particular compliance problem, which creates difficulties for reuse and evolution. Furthermore, compliance and business concepts may be treated differently by different stakeholders. This ambiguity results in inconsistency, which makes it difficult to share and reuse business and compliance specifics. All these problems make it infeasible for automated compliance checking and analysis at any phases of the Business Process Management (BPM) lifecycle, i.e., design-time, runtime, and offline monitoring.

Based on the structured comparative analysis we conducted in [13, 14], a set of requirements have been drawn that must be supported by any compliance management framework. “*Formality*”, “*Expressiveness*”, “*Usability*”, “*Declarativeness*”, “*Semantic alignment*”, and “*Intelligible feedback*” are among the validated requirements of highest priority. These can be described as follows:

- *Formality*: The framework should be formally grounded to pave the way for the application of associated automatic analysis, reasoning, and verification tools and techniques.
- *Expressiveness*: The framework should incorporate a compliance specification language that is expressive enough to be able to capture the intricate semantics of compliance requirements.
- *Usability*: The languages and framework components should not be excessively complex for the prospective users (business or legal users with no formal/mathematical background) so that they would be able to understand and use them.
- *Declarativeness*: Compliance requirements are commonly normative and descriptive, indicating what needs to be done [35]. Therefore, declarative languages are more suited for their formal representation, as opposed to Business Process (BP) models, which are often captured using prescriptive languages/models.
- *Semantic alignment*: The need to manage regulatory and compliance data, especially in heavily regulated domains, exceeds the abilities of modern information systems. Our research indicates that a framework for compliance management should be founded on a semantic knowledge base that incorporates

ontologies capturing the different perspectives of the compliance and business domains [9, 11].

- *Intelligible feedback*: In case of a violation of a compliance rule, it is important to provide the user with guidance of why a violation occurs and how to resolve compliance deviations [12, 15].

In this chapter, we introduce a design-time compliance verification, analysis, and management framework that realizes the aforementioned requirements. In the heart of the framework is the Compliance Request Language (CRL) [11, 16], which adopts a model-driven engineering (MDE) [21] approach and is a graphical pattern-based domain-specific language, capturing and formally representing recurring compliance patterns, primarily in financial domains. CRL capacitates the abstract specification of compliance requirements to alleviate the complexities of formal/mathematical languages to cross the usability gap, which generally represents the main obstacle in adopting powerful and mathematically proven methods for solving various potential problems in various domains.

From a structural perspective, compliance requirements may fall into four classes that pertain to the basic structure of business processes, which are [22]: (i) workflow constraints (control-flow requirements), (ii) information usage (data validation requirements), (iii) employed resources (task allocation and access rights), and (iv) real-time constraints. The compliance patterns incorporated in CRL support these four structural facets of BPs.

CRL and its supporting framework address the “Read” operation of the classical “CRUD” querying operations to support the “Check Compliance” use case as a sub-type of the “Check Conformance” process query use case. The framework introduced in this chapter focuses on design-time BP compliance management; however, CRL can also be used for compliance checking of the subsequent BP phases including runtime and offline monitoring, as proposed in [10, 36]. One of the strengths of an MDE approach is that the same Platform Independent Model (PIM) is used and can be mapped to different Platform Specific Models for diverse objectives. However, to keep the discussion focused, in this chapter, only design-time compliance checking is considered and presented.

The subsequent discussion is organized as follows: CRL design-time compliance verification and analysis framework is discussed in Sect. 2. This is followed by a description of a real-life case study used as a running scenario in this chapter. The framework preliminaries/background is presented in Sect. 4. Then, CRL, its syntax, semantics, and notation are discussed in Sect. 5. Implementation, evaluation and validation efforts are explained in Sects. 6 and 7. Finally, Sect. 8 concludes the chapter.

## 2 CRL Framework

This section presents the CRL compliance verification, analysis, and management framework as an instantiation of the generic BP querying framework presented

in [33]. The proposed framework mainly realizes the newly introduced “Check Compliance” use case as a sub-type of the “Check Conformance” process query use case introduced in [33], by focusing on design-time compliance checking and analysis. Figure 1 shows a schematic view of the proposed framework.

To support different types of queries, three main components should be designed, formalized, and implemented: a data model, where queries/rules are evaluated; an expressive query/specification language; and an efficient evaluation algorithm. These components are realized in our framework as

- *Expressive query/specification language*: CRL represents the query/specification language component, mainly supporting design-time compliance verification and

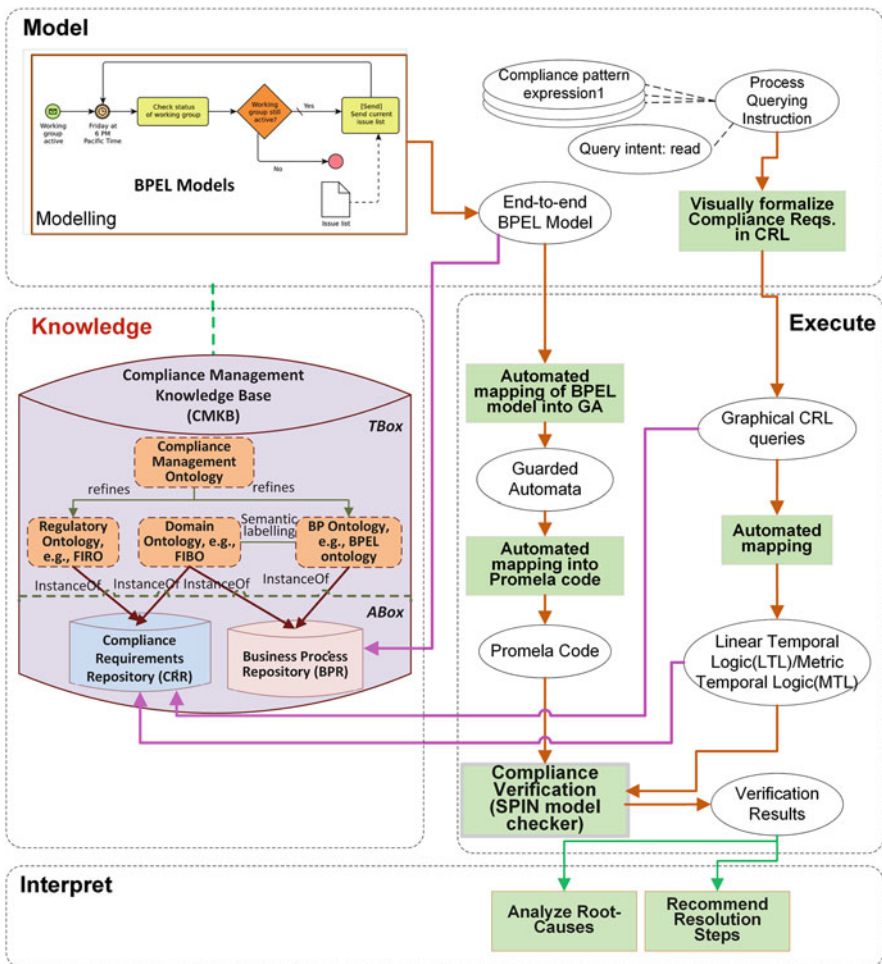


Fig. 1 CRL compliance management framework

analysis. This meets “*Formality*”, “*Usability*”, “*Expressiveness*”, and “*Declarativeness*” requirements discussed in Sect. 1.

- *Data model* represented by the Compliance Management Knowledge Base (CMKB), which meets the “*Semantic Alignment*” requirement discussed in Sect. 1 (see “*Knowledge*” component of the framework proposed in Fig. 1).
- *Evaluation algorithm*: It refers to formal model-checking techniques and addresses the “*Formality*” requirement discussed in Sect. 1.

Following the generic BP querying framework proposed in [33], the CRL compliance management framework constitutes three logical parts (depicted as dashed rounded rectangles in Fig. 1): (i) Model, (ii) Execute, and (iii) Interpret. Furthermore, we introduce an additional fourth part, (iv) Knowledge, as an extension to the original reference framework in [33].

In Fig. 1, rectangles represent actions (active components), while ovals represent objects and aggregations of objects, which are called passive components and they act as inputs/outputs of actions. Solid arrows represent inputs/outputs that flow to/from actions (active components). Dashed lines are used to represent aggregation relationships, e.g., in the figure, a “*Process Querying Instruction*” aggregates one query intent and one or more CRL pattern expressions. A *query intent* is an instruction and CRUD (Create, Read, Update, Delete) operation(s) to elicit requirements and categories of process querying problems [33], e.g., *Design model* use case is associated with the *Create* operation.

## 2.1 “*Model*” Part

The first part is the “*Model*” (the upper part of Fig. 1). It represents the practices of designing (i) process models, including service descriptions (left-hand side of the “*Model*” part) and (ii) process queries that represent compliance requirements in our case (right-hand side of the “*Model*” part). The Business Process (BP) definition involves the specification of process models using the widely used Business Process Execution Language (BPEL) standard [29]. However, other BP modeling languages, such as BPMN [30], can also be treated in a similar way.

Compliance management practices (right-hand side of the “*Model*” part in Figure 1) commence with the refinement of compliance constraints originating from various compliance sources into a set of organization-specific compliance requirements. This involves not only compliance but also business process domain knowledge. The proposed refinement approach is based on the COSO [4] framework, which is the original and primary source used for establishing efficient internal control systems in organizations. For a more detailed discussion on this refinement methodology and its application in two real-life case studies, we refer the interested reader to [37, 38].

The refinement methodology concludes with combining compliance patterns using the CRL to render organization-specific compliance requirements. This



satisfies the *usability* requirement discussed in Sect. 1 and serves as an auxiliary step to represent refined compliance requirements into formal statements (e.g., Linear Temporal Logic (LTL)/Metrical Temporal Logic (MTL) formulas) meeting the “*formality*”, “*declarativeness*”, and “*expressiveness*” requirements.

## 2.2 “*Knowledge*” Part

CMKB represents the backbone of the framework for maintaining and semantically aligning business (BP models and artifacts) and compliance specifics (compliance sources linked to internal controls all the way to corresponding compliance rules following the refinement methodology in [37, 38]) specified in the *Model* part, as illustrated in Sect. 2.1. CMKB is a semantic knowledge base that incorporates and integrates a set of ontologies capturing the different perspectives of the compliance and business spheres. Therefore, the *Knowledge* part represents a uniform conceptualization of the process and compliance spaces, enabling the sharing and reusing of compliance and business knowledge, eliminating ambiguities, and improving the level of automation. The semantic compliance management framework, as reported in [9, 11], is agnostic of the underlying adopted languages/technologies, and identifies the set of minimal ontologies attempting to address various compliance problems.

Typically, the design and development of ontologies constitutes two main parts/components [26, 39]: (i) Terminological component (TBox), which is a conceptualization of a given domain of interest, and (ii) Assertional component (ABox), which represents the individuals or the instances of the TBox. We have identified three main ontologies as part of the TBox:

- *BP Ontology*: An ontology capturing the semantics of the adopted BP language, e.g., BPMNO [17], BPEL ontology [28]. Since we consider BPEL in this chapter, then BPEL ontology [28] is used.
- *Domain Ontology*: An ontology representing the concepts and relationships that exist in the domain of interest, e.g., medical, transport, banking, aerospace, finance, etc. Since the case study presented in the next section addresses the financial domain, we adopted the OMG Financial Industry Business Ontology (FIBO)<sup>1</sup> standard.
- *Regulatory Ontology*: An ontology capturing the requirements, controls, and rules of compliance imperatives. We have developed such an ontology in [9, 11], which we call Financial Industry Regulatory Ontology (FIRO).

The TBox also constitutes a high-level ontology that captures and links important high-level concepts of the business and compliance spheres, i.e., compliance management ontology, which is refined into the BP ontology and Regulatory

---

<sup>1</sup> FIBO: <http://www.omg.org/hot-topics/fibo.htm>.

ontology. The ABox component of the CMKB as shown in Fig. 1 maintains the instances of the semantically aligned compliance rules and BPEL models, which are maintained in the Compliance Requirements Repository (CRR) and Business Process Repository (BPR), respectively, for maintenance and reusability purposes.

### 2.3 “Execute” Part

The flow then proceeds to the *Execute* part of the framework, where the query/compliance evaluation/verification takes place. At one end of the spectrum (compliance requirements side—right-hand side of the *Execute* part), graphical CRL expressions are automatically transformed into LTL formulas. The verification of business process specifications mainly involves checking formal business process (BPEL) specifications (i.e., Promela code) against formal compliance rules (LTL rules) using the SPIN model checker [24]. SPIN is a popular open-source software tool that is intensively used in both academia and industry for the formal verification of large-scale distributed software and hardware systems. SPIN takes as inputs: (i) a Promela (Program Meta Language) code that captures the behavior of a BPEL specification, and (ii) a set of LTL rules capturing relevant compliance requirements, and verifies whether the Promela code complies or violates each LTL rule.

For the automated mapping of BPEL models to Promela code, we utilize WSAT [19] (an open-source tool) and its underlying formal mapping approach [18]. Accordingly, first, a BPEL specification is abstracted into a Guarded Automaton (GA) representing the global sequence of messages exchanged between participating services. GA is a Finite State Automaton augmented with an unbounded queue for incoming messages. Guards can be specified on transitions that are represented as XPath expressions, which enable rich data manipulation and analysis. Next, GA is mapped into the Promela code. Promela is the input language accepted by SPIN model checker. As advocated in [18], having BPEL specifications intermediary represented as GA decouples the BP specification language and formal verification tools from a translator, which enables the loose-coupling and the modular integration of other formal languages, if needed. In addition, it enables the application of other static analysis techniques, e.g., synchronizability and realizability analysis. After the SPIN model checker automatically checks the compliance between the two specifications, the “Verification Results” merely indicate whether a compliance requirement is satisfied or violated, without providing any insights of the possible causes of such violations and guidance on how to resolve them.

### 2.4 “Interpret” Part

The *Interpret* part of the framework analyzes and reasons about root-causes of detected compliance violations. The outcome of SPIN is typically a “yes-no” answer

indicating whether the LTL rule is satisfied or violated. In case of violations, root-causes can be induced by applying the integrated root-cause analysis approach we proposed in [12, 15] based on CRL that utilizes formal Current Reality Trees (CRTs) technique [5]. The root-cause analysis approach also provides the user with suggestive guidelines of how compliance anomalies can be resolved. The business expert(s) can then alter the process specifications (reverting to the *Model* part) taking these guidelines into consideration, which is followed by the automated re-mapping of the BPEL specification into Promela and then the SPIN model checker re-verifies it against the set of applicable formal compliance rules. This forms a closed feedback loop to the *Execute* part, which is terminated when all violated CRs are resolved and a statically compliant business process model is produced.

### 3 Case Study

The case study that is used as a running scenario in this chapter represents a simplified version of an Anti-money laundering (AML) business process. The AML process was conducted as a part of Governance, Risk, and Compliance Technology Center (GRCTC: <http://www.grctc.com/>) Irish project and is presented in [11].

AML is a pressing concern to any organization operating in the financial industry, as it is tightly related to terrorism and proliferation financing. Despite the fact that it is not possible to precisely quantify the amount of money laundered every year, in [34], it has been shown that billions of US dollars are certainly laundered every year. As part of our previous work [9], we built an end-to-end BP model that captures money laundering detection and reporting of the US Patriot Act.<sup>2</sup> The BP model is established based on the best practices and the 40 recommendations of the Financial Action Task Force (FATF).<sup>3</sup>

Figure 2 refers to this process, which proceeds as follows: It starts by a customer initiating a money transfer. Once the order is received by a bank, and if the order amount is greater than 5,000 Euros, an automated check is carried out to detect if the transaction is suspicious. If the automated module detects that the transaction is suspicious, the transaction is marked for manual re-checking by reviewing clearance records and other available records, and contacting the customer for further information, if necessary.

If the transaction is proved to be suspicious, the transaction is flagged as suspicious and then deferred, and a Suspicious Activity Report (MSB) is sent to FinCEN (Financial Crimes Enforcement Network).<sup>4</sup> The customer will be notified

---

<sup>2</sup> <https://gettingthedealthrough.com/area/50/jurisdiction/23/anti-money-laundering-2017-united-states/>.

<sup>3</sup> The 40 Recommendations: <http://www.fatf-gafi.org/publications/fatfrecommendations/documents/the40recommendationspublishedoctober2004.html>.

<sup>4</sup> FinCEN: <http://www.fincen.gov/>.

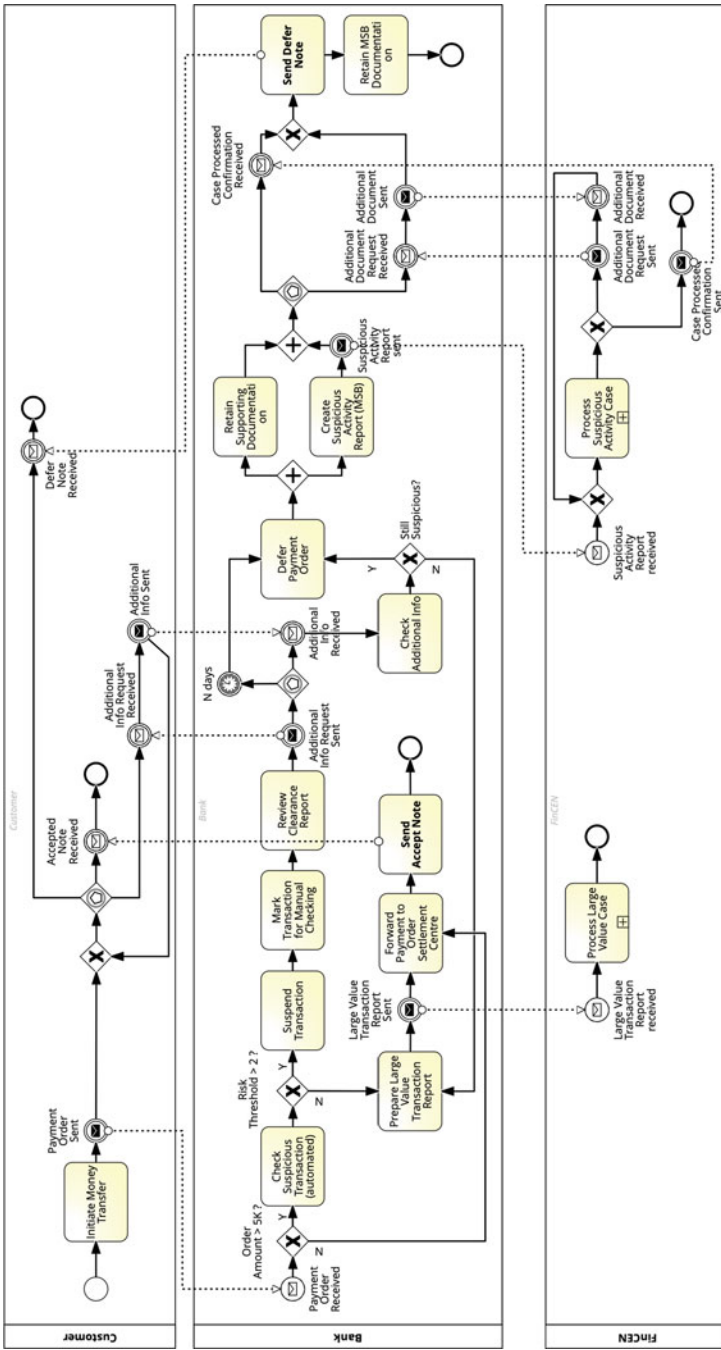


Fig. 2 Money laundering detection and reporting process represented in BPMN v2

**Table 1** Excerpt compliance requirements relevant to the AML case study

ID	Control	Comp. req.	Comp. source
C1	<i>It is obligatory</i> that the financial institution reports any suspicious transaction that involves or aggregates funds of at least \$5,000.	Identity Reporting related provisions	US Patriot Act. §1022. 210 §1022.320
C2	<i>It is necessary</i> that customer identification and verification includes name, date of birth, address, and identification number of a person.		
C3	<i>It is obligatory</i> that the identification of a suspicious transaction is from a manager role reviewing clearance record or other record of money order that is sold and processed, which should be segregated from the activity of suspending a transaction.		
C4	<i>It is obligatory</i> that the customer gets notified by either the acceptance or deference of the money order.	Transparency of the transaction	Internal Policy
C5	<i>It is obligatory</i> that a financial institution maintains each copy of each Suspicious Activity Report-MSB filed for 5 calendar years.	Retention of Record of related provisions	US Patriot Act. §1022.320(c)

in both cases on the status of her transaction, while retaining supporting documents in case they are requested by FinCEN during its investigation.

Table 1 lists a selection of the compliance requirements (CRs) applicable to this case study. These CRs are specified in CRL in Sect. 5. These are verified against the BP model shown in Fig. 2 following the steps of our compliance verification framework (cf. Sect. 2) to deduce whether each of these CRs is satisfied or violated.

## 4 Linear Temporal Logic

Linear Temporal Logic (LTL) [32] is a temporal logic used to formally specify temporal properties of software or hardware designs. In LTL, each state has one possible future and can be represented using linear state sequences, which corresponds to a single execution of the system.

LTL formulas are of the form  $\varphi = Ap$ , where  $A$  is a universal path quantifier and  $p$  is a path formula. A path formula is composed of a finite alphabet of *atomic propositions*  $\mathcal{P}$ . The formation rules of LTL formulas are as follows [32]:

- $\top$  and  $\perp$  are formulas (where  $\top$  represents tautology and  $\perp$  represents contradiction);
- If  $p$  and  $q$  are path formulas, then  $\neg p$ ,  $p \vee q$ ,  $p \wedge q$ ,  $X p$ ,  $F p$ ,  $G p$ ,  $p U q$ ,  $p W q$ , and  $p R q$  are path formulas (where “ $\vee$ ” represents the “or” logical operator and “ $\wedge$ ” represents the “and” logical operator), such that:

- $G p$  (always) holds if formula  $p$  is true in all the states of the path.
- $X p$  (next) holds if formula  $p$  is true in the next state of the path.
- $F p$  (eventually) holds if  $p$  is true at some state in the future.
- $p U q$  (until) holds if the second formula  $q$  is true in some state in the future; then, the first formula  $p$  must be true in all the subsequent states within the path until state  $x$ .
- $p W q$  (weak until) holds the same semantics as  $p U q$ ; however it evaluates to true if  $q$  never occurs.
- $p R q$  (release) holds if the second formula  $q$  is true until and including the point where the first formula  $p$  first becomes true; if  $p$  never becomes true,  $q$  must remain true forever.  $R$  (release) is the dual of  $U$  (until).

### **LTL Semantics:**

- LTL formula  $\varphi$  stands for properties of paths (traces) and a path can either fulfill the LTL formula or not.
- The semantics of  $\varphi$  is defined as a language  $Words(\varphi)$ , where  $Words(\varphi)$  contains all infinite words over the alphabet  $\mathcal{P}$  that satisfy  $\varphi$ .
- The semantics of  $\varphi$  can be extended to interpretations over paths and states of a transition system  $K$  (i.e., Kripke Structure representation of the system under consideration, BP model in our case, as formally defined below).
- A transition system  $K$  satisfies LTL property  $p$  if all  $K$  traces respect  $p$ ; that is, if all behaviors of  $K$  are admissible.
- A state satisfies  $p$  whenever all traces (paths) starting in this state fulfill  $p$ .
- The Kripke structure  $K$  satisfies  $\varphi$  if  $K$  satisfies LTL property  $Words(\varphi)$ .

**Definition 4.1** A Kripke structure is a tuple  $K = \langle S, s_0, \rightarrow, L \rangle$  where:

- $S$  is a set of *states*.
- $s_0 \in S$  is a designated *initial state*.
- $\rightarrow: S \times S$  is a *transition relation*.
- $L: S \rightarrow 2^{\mathcal{P}}$  is a *labeling function*.

**Definition 4.2** A *path* in a Kripke structure  $K$  is an infinite sequence  $\pi := \langle s_0, s_1, s_2, s_3, \dots \rangle$ , such that for all  $i \geq 0$ , we have  $s_i \rightarrow s_{i+1}$ ;  $\pi(i)$  denotes the  $i^{th}$  state on the path and  $\pi_i$  denotes the  $i^{th}$  suffix  $\langle s_i, s_{i+1}, s_{i+2}, \dots \rangle$  of  $\pi$ .

**Definition 4.3** LTL semantics is formally defined as ( $\models$  denotes the satisfaction relationship and  $\not\models$  denotes the dissatisfaction relationship) [32]:

- $K, \pi \models p \Leftrightarrow p \in L(\pi(0))$
- $K, \pi \models \neg\varphi \Leftrightarrow K, \pi \not\models \varphi$
- $K, \pi \models \varphi \wedge \psi \Leftrightarrow K, \pi \models \varphi$  and  $K, \pi \models \psi$
- $K, \pi \models X\varphi \Leftrightarrow K, \pi_1 \models \varphi$
- $K, \pi \models \varphi U \psi \Leftrightarrow$  there exists  $k \in \mathbb{N}$  such that  $K, \pi_k \models \psi$  and  $K, \pi_i \models \varphi$  for all  $0 \leq i < k$
- $K, \pi \models F\varphi \Leftrightarrow K, \pi \models \top U \varphi$
- $K, \pi \models G\varphi \Leftrightarrow K, \pi \models \neg F \neg\varphi$
- $K, \pi \models \varphi R \psi \Leftrightarrow K, \pi \models \neg(\neg\varphi U \neg\psi)$

LTL is used in the discussion of the next section as the formal foundation of compliance patterns pertaining the control-flow, data validation, and task allocation CRs (as discussed in Sect. 1). However, LTL lacks the support to the real-time aspect of BPs, which represents real-time compliance requirements. Various extensions to LTL have been proposed in the literature to overcome this limitation, e.g., Metrical Temporal Logic (MTL) [2] and ForSpec Temporal Logic (FTL) [3]. We have selected MTL as the formal foundation of real-time CRs, mainly due to its successful use in the literature, and it is supported by SPIN model checker (presented in our framework in Sect. 2 and discussed in Sect. 6).

MTL is interpreted over a discrete time domain (over the set of natural numbers  $\mathbb{N}$ ). Since MTL extends LTL, it holds the same semantics (and formation rules) as LTL. In addition, in MTL, temporal operators can be annotated with a real-time expression that represents a specific time interval, e.g.,  $F_{>5} \varphi$ , which means that in some future state after at least a delay of 5 time units, the path formula  $\varphi$  must hold. MTL uses the digital-clock model [2], such that an external, discrete clock progresses at a fixed rate. The granularity of the time can be set.

## 5 Compliance Request Language

In [1], we analyzed a wide range of compliance legislations and frameworks, including Sarbanes-Oxley, Basel III, IFRS, FINRA (NASD/SEC), COSO, COBIT, and OCEG, and examined a variety of relevant works on the specification of associated compliance requirements. Based on this analysis and our joint work with two industrial companies (*PriceWaterHouseCoopers*<sup>TM</sup>, *PwC*, the Netherlands, and *Thales Services*<sup>TM</sup>, France), we have iteratively and incrementally identified structural patterns of frequently recurring compliance requirements imposed on business processes. Based on the findings of the analysis, we have also identified a set of features of a formal language for expressing compliance requirements (for details refer to [13, 14]). Based on the identified features, we have iteratively built CRL following the structured requirements engineering approach that spans over the four structural aspects of compliance requirements discussed in Sect. 1 (control-flow requirements, data validation requirements, resource allocation, and real-time constraints).

In the remaining of this section, we present the syntax, notation, and semantics of CRL (Sect. 5.1), which is followed by a discussion of the four identified classes of CRL patterns, namely, Atomic patterns (Sect. 5.2), Resource patterns (Sect. 5.3), Composite patterns (Sect. 5.4), and Real-time patterns (Sect. 5.5).

### 5.1 Syntax, Notation, and Semantics

Figure 3 presents the meta-model of CRL as a UML class diagram. The *Compliance Pattern* class in the figure is the core element of the language, and each pattern

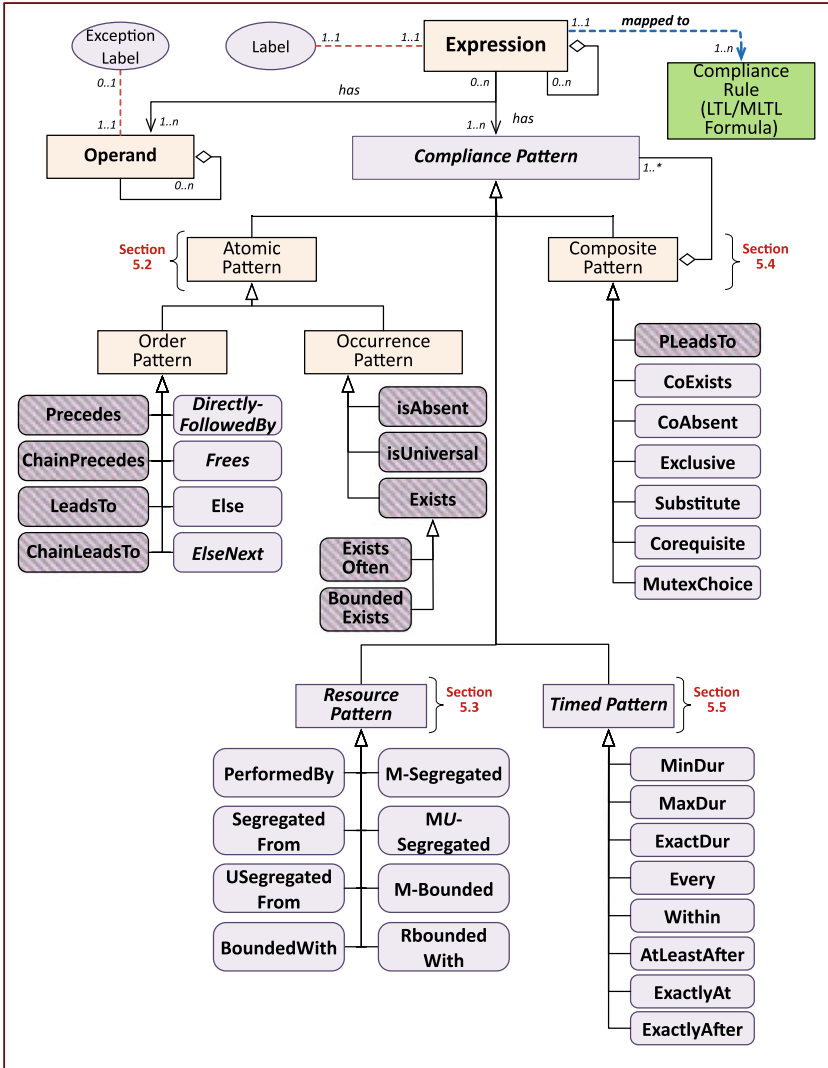


Fig. 3 Compliance request language meta-model

is a sub-type of this class. A *Compliance Pattern* is a template that represents a frequently recurring compliance rule. The *Compliance Pattern* class has four sub-classes: *Atomic Pattern*, *Resource Pattern*, *Composite Pattern*, and *Real-time Pattern* (or *Timed Pattern*).

*Atomic Patterns* deal with occurrence and ordering constraints. Some patterns (those that are line-shaded in Fig. 3) are adopted from Dwyer’s property specification patterns [7]. A *Resource Pattern* captures requirements related to task assignments and authorizations, such as segregation of duties. A *Composite Pattern*



is built from combinations (nesting) of multiple atomic patterns via Boolean operators (not, and, or, xor) to allow for the definition of complex requirements. A *Timed Pattern* allows capturing time-dependent real-time constraints.

As shown in Fig. 3, an *expression* comprises *Compliance Pattern* (*pattern* in short) and one or more *operand(s)*. Expressions can combine multiple (sub-)expressions by using Boolean operators. For example, the expression  $((P \text{ Precedes } Q) \text{ and } (R \text{ Exists}))$  comprises two sub-expressions:  $(P \text{ Precedes } Q)$  is the first expression and  $(R \text{ Exists})$  is the second expression, aggregated by the “and” Boolean operator. *Operands* take the form of BP elements (such as activities, events, business objects, etc.), their attributes, or conditions on them. For example, the unary expression “*CheckSuspiciousTransaction Exists*” consists of one operand, which is *CheckSuspiciousTransaction* BP activity as defined in the AML BP model in Fig. 2, and uses the *Exists* atomic pattern, which mandates that the operand holds at least once within the BP model.

“*Label*” and “*Exception label*” attributes are used to capture and automatically verify non-monotonic compliance rules. “*Label*” is mandatory to identify the rule, while “*Exception label*” is optional. Non-monotonic rules are less strict rules that can be overridden under certain pre-defined conditions such that the rule (identified by *Label*) is still considered as satisfied if it is overridden by one of its pre-defined exceptions (identified by *Exception Label*). For example, the CR that states that “*Checking banking privileges is optional for trusted (gold) customers*” [16] represents a non-monotonic rule, such that the business expert is given the flexibility to conduct this check (*CheckingBankingPrivilege*) or not. This is a necessary requirement to enable relaxations and, thereby, handling exceptional situations. Details about CRL and the framework’s support to these kind of requirements can be found in [8, 16].

An expression built from *Compliance Patterns and Operand(s)* has a direct mapping into an LTL formula. The formal description of CRL grammar defining its syntax in EBNF [20] is presented in [8]. CRL syntax and notation are presented in this section in textual format to enable the conceptualization, sharing, and reuse of knowledge in the CMKB (cf. Sect. 2). However, to address the usability concern of the language, CRL is implemented as a graphical Domain-Specific Language (refer to Sect. 6).

CRL has a formally defined operational semantics given by its mapping to LTL/MTL, as presented in the subsequent sub-sections. Next, we describe *Compliance Pattern* classes in more detail by exemplifying them using AML scenario introduced in Sect. 3. Due to space limitations, only a subset of each class of patterns is discussed. For the complete presentation of patterns, we refer the reader to [8, 16].

## 5.2 Atomic Patterns

*Atomic pattern* class can be used to describe the requirements that involve basic occurrence and ordering of BP elements. They are founded on Dwyer’s property

**Table 2** Mapping rules from atomic patterns to LTL.

CRL expression	Description	LTL representation
$P$ isAbsent	$P$ must not occur	$G(\neg P)$
$P$ Exists	$P$ must occur at least once	$F(P)$
$P$ BoundedExists $\leq 2$ * with bound $\leq 2$	$P$ must occur at most 2 times	$\neg P W (P W (\neg P W (P W \neg F(P))))$
$P$ BoundedExists $\geq 2$ * with bound $\geq 2$	$P$ must occur at least 2 times	$\neg P W (P W (\neg P W (P)))$
$P$ isUniversal	$P$ must always be true	$G(P)$
$P$ Precedes $Q$	$Q$ must always be preceded by $P$	$\neg Q W P$
$P$ Precedes $(S, T)$	A sequence of $S, T$ must be preceded by $P$	$(F(S \wedge XF(T))) \Rightarrow ((\neg S) U P)$
$(S, T)$ Precedes $P$	$P$ must be preceded by a sequence of $S, T$	$F(P) \Rightarrow (\neg P U (S \wedge \neg P \wedge X(\neg P U T)))$
$P$ LeadsTo $Q$	$P$ must always be followed by $Q$	$G(P \Rightarrow F(Q))$
$P$ LeadsTo $(S, T)$	$P$ must be followed by a sequence of $S, T$	$G(P \Rightarrow F(S \wedge XF(T)))$
$(S, T)$ LeadsTo $P$	A sequence of $S, T$ must be followed by $P$	$G(S \wedge XF(T) \Rightarrow X(F(T \wedge F(P))))$
$P$ ExistsOften	$P$ must occur frequently	$GF(P)$
$P$ DirectlyFollowedBy $Q$	$P$ must be directly followed by $Q$	$G(P \Rightarrow X(Q))$
$P$ Frees $Q$	The second operand $Q$ must be true until and including the point where the $P$ first becomes true	$P R Q$

specification patterns [6]. We have extended Dwyer's patterns with four atomic patterns: *Else*, *ElseNext*, *DirectlyFollowedBy*, and *Frees*.

Table 2 presents atomic patterns and their mappings to LTL formulas. "Else" and "ElseNext" atomic patterns are used to represent compensations in a way analogous to If-then-else statements, which are omitted in Table 2 due to space limitations (more details are available [8, 16]).

For example, CRs  $C1$  and  $C2$  in Table 1 can be textually represented in CRL as

$C1$ : (*Order.Amount* > 5000 and *StillSuspiciousTransaction* = "True")  
LeadsTo (*SendSuspiciousActivityReport*)

$C2$ : (*VerifyCustID*) Exists and (*VerifyCustID.name* != null) and  
(*VerifyCustID.DOB* != null) and (*VerifyCustID.address* != null) and  
(*VerifyCustID.ID* != null)

By applying the mapping rules Table 2, the corresponding LTL formulas for  $C1$  and  $C2$  are (where  $R1$  and  $R2$  correspond to  $C1$  and  $C2$ , respectively):

- R1:  $G (((Order.Amount > 5000) \wedge (StillSuspiciousTransaction = "Yes")) \Rightarrow X (SendSuspiciousActivityReport))$
- R2:  $(F (VerifyCustID)) \wedge (VerifyCustID.name \neq "" \wedge VerifyCustID.DOB \neq "" \wedge VerifyCustID.address \neq "" \wedge VerifyCustID.ID \neq "")$

### 5.3 Resource Patterns

Resource patterns involve task allocations, access control, and authorization constraints and constitute one of the important structural facets of the BP compliance. CRL addresses this dimension through the *Resource pattern* class, which involves basic BP concepts, like role, user (actor), and task (BP activity). We assume that tasks are assigned to roles and users perform the tasks through the roles they play. As shown in Fig. 3, we introduce eight resource patterns. A subset of these patterns is described in Table 3, along with their mapping rules to LTL.

For example, to express C3 in Table 1 *PerformedBy*, *SegregatedFrom* resource patterns and *Precedes* atomic pattern can be used and specified as three CRL expressions (note that the three rules can be expressed as one rule using  $\wedge$ ; however, for simplicity, we present them as three simple rules):

- C3.1: *ReviewClearanceRecord PerformedBy Manager*
- C3.2: *SuspendTransaction SegregatedFrom ReviewClearanceRecord*
- C3.3:  $(ReviewClearanceRecord \text{ and } ReviewClearanceRecord.Manual = "Yes") \text{Precedes } (SendSuspiciousActivityReport)$

By applying the mapping rules in Table 3, the corresponding LTL formulas are:

- R3.1:  $G (ReviewClearanceRecord \Rightarrow ReviewClearanceRecord.Role(Manager))$
- R3.2:  $G ((SuspendTransaction.Role(R) \Rightarrow G (\neg (ReviewClearanceRecord.Role(R))))))$

**Table 3** Resource patterns descriptions and their mapping rules to LTL

CRL expression	Description	LTL mapping rule
$t$ <i>PerformedBy</i> $R$	No other role than $R$ is allowed to perform activity $t$	$G (t \Rightarrow t.Role(R))$
$t_1$ <i>SegregatedFrom</i> $t_2$	Activities $t_1$ and $t_2$ must be performed by different roles and users	$G (t_1.Role(R) \Rightarrow \neg(t_2.Role(R)) \wedge G (t_1.User(U) \Rightarrow \neg(t_2.User(U))))$
$t_1$ <i>USegregatedFrom</i> $t_2$	Activities $t_1$ and $t_2$ must be performed by different users	$G (t_1.User(U) \Rightarrow \neg(t_2.User(U)))$
$t_1$ <i>BoundedWith</i> $t_2$	Activities $t_1$ and $t_2$ must be performed by the same user	$G (t_1.User(U) \Rightarrow t_2.User(U)) \wedge G (t_2.User(U) \Rightarrow t_1.User(U))$

*R3.3:  $((\neg \text{SendSuspiciousActivityReport}) \cup (\text{ReviewClearanceRecord} \wedge \text{ReviewClearanceRecord.Manual} = \text{"Yes"})) \vee (G (\text{ReviewClearanceRecord} \wedge \text{ReviewClearanceRecord.Manual} = \text{"Yes"}))$*

During design-time verification (cf. Sect. 6), only the roles that perform certain tasks can be checked, but not the users (actors). This is due to the lack of such contextual information during design-time, which is only available during runtime. As mentioned in Sect. 2, CRL aims to provide an integrated solution addressing both design-time and runtime phases of the BP lifecycle, where CRs formalized as CRL expressions can be mapped into rules appropriate to the target Platform Specific Language (PSM) for the specific verification phase. That is, during design-time, which is in the focus of this chapter, CRL is mapped to LTL as the target language; while during runtime, in [10] we proposed to map CRL into BPath expressions [36] to achieve an integrated runtime compliance verification.

## 5.4 Composite Patterns

To facilitate the definition of complex requirements, composite patterns utilize Boolean logical operators (*not*, *and*, *or*) to enable the nesting of multiple patterns. For example, “*P* *PLeadsTo* *Q*” pattern introduced in [40] is a conjunction of “*P* *Precedes* *Q*” and “*P* *LeadsTo* *Q*”, which indicates that operands *P* and *Q* should “occur” and must take place sequentially. The semantics of “*P* *Precedes* *Q*” alone is that it holds true if *Q* never occurs; i.e., the violation to this rule happens if *Q* occurred and *P* never happened before it. Analogously, “*P* *LeadsTo* *Q*” is true if *P* never happened; i.e., its violation occurs when *P* happened and *Q* did not appear after it.

Table 4 presents a selection of composite patterns together with their mapping rules to LTL. For full description, we refer the reader to [8, 16].

As an example, *C4* in Table 1 can be represented by utilizing the *MutexChoice* composite pattern and *LeadsTo* atomic pattern as follows:

*C4: InitiateMoneyTransfer LeadsTo (SendAcceptNote MutexChoice SendDeferNote)*

This is mapped to LTL by applying the mapping rule in Table 4:

*R4:  $G (\text{InitiateMoneyTransfer} \Rightarrow F ((F (\text{SendAcceptNote}) \wedge G (\neg \text{SendDeferNote})) \vee (F (\text{SendDeferNote}) \wedge G (\neg \text{SendAcceptNote}))))$*

**Table 4** Mapping rules from composite patterns into LTL.

CRL expression	Description	Atomic pattern equivalence	LTL representation
$P \text{ CoExists } Q$	The presence of $P$ mandates that $Q$ is also present	$(P \text{ Exists}) \Rightarrow (Q \text{ Exists})$	$F(P) \Rightarrow F(Q)$
$P \text{ Exclusive } Q$	The presence of $P$ mandates the absence of $Q$ , and presence of $Q$ mandates the absence of $P$	$((P \text{ Exists}) \Rightarrow (Q \text{ IsAbsent})) \wedge ((Q \text{ Exists}) \Rightarrow (P \text{ IsAbsent}))$	$(F(P) \Rightarrow G(\neg Q)) \wedge (F(Q) \Rightarrow G(\neg P))$
$Q \text{ Substitute } P$	$Q$ substitutes the absence of $P$	$(P \text{ IsAbsent}) \Rightarrow (Q \text{ Exists})$	$G(\neg P) \Rightarrow F(Q)$
$P \text{ MutexChoice } Q$	Either $P$ or $Q$ exists but not any of them or both of them	$(P \text{ Exists}) \text{Xor } (Q \text{ Exists}) = ((P \text{ Exists}) \wedge (Q \text{ IsAbsent})) \vee ((Q \text{ Exists}) \wedge (P \text{ IsAbsent}))$	$(F(P) \wedge G(\neg Q)) \vee (F(Q) \wedge G(\neg P))$

## 5.5 Timed Patterns

The real-time dimension is another key aspect in BP compliance and CRL addresses time-related requirements with seven patterns: *MinDur*, *MaxDur*, *Every*, *Within*, *AtLeastAfter*, *ExactlyAt*, *ExactlyAfter*. Timed patterns can be used in combination with other compliance patterns (atomic or composite patterns) to form a “timed composite pattern” expression. However, not every timed pattern can be composed with all compliance patterns. In total, we defined 51 possible combinations, from which a subset is presented in Table 5. For the complete list of combinations, the reader is referred to [8]. Regarding the mapping from timed patterns to corresponding formal statements, MTL is used as presented in Sect. 4.

Similar to *resource patterns*, many of the rules generated using timed patterns can be fully checked only at runtime, as we typically lack the time information at design-time. Therefore, for knowledge encapsulation purposes as discussed in Sect. 2.2 (maintained in CMKB), respective timed compliance rules are specified in CRL and reserved for subsequent runtime compliance monitoring. If the timing information is encoded in the BP model, it could also be checked at design-time. However, BP modeling languages, such as BPEL and BPMN have only limited support to timing specifications through timeouts; i.e., it is not possible to enforce, for example, when a specific activity starts or ends, or how activities are related in real-time to each other (e.g., an activity should start within  $k$  time units after another activity starts or ends).

CR C5 in Table 1 exemplifies timed patterns by exploiting *LeadsTo* atomic pattern and *IsAbsent Before* timed pattern composite expression, as follows:

C5.1: *CreateSuspiciousActivityReport LeadsTo RetainCopyOfMSB*

C5.2: *DeleteSuspiciousTransRecords IsAbsent Before RetainCopyOfMSB.time + 5*

**Table 5** Mapping rules from combinations of compliance/timed patterns into MTL

Timed pattern	Compliance pattern	CRL expression	Description	MTL representation
<i>Before</i>	<i>Exists</i>	$P \text{ Exists Before } k$	Specifies that BP element $P$ must hold at sometime before $k$	$F_{\leq k}(P)$
	<i>IsAbsent</i>	$P \text{ IsAbsent Before } k$	Specifies that BP element $P$ must <i>not</i> hold anytime before $k$	$G_{\leq k}(\neg P)$
<i>Within</i>	<i>LeadsTo</i>	$P \text{ LeadsTo } Q \text{ Within } k$	Indicates that BP element $Q$ has to follow $P$ within $k$ time units after the occurrence of $P$	$G(P \Rightarrow F_{\leq k}(Q))$
	<i>Substitute</i>	$P \text{ Substitutes } Q \text{ Within } k$	$Q$ substitutes the absence of $P$ within at most $k$ time units from the start of the BP	$G(\neg P \Rightarrow F_{\leq k}(Q))$
<i>AtLeastAfter</i>	<i>LeadsTo</i>	$P \text{ LeadsTo } Q \text{ AtLeastAfter } k$	Indicates that BP element $Q$ has to follow $P$ after $k$ time units after the occurrence of $P$	$G(P \Rightarrow F_{\geq k}(Q))$

CRL expression *C5.2* states that no *DeleteSuspiciousTransRecords* activity should take place before the elapse of 5 years after the time of retaining of MSB report (captured by *RetainCopyOfMSB.time*). It is assumed here that the time unit is a year. This can be automatically mapped to MTL as

*R5.1:*  $G(\text{CreateSuspiciousMSB} \Rightarrow F(\text{RetainCopyOfMSB}))$

*R5.2:*  $G_{\leq \text{RetainCopyOfMSB.time}+5}(\neg \text{DeleteSuspiciousTransRecords})$

## 6 Implementation

We have developed a prototypical tool-suite for design-time BP compliance management, implementing the framework/approach described in this chapter for design-time compliance verification and management. Figure 4 presents three main components of the tool-suite and their relationships: *Compliance Rule Manager (CRM)*, *Design-time Compliance Verification Manager (DCVM)*, and *Web Service Analysis Tool (WSAT)*. *Business Process Repository* and *Compliance Repository* in Fig. 4 represent an abstract view of the implementation of the Knowledge component presented in Sect. 2.2 as a set of ontologies using the Web Ontology Language (OWL) [39]. For details on the implementation of the knowledge component, the interested reader is referred to [9, 11].

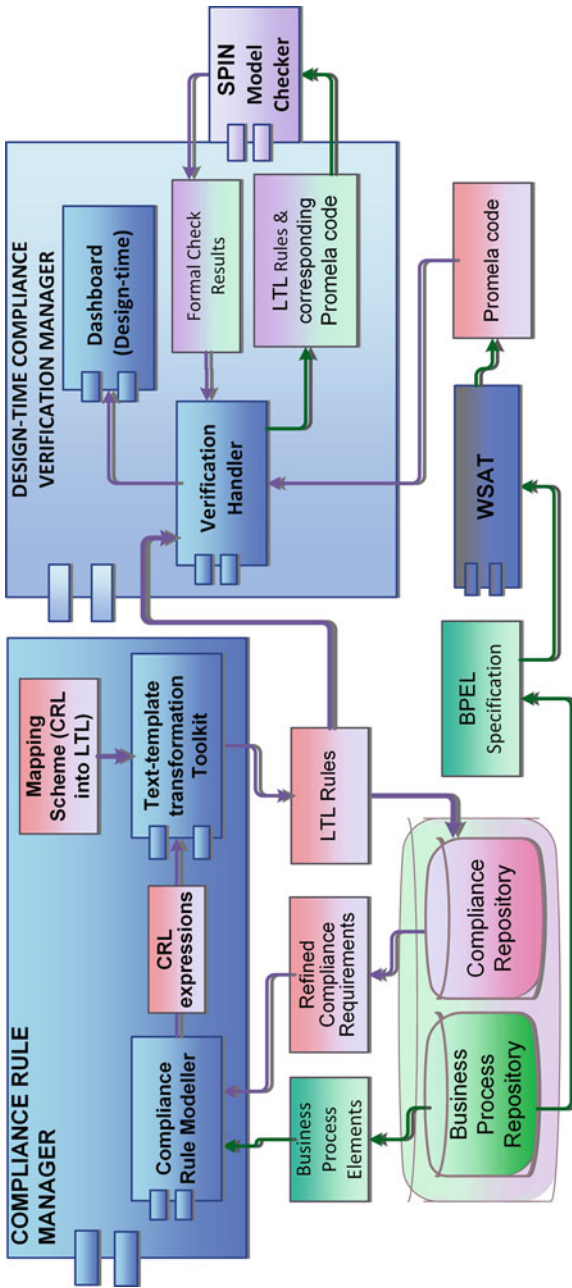


Fig. 4 A high-level architectural view of the interacting components of the tool-suite

**Compliance Rule Manager** CRM represents the CRL graphical editor that also implements the mapping module of CRL expressions into LTL/MTL rules. CRM is a standalone application developed in C# programming language. The upper left hand side of Fig. 4 depicts the internal architecture of CRM, its components, and their interaction with the *Business Process* and *Compliance Repositories*.

CRM comprises two sub-components: *Compliance Rule Modeler* and *Text Template Transformation Toolkit*. Compliance Rule Modeler is a graphical modeler that is used to visually design and create CRL expressions of CRs in a drag-and-drop fashion. Figure 5 shows a screenshot of CRM. The patterns and the operand types are situated on the left side of the GUI. The users drag and drop these constructs on the drawing canvas to build CRL expressions. As described in Sect. 5, *patterns* and *operands* are the main elements that comprise CRL expressions. When an operand type, such as an activity, is selected from the toolbox and dragged onto a swimlane, CRM retrieves the selected type of business process elements available in the Business Process Repository and presents the list to the user for selection (e.g., “Select an Activity” dialog box shown in Fig. 5).

The drawing canvas is divided into swimlanes to enable the specification of multiple rules. For example, the upper swimlane in Fig. 5 (titled *R3*) shows a pattern-based representation that merges *R3.1* and *R3.2* of the running scenario, as described in Sect. 5.3, which uses “*SegregatedFrom*” binary resource pattern that connects two activities (“SuspendTransaction” and “ReviewClearanceRecord”), and “*PerformedBy*” unary resource pattern, represented graphically by the link that connects the “Manger” role to the “ReviewClearanceRecord” activity. The *Text Template Transformation Toolkit* enables the automatic generation of formal compliance rules (as LTL/MTL formulas) from CRL expressions. The output is an XML document that contains compliance rules as LTL/MTL formulas and their properties through the implementation of the LTL/MTL mapping scheme discussed in Sect. 5.

**Design-Time Compliance Verification Manager** Reverting back to Fig. 4, DCVM supports the static (design-time) verification of BPEL specifications against formal LTL rules. First, it retrieves a relevant BPEL model from the *BP Repository* and transforms it into Promela code, as described in Sect. 2 (using the integrated WSAT tool,<sup>5</sup>) so that it can be checked against applicable LTL/MTL rules using the SPIN Model Checker.<sup>6</sup> SPIN implements exhaustive state search as well as multiple optimized evaluation algorithms, e.g., partial-order reduction, hash-compact searches, which helps to solve the typical state explosion problem.

In order to investigate the performance of the tool-suite, we conducted an experiment over the case study presented in Sect. 3. The experiment was conducted on a machine with an Intel Pentium 4 processor 1.7 GHz with 4 GB RAM and

---

<sup>5</sup> WSAT tool: <http://www.cs.ucsb.edu/~su/WSAT>.

<sup>6</sup> SPIN Model Checker: <http://spinroot.com>.



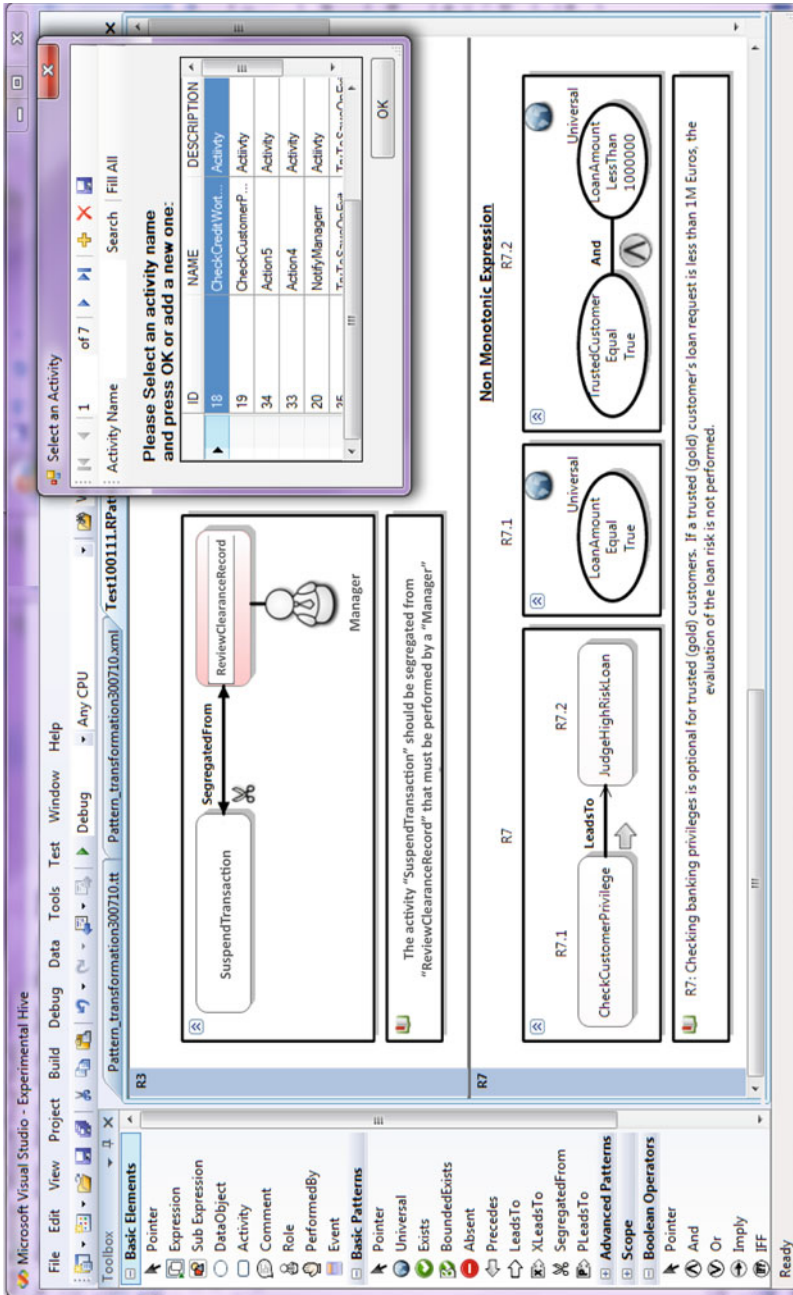


Fig. 5 Screenshot of compliance rule modeler (CRM)

Microsoft Windows 7 operating system installed. The following results are reported: WSAT took 49 seconds on average to transform the BPEL process of the running scenario into its corresponding Promela code. The checking of the generated LTL rules (only a subset is presented in this chapter) consumed between 0.035 to 15 seconds, for each of the rules. The average time of executing all the other rules was 4.6 seconds.

These verification results indicate that the following CRs are satisfied: *C1*, *C4*, and *C5*, and these are violated: *C2*, *C3*. Finally, the results are retrieved from SPIN and reported at the dashboard in a graphical and user-friendly manner.

With regard to the verification results, SPIN indicated that the following LTL rules are satisfied with respect to the AML BP model in Fig. 2: *R1*, *R3.3*, *R4*, *R5.1*; while these CRs are violated: *R2*, *R3.1*, *R3.2*. It was not possible to check the compliance rule *R5.2*, since the exact time when each BP activity takes place is not modeled in BPEL. However, this rule is maintained in the *Compliance Repository* and marked for runtime monitoring.

## 7 Validation and Evaluation

The utility of a design artifact must be rigorously demonstrated via well-executed evaluation methods [23]. Observational methods, such as case studies, allow an in-depth analysis of the artifact and the monitoring of its use in multiple projects within the technical infrastructure of the business environment. In [8, 16], we evaluated and validated the expressiveness of CRL by applying it to the case study presented in Sect. 3. We have also performed two other case studies conducted as part of the EU funded COMPAS project (<http://www.compas-ict.eu>).

The other two case studies were performed in companies operating in different industry sectors and covered processes from the e-business and banking domains. Taking into account the demands for strong regulation compliance schemes, such as Sarbanes-Oxley (SOX), ISO 27000 and sometimes contradictory needs of the different stakeholders, such business environments raise several interesting compliance requirements. The first case study involved an Internet re-seller company that offers products through online systems. The study covered a wide range of BPs, such as order processing, invoicing, cash receipting, and delivery. The second case study covered “loan origination and approval” process that takes place in the banking domain.

As a result of these case studies, we concluded that 72 out of 82 of these requirements could fully take advantage of the proposed language and framework, including their specification, verification, and analysis. The remaining 10 requirements concern mainly the data processing (e.g., rules that are related to the structure and integrity of the data manipulated within the processes) and physical constraints (that demanded locks or guards to protect against unauthorized access to physical assets), which could not be represented in CRL. However, the framework partially supports these requirements in capturing and encoding of knowledge, which is of

great value to business organizations as validated by COMPAS industrial partners (PwC, the Netherlands and Thales Services, France). We refer the reader to [8, 16] for detailed information about the case studies, experiments and findings, and the further validation of CRL by means of functional testing and application of the guidelines developed by Hevner et al. [23].

## 8 Discussion and Conclusion

Compliance Request Language is a specification language geared to capture compliance requirements for their analysis and verification. It spans the four structural aspects of business processes: control-flow, data validation, employed resources, and real-time aspects. CRL meets the requirements/features that we have identified as important for the language that aims to support business process compliance management, based on a comparative analysis conducted in [13, 14].

CRL is an *open* and *extensible* language, which means that based on the considered problem domain, new patterns/pattern classes can be introduced in a plug-and-play fashion, and the mapping scheme from CRL to other target/PSM languages can also be defined. This can enable the specification of certain requirements that are not expressible in LTL/MTL (and consequently in the current version of CRL).

In regard to the *expressive* power of CRL, it is *limited* to the underlying target language; i.e., the expressive power of LTL/MTL in our case, and also the set of defined patterns (that we have defined based on our extensive analysis of regulatory frameworks, laws and regulations, and standards, as well as several cases [16]). As demonstrated in our evaluation, CRL is *not* complete, i.e., it cannot represent all compliance requirements that business processes are subjected to; instead, it targets frequently recurring compliance requirements. In designing CRL, we targeted the relevance and usability of CRL in practice. It is well-recognized that there is usually a trade-off between these attributes and the expressiveness of the language/notation [25], and finding an appropriate balance between these two conflicting aspects (expressiveness and usability) is of utmost importance. Therefore, we designed CRL as an extensible language. By analyzing diverse application domains, such as healthcare, agriculture, energy, and manufacturing, new pattern classes and compliance patterns can be identified and incorporated into CRL.

Identification of new pattern classes and domain-specific patterns forms one of the central themes in our future research efforts. In our ongoing work, we target at the financial and healthcare domains, with the aim to identify new domain-specific compliance patterns, and to apply the existing ones to provide further evidence for their applicability and expressive power. Our future work will also focus on improvements in our tool-suite for increasing its efficiency and usability in order to facilitate its use in practice.

## References

1. Compas project, deliverable 2.1, state-of-the-art in the field of compliance languages (2008)
2. Alur, R., Henzinger, T.A.: Real-time logics: complexity and expressiveness. *Inf. Comput.* **104**(1), 35–77 (1993). <https://doi.org/10.1006/inco.1993.1025>
3. Armoni, R., Fix, L., Flaisher, A., Gerth, R., Ginsburg, B., Kanza, T., Landver, A., Mador-Haim, S., Singerman, E., Tiemeyer, A., Vardi, M.Y., Zbar, Y.: The forspec temporal logic: a new temporal property-specification language. In: Katoen, J., Stevens, P. (eds.) *Tools and Algorithms for the Construction and Analysis of Systems*, 8th International Conference, TACAS 2002, Held as Part of the Joint European Conference on Theory and Practice of Software, ETAPS 2002, Grenoble, April 8–12, 2002, Proceedings. *Lecture Notes in Computer Science*, vol. 2280, pp. 296–211. Springer, New York (2002). [https://doi.org/10.1007/3-540-46002-0\\_21](https://doi.org/10.1007/3-540-46002-0_21)
4. COSO: Internal control – integrated framework. the committee of sponsoring organizations of the treadway commission (1994)
5. Dettmer, H.: *Goldratt's theory of constraints: A systems approach to continuous improvement* (1997)
6. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Property specification patterns for finite-state verification. In: Ardis, M.A., Atlee, J.M. (eds.) *Proceedings of the Second Workshop on Formal Methods in Software Practice*, March 4–5, 1998, Clearwater Beach, FL, pp. 7–15. ACM, New York (1998). <https://doi.org/10.1145/298595.298598>
7. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Patterns in property specifications for finite-state verification. In: Boehm, B.W., Garlan, D., Kramer, J. (eds.) *Proceedings of the 1999 International Conference on Software Engineering, ICSE'99*, Los Angeles, CA, May 16–22, 1999, pp. 411–420. ACM Press, New York (1999). <https://doi.org/10.1145/302405.302672>
8. Elgammal, A.: *Towards a comprehensive framework for business process compliance*. Ph.D. thesis, Information Management Department, Tilburg University (2012)
9. Elgammal, A., Butler, T.: Towards a framework for semantically-enabled compliance management in financial services. In: Toumani, F., Pernici, B., Grigori, D., Benslimane, D., Mendling, J., Hadj-Alouane, N.B., Blake, M.B., Perrin, O., Saleh, I., Bhiri, S. (eds.) *Service-Oriented Computing - ICSOC 2014 Workshops - WESOA; SeMaPS, RMSOC, KASA, ISC, FORMOVES, CCSA and Satellite Events*, Paris, France, November 3–6, 2014, Revised Selected Papers. *Lecture Notes in Computer Science*, vol. 8954, pp. 171–184. Springer, New York (2014). [https://doi.org/10.1007/978-3-319-22885-3\\_15](https://doi.org/10.1007/978-3-319-22885-3_15)
10. Elgammal, A., Sebah, S., Turetken, O., Hacid, M.S., Papazoglou, M., van den Heuvel, W.: *Business process compliance management : an integrated proactive approach* (2014)
11. Elgammal, A., Turetken, O.: Lifecycle business process compliance management: a semantically-enabled framework. In: *2015 International Conference on Cloud Computing (ICCC)*. IEEE, New York (2015). <https://doi.org/10.1109/cloudcomp.2015.7149646>
12. Elgammal, A., Turetken, O., van den Heuvel, W.: Using patterns for the analysis and resolution of compliance violations. *Int. J. Cooperative Inf. Syst.* **21**(1), 31–54 (2012). <https://doi.org/10.1142/S0218843012400023>
13. Elgammal, A., Turetken, O., Heuvel, W., Papazoglou, M.: On the formal specification of business contracts and regulatory compliance. In: *BMC Health Services Research* (2010)
14. Elgammal, A., Turetken, O., van den Heuvel, W., Papazoglou, M.P.: On the formal specification of regulatory compliance: A comparative analysis. In: Maximilien, E.M., Rossi, G., Yuan, S., Ludwig, H., Fantinato, M. (eds.) *Service-Oriented Computing - ICSOC 2010 International Workshops, PAASC, WESOA, SEE, and SOC-LOG*, San Francisco, CA, December 7–10, 2010, Revised Selected Papers. *Lecture Notes in Computer Science*, vol. 6568, pp. 27–38 (2010). [https://doi.org/10.1007/978-3-642-19394-1\\_4](https://doi.org/10.1007/978-3-642-19394-1_4)
15. Elgammal, A., Turetken, O., van den Heuvel, W., Papazoglou, M.P.: Root-cause analysis of design-time compliance violations on the basis of property patterns. In: Maglio, P.P., Weske, M., Yang, J., Fantinato, M. (eds.) *Service-Oriented Computing - 8th International*

- Conference, ICSOC 2010, San Francisco, CA, December 7–10, 2010. Proceedings. Lecture Notes in Computer Science, vol. 6470, pp. 17–31 (2010). [https://doi.org/10.1007/978-3-642-17358-5\\_2](https://doi.org/10.1007/978-3-642-17358-5_2)
16. Elgammal, A., Turetken, O., van den Heuvel, W., Papazoglou, M.P.: Formalizing and applying compliance patterns for business process compliance. *Software Syst. Model.* **15**(1), 119–146 (2016). <https://doi.org/10.1007/s10270-014-0395-3>
  17. Francescomarino, C.D., Ghidini, C., Rospocher, M., Serafini, L., Tonella, P.: Reasoning on semantically annotated processes. In: Bouguettaya, A., Krüger, I., Margaria, T. (eds.) *Service-Oriented Computing - ICSOC 2008*, 6th International Conference, Sydney, December 1–5, 2008. Proceedings. Lecture Notes in Computer Science, vol. 5364, pp. 132–146 (2008). [https://doi.org/10.1007/978-3-540-89652-4\\_13](https://doi.org/10.1007/978-3-540-89652-4_13)
  18. Fu, X., Bultan, T., Su, J.: Analysis of interacting BPEL Web services. In: Feldman, S.I., Uretsky, M., Najork, M., Wills, C.E. (eds.) *Proceedings of the 13th International Conference on World Wide Web, WWW 2004*, New York, NY, May 17–20, 2004, pp. 621–630. ACM, New York (2004). <https://doi.org/10.1145/988672.988756>
  19. Fu, X., Bultan, T., Su, J.: WSAT: A tool for formal analysis of Web services. In: Alur, R., Peled, D.A. (eds.) *Computer Aided Verification*, 16th International Conference, CAV 2004, Boston, MA, July 13–17, 2004, Proceedings. Lecture Notes in Computer Science, vol. 3114, pp. 510–514. Springer, New York (2004). [https://doi.org/10.1007/978-3-540-27813-9\\_48](https://doi.org/10.1007/978-3-540-27813-9_48)
  20. Garshol, L.M.: BNF and EBNF: What are they and how do they work? (2008). <http://www.garshol.priv.no/download/text/bnf.html>. Accessed on 8 Nov 2020
  21. Gašević, D., Djuric, D., Devedžić, V.: Model driven engineering. In: *Model Driven Engineering and Ontology Development*, pp. 125–155. Springer, Berlin, Heidelberg (2009). [https://doi.org/10.1007/978-3-642-00282-3\\_4](https://doi.org/10.1007/978-3-642-00282-3_4)
  22. Goedertier, S., Vanthienen, J.: Designing compliant business processes with obligations and permissions. In: Eder, J., Dustdar, S. (eds.) *Business Process Management Workshops, BPM 2006 International Workshops, BPD, BPI, ENEI, GPWW, DPM, semantics4ws*, Vienna, September 4–7, 2006, Proceedings. Lecture Notes in Computer Science, vol. 4103, pp. 5–14. Springer, New York (2006). [https://doi.org/10.1007/11837862\\_2](https://doi.org/10.1007/11837862_2)
  23. Hevner, A.R., March, S.T., Park, J., Ram, S.: Design science in information systems research. *MIS Quart.* **28**(1), 75–105 (2004). <http://misq.org/design-science-in-information-systems-research.html>
  24. Holzmann, G.J.: The model checker SPIN. *IEEE Trans. Software Eng.* **23**(5), 279–295 (1997). <https://doi.org/10.1109/32.588521>
  25. Karsai, G., Krahn, H., Pinkernell, C., Rumpe, B., Schindler, M., Völkel, S.: Design guidelines for domain specific languages. CoRR abs/1409.2378 (2014). <http://arxiv.org/abs/1409.2378>
  26. Krötzsch, M., Simancik, F., Horrocks, I.: A description logic primer. CoRR abs/1201.4089 (2012). <http://arxiv.org/abs/1201.4089>
  27. Ly, L.T., Rinderle-Ma, S., Göser, K., Dadam, P.: On enabling integrated process compliance with semantic constraints in process management systems - requirements, challenges, solutions. *Inf. Syst. Front.* **14**(2), 195–219 (2012). <https://doi.org/10.1007/s10796-009-9185-9>
  28. Nitzsche, J., Wutke, D., van Lessen, T.: An ontology for executable business processes. In: Hepp, M., Hinkelmann, K., Karagiannis, D., Klein, R., Stojanovic, N. (eds.) *Proceedings of the Workshop on Semantic Business Process and Product Lifecycle Management SBPM 2007*, held in conjunction with the 3rd European Semantic Web Conference (ESWC 2007), Innsbruck, June 7, 2007, *CEUR Workshop Proceedings*, vol. 251. CEUR-WS.org (2007). <http://ceur-ws.org/Vol-251/paper8.pdf>
  29. OASIS: Business process execution language (bpel) (2007)
  30. OMG: Business Process Model and Notation (BPMN), Version 2.0 (2011)
  31. Papazoglou, M.P., van den Heuvel, W.: Business process development life cycle methodology. *Commun. ACM* **50**(10), 79–85 (2007). <https://doi.org/10.1145/1290958.1290966>
  32. Pnueli, A.: The temporal logic of programs. In: *18th Annual Symposium on Foundations of Computer Science*, Providence, Rhode Island, 31 October–1 November 1977, pp. 46–57. IEEE Computer Society (1977). <https://doi.org/10.1109/SFCS.1977.32>

33. Polyvyanyy, A., Ouyang, C., Barros, A., van der Aalst, W.M.P.: Process querying: enabling business intelligence through query-based process analytics. *Dec. Support Syst.* **100**, 41–56 (2017). <https://doi.org/10.1016/j.dss.2017.04.011>
34. Reuter, P.: *Chasing Dirty Money: The Fight Against Money Laundering. Recording for the Blind & Dyslexic* (2005)
35. Sadiq, S.W., Governatori, G., Namiri, K.: Modeling control objectives for business process compliance. In: Alonso, G., Dadam, P., Rosemann, M. (eds.) *Business Process Management, 5th International Conference, BPM 2007, Brisbane, September 24–28, 2007, Proceedings. Lecture Notes in Computer Science*, vol. 4714, pp. 149–164. Springer, New York (2007). [https://doi.org/10.1007/978-3-540-75183-0\\_12](https://doi.org/10.1007/978-3-540-75183-0_12)
36. Sebahi, S., Hacid, M.: Business process monitoring with BPath. In: Meersman, R., Dillon, T.S., Herrero, P. (eds.) *On the Move to Meaningful Internet Systems: OTM 2010 - Confederated International Conferences: CoopIS, IS, DOA and ODBASE, Hersonissos, Crete, Greece, October 25–29, 2010, Proceedings, Part I. Lecture Notes in Computer Science*, vol. 6426, pp. 446–453. Springer, New York (2010). [https://doi.org/10.1007/978-3-642-16934-2\\_33](https://doi.org/10.1007/978-3-642-16934-2_33)
37. Türetken, O., Elgammal, A., van den Heuvel, W., Papazoglou, M.P.: Enforcing compliance on business processes through the use of patterns. In: Tuunainen, V.K., Rossi, M., Nandhakumar, J. (eds.) *19th European Conference on Information Systems, ECIS 2011, Helsinki, June 9–11, 2011*, p. 5 (2011). <http://aisel.aisnet.org/ecis2011/5>
38. Türetken, O., Elgammal, A., van den Heuvel, W., Papazoglou, M.P.: Capturing compliance requirements: A pattern-based approach. *IEEE Software* **29**(3), 28–36 (2012). <https://doi.org/10.1109/MS.2012.45>
39. W. W. W. C. (W3C): *OWL Web ontology language overview* (2011)
40. Yu, J., Manh, T.P., Han, J., Jin, Y., Han, Y., Wang, J.: Pattern based property specification and verification for service composition. In: Aberer, K., Peng, Z., Rundensteiner, E.A., Zhang, Y., Li, X. (eds.) *Web Information Systems - WISE 2006, 7th International Conference on Web Information Systems Engineering, Wuhan, October 23–26, 2006, Proceedings. Lecture Notes in Computer Science*, vol. 4255, pp. 156–168. Springer, New York (2006). [https://doi.org/10.1007/11912873\\_18](https://doi.org/10.1007/11912873_18)

# Process Query Language



Artem Polyvyanyy

**Abstract** A *process* is a collection of actions that were already, are currently being, or must be taken in order to achieve a goal, where an *action* is an atomic unit of work, for instance, a business activity or an instruction of a computer program. A *process repository* is an organized collection of models that describe processes, for example, a business process repository and a software repository. Process repositories without facilities for *process querying* and *process manipulation* are like databases without Structured Query Language, that is, collections of elements without effective means for deriving value from them. Process Query Language (PQL) is a domain-specific programming language for managing processes described in models stored in process repositories. PQL can be used to query and manipulate process models based on possibly infinite collections of processes that they represent, including processes that support concurrent execution of actions. This chapter presents PQL, its current features, publicly available implementation, planned design and implementation activities, and open research problems associated with the design of the language.

## 1 Introduction

Computing revolutionizes many aspects of our lives by innovating how data is collected and processed. The innovations often stem from the ability to design, manage, and automatically learn semantically rich artifacts from the data, for example, using machine learning, statistical analysis, and data and process mining techniques. Such semantically rich artifacts reflect different types of patterns present in the data, calling for dedicated methods for querying and manipulating them to allow systematic derivation of value. One such type of patterns concerns temporal aspects of the data, capturing how work is carried out in processes.

---

A. Polyvyanyy (✉)

School of Computing and Information Systems, Faculty of Engineering and Information Technology, The University of Melbourne, Parkville, VIC, Australia

e-mail: [artem.polyvyanyy@unimelb.edu.au](mailto:artem.polyvyanyy@unimelb.edu.au)

© Springer Nature Switzerland AG 2022

A. Polyvyanyy (ed.), *Process Querying Methods*,

[https://doi.org/10.1007/978-3-030-92875-9\\_11](https://doi.org/10.1007/978-3-030-92875-9_11)

A *process* is a collection of actions that, when executed, lead to the accomplishment of a goal. An *action* is an atomic unit of work. For example, an action can represent a business activity or a computer program instruction. Execution of an action in a process leads to a change in the state of the process. A process can contain already executed actions, actions that are currently being executed, and actions that yet are awaiting their execution. A process solely composed of already executed actions represents a historical process that was observed in the real-world. In turn, a process comprising only designed but not executed actions is an envisioned process that may be observed in the future. A *process model* is a model that describes a collection of processes that encode different ways to accomplish the same goal. Note that a process model often describes an infinite collection of processes to address the need to iterate certain actions an initially unknown number of times to achieve the desired process state. Finally, a *process repository* is an organized collection of process models. For example, models can be organized into folders to impose their logical grouping. Examples of process repositories include business process repositories and software repositories.

Process repositories without *process querying* and *process manipulation* capabilities are of low practical utility, as their manual processing is often infeasible. Process Query Language (PQL) is a domain-specific programming language for querying and manipulating process models based on the processes these models describe. It is a declarative language with SQL-like syntax. PQL programs are also called *queries*.

To support process querying, PQL implements two classes of predicates. The first class comprises the *4C behavioral predicates*, a collection of constraints that systematically explore the fundamental behavioral relations of co-occurrence, conflict, causality, and concurrency in processes [19, 25]. These predicates, for instance, can be used to retrieve models that describe processes in which a given action always occurs or in which a given pair of actions can be executed concurrently. The second class is composed of *process scenarios*, sequences of actions with wildcards [22]. Despite being declarative, process scenarios allow checking whether a model describes processes that contain requested sequences of actions. Hence, process scenarios can be used to retrieve models that describe processes that obey the requested imperative constraints.

PQL supports statements for process manipulation. Concretely, one can use PQL to specify and execute instructions for manipulating models to insert, delete, and update processes in the collections of processes these models describe. The process insertion capabilities of PQL are implemented as a solution to the *process repair* problem [17, 22]. The delete and update process manipulations are not implemented in the current version of PQL. Still, they are demonstrated here for the completeness of the discussion of the intended scope for the language.

The next section presents several motivating examples of PQL programs for querying and manipulating processes. Section 3 gives an overview of the features currently supported by PQL. To facilitate the comparison of PQL with another process querying methods, Sect. 4 positions PQL within the Process Querying Framework [21]. Then, Sect. 5 discusses our open-source implementation of a



process repository that supports PQL. Section 6 surveys open research problems triggered by the design of PQL and lists planned efforts that aim to shape the language. Finally, Sect. 7 closes the chapter with conclusions.

## 2 Motivating Examples

In this section, we present several motivating examples of PQL programs for querying and manipulating process models. To this end, we use an example process repository composed of six process models shown in Fig. 1. The models are captured in Business Process Model and Notation (BPMN). In BPMN, rectangles with rounded corners denote actions. Gateways are visualized as diamonds. Exclusive gateways use the “×” marker inside the diamond shape, whereas parallel gateways use the “+” marker. Directed arcs encode control flow dependencies. For simplicity, the models in the example repository use abstract action labels; see labels A through G in the figure. In general, an action label specifies the meaning of the action, for example, “assess claim” or “archive case”. Models can be further supplied with attributes, for instance, unique identifier, version, creation date, and author. Models can be grouped into collections in a repository by putting them into folders, which, similar to folders of a file system, can form a folder hierarchy.

Models in a repository can be queried using PQL `SELECT` statements. For example, PQL queries Q1 and Q2 listed below implement process querying using the 4C predicates, while PQL queries Q3 and Q4 use process scenarios.

```

Q1.  SELECT * FROM *
      WHERE AlwaysOccurs("C") AND
      Cooccur("B", "C");
Q2.  SELECT "Author", "Version" FROM "/examples"
      WHERE (CanOccur("G") AND
      (NOT Conflict("E", "G"))) OR
      (TotalConcurrent("C", {"B", "D"}, ANY) AND
      AlwaysOccurs("C"));

```

Query Q1 requests to retrieve every model and all its attributes (see “`SELECT *`”) from every folder of the repository (“`FROM *`”) that describes (“`WHERE`”) a collection of processes in which every process contains at least one occurrence of action C (“`AlwaysOccurs("C")`”) and actions B and C cooccur in the processes (“`Cooccur("B", "C")`”), that is, B cannot occur without C in a process, C cannot occur without B in a process, and there exists at least one process in the collection in which both actions B and C appear. Model 1 in Fig. 1 matches query Q1 and, thus, should be retrieved if Q1 is executed over the repository. Indeed, model 1 describes four processes:  $\langle A, B, C, D, E, F \rangle$ ,  $\langle A, C, B, D, E, F \rangle$ ,  $\langle A, B, C, D, B \rangle$ , and  $\langle A, C, B, D, B \rangle$ ; we map BPMN models to Petri nets to interpret them as collections of processes [4]. Note that action C occurs in every process, while actions B and C cooccur in the processes of the model. Models 5 and 6 also match query Q1. It is easy to verify that both actions B and C occur in all processes

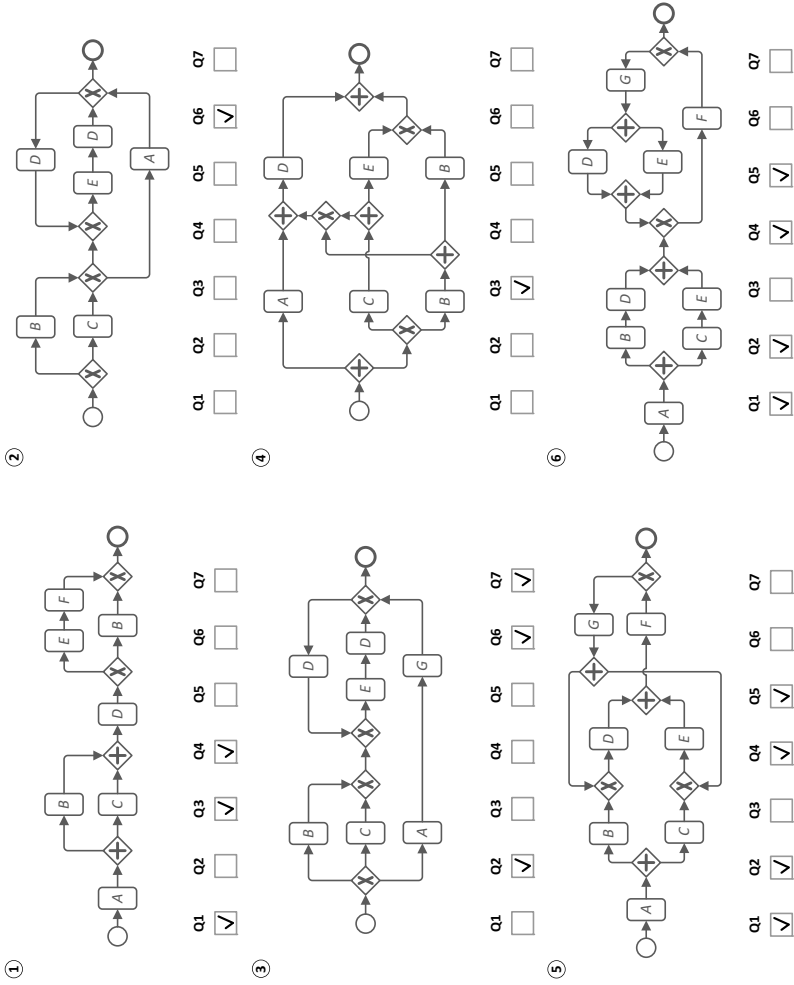


Fig. 1 An example process repository

these two models describe, as every process starts with one of these two prefixes:  $\langle A, B, C \rangle$  or  $\langle A, C, B \rangle$ . To denote which models match which queries, in Fig. 1, under each model, we mark corresponding checkboxes. Hence, models 2, 3, and 4 do not match query Q1. For instance, the process  $\langle B, A \rangle$  described by model 2 confirms that neither C always occurs nor B and C cooccur in the processes of model 2.

Query Q2 requests to retrieve process models and their attributes Author and Version (“SELECT "Author", "Version"”) located in the “/examples” folder of the repository (“FROM "/examples"”) that satisfy at least one of the two following conditions. First, the model should describe at least one process in which action G occurs at least once (“CanOccur ("G"”) and actions E and G do not conflict (“NOT Conflict ("E", "G"”)”, where actions E and G conflict if the model describes at least one process in which E occurs but G does not occur, at least one process in which G occurs but E does not occur, and the model does not describe a process in which both E and G occur. Second, in every process of the model, action C occurs (“AlwaysOccurs ("C"”)”, and all occurrences of action C are either concurrent with all occurrences of B or with all occurrences of D (“TotalConcurrent ("C", {"B", "D"}, ANY)”). In general, two actions A and B are in the total concurrent relation if in every process in which both A and B occur, every occurrence of action A is concurrent with every occurrence of action B; refer to Sect. 3 for details.

Assuming that all models in Fig. 1 are stored in the “/examples” folder, models 3, 5, and 6 match query Q2. In model 3, action G can occur, consider, for example, the process  $\langle A, G \rangle$  of the model, and actions E and G do not conflict, as evidenced, for instance, by the process  $\langle A, G, D, E, D \rangle$  of the model. In models 5 and 6, in turn, action C always occurs and actions B and C are in the total concurrent relation. Fig. 2 shows three, out of infinitely many, concurrent processes described by model 5. In these three processes, actions B and C occur once and are concurrent; there is no directed path between these actions; for details, again, see Sect. 3. The same phenomenon can be observed for all the other processes of model 5. Note that the only occurrence of action C is concurrent with the only occurrence of action D in process 1. However, in processes 2 and 3, there are occurrences of action D that are not concurrent with the occurrence of action C. These occurrences are highlighted with gray background in the figure.

PQL query Q3 below requests to retrieve all process models in the repository that support a process that commences with zero or more actions before action B occurs, then eventually action D occurs in the process, followed eventually by another occurrence of action B, and then the process completes via zero or more occurrence of any other actions. Models 1 and 4 from the repository in Fig. 1 match query Q3. This fact is evidenced by processes  $\langle A, B, C, D, B \rangle$  and  $\langle A, B, D, B \rangle$  described by models 1 and 4, respectively. Query Q4, also shown below, requests to retrieve models that describe the process  $\langle A, B, C, D, E, F \rangle$  and does not describe processes with two consecutive occurrences of action B. Models that match this query are models 1, 5, and 6.

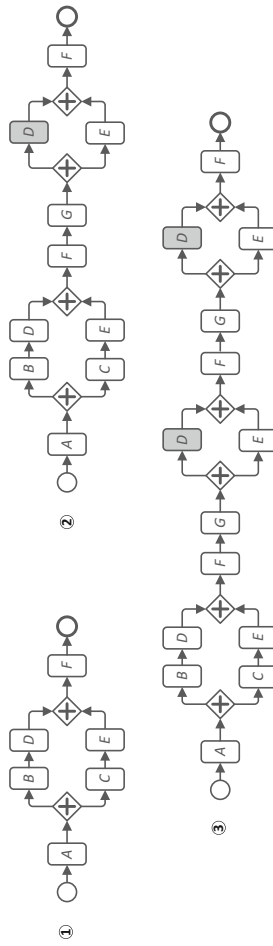


Fig. 2 Three concurrent processes of model 5 from Fig. 1

- Q3. **SELECT \* FROM \***  
**WHERE** Executes (<\*, "B", \*, "D", \*, "B", \*>);
- Q4. **SELECT \* FROM \***  
**WHERE** Executes (<"A", "B", "C", "D", "E", "F">) **AND**  
**NOT** Executes (<\*, "B", "B", \*>);

The attentive reader has noticed that models 5 and 6 from the repository describe the same processes. Thus, these two models, besides being structurally different, are behaviorally equivalent. The result of a PQL query depends on the processes the models describe and is independent of the particular way the models are structured. Consequently, models 5 and 6 either both match or both do not match a given PQL query; refer to the checkboxes next to these two models in Fig. 1.

PQL queries Q5–Q7 below capture instructions for manipulating process models.

- Q5. **INSERT** <\*, "F", "D", "G", \*> **INTO** \*  
**WHERE** Executes (<\*, "F", "G", \*>);
- Q6. **DELETE** <"A", "G"> **FROM** \*  
**WHERE** GetTasksAlwaysOccurs (GetTasks ())  
**EQUALS** {};
- Q7. **UPDATE** <"A", "G", \*>  
**SET** <"A", "F", \*>  
**FOR** \*;

Query Q5 ensures that each model from every folder of the repository (“**INTO** \*”) that describes a process in which an occurrence of action G immediately follows an occurrence of action F (“**WHERE** Executes (<\*, "F", "G", \*>)”) also describes a process in which an occurrence of F is immediately followed by an occurrence of D that, in turn, is immediately followed by an occurrence of G (“**INSERT** <\*, "F", "D", "G", \*>”). If a model that describes the former process also describes the latter requested process, the model is not manipulated. Otherwise, the model is manipulated to obtain an extended version of the model that also describes the requested latter process. Models 5 and 6 in the repository describe processes in which F is immediately followed by G and, hence, must be manipulated. Model 7 in Fig. 3 is a model that can be created based on model 5 as a result of executing PQL query Q5. Note that model 7 describes the requested process. Note also that the requested manipulation can be implemented in several ways, which raises the question of the quality of the resulting model. This aspect is a subject of ongoing research and is discussed in Sect. 6.

Query Q6 captures a request to manipulate every process model in the repository (“**FROM** \*”) that does not contain an action that occurs in each of its processes (“**WHERE** GetTasksAlwaysOccurs (GetTasks ()) **EQUALS** {}”) and describes the process that starts with an occurrence of action A and then immediately completes with an occurrence of action G so that the resulting model does not describe that process (“**DELETE** <"A", "G">”). Models 2 and 3 from the repository match the condition in the **WHERE** clause. However, only model 3 describes process ⟨A, G⟩, and, thus, should be manipulated. The resulting, manipulated by PQL, model is added as a fresh model to the repository. Similar as for the **INSERT** statement, several valid resulting models can be considered. For example, models 8 and 9 in Fig. 3 can be accepted as models that result from

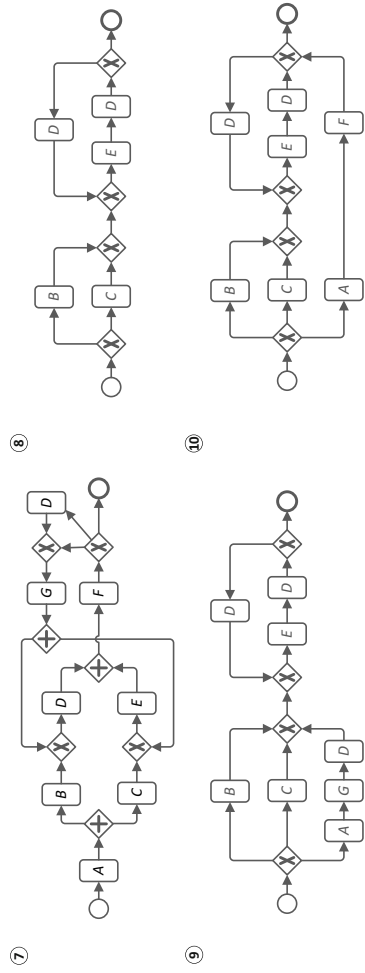


Fig. 3 Manipulated process models

executing query Q6 over model 3. While model 8 does not describe all the processes with prefix  $\langle A, G \rangle$  described by model 3, including the requested process  $\langle A, G \rangle$ , the processes described by model 9 differ from those described by model 3 by exactly one process  $\langle A, G \rangle$ . Note that the implementation of the DELETE statement can vary between versions of PQL.

Finally, query Q7 requests to update all models in the repository (“FOR \*”) by updating processes that start with the prefix  $\langle A, G \rangle$  (“UPDATE <"A", "G", \*>”) to start with the prefix  $\langle A, F \rangle$  (“SET <"A", "F", \*>”). Again, multiple implementations of the UPDATE statement can be envisaged, and model 10 in Fig. 3 is a possible result of executing query Q7 over model 3, which is also the only model in Fig. 1 that must be manipulated according to query Q7.

### 3 Process Query Language

This section reviews the core features of PQL. First, Sect. 3.1 discusses the main primitives of PQL for querying process models. Then, Sect. 3.2 presents the currently implemented PQL mechanisms for manipulating process models.

#### 3.1 Process Querying

For the purpose of process querying, PQL interprets a process model as a collection of *concurrent processes*. A *concurrent process* is a collection of actions such that for some pairs of actions in the collection, it is specified that one of the actions *causally precedes* the other in the executions of the process. The control flow arcs and the transitive dependencies that these arcs induce in Fig. 2 define the causal precedence relations of the corresponding concurrent processes. In concurrent process 1 in Fig. 2, for example, action A causally precedes action E, which, in turn, causally precedes action F. In contrast, for the pairs of actions that are not in the causal precedence relation, it is accepted that they are independent, or *concurrent*, and, thus, can be performed simultaneously in the executions of the process. For instance, actions B and E are concurrent in process 1 in Fig. 2. As already explained in Sect. 2, a process model can describe infinitely many concurrent processes.

Every concurrent process describes a collection of (sequential) processes. These are processes that do not violate the causal precedence constraints of the concurrent process. For example, concurrent process 2 in Fig. 2 describes twelve sequential processes, induced by all the interleavings of the concurrent actions; these twelve processes include, for instance, processes  $\langle A, B, C, E, D, F, G, D, E, F \rangle$  and  $\langle A, C, B, E, D, F, G, E, D, F \rangle$ . Every concurrent process describes a finite collection of sequential processes. But, needless to say, a model that describes infinitely many concurrent processes also describes infinitely many sequential processes.

To perform process querying using PQL, the user can specify a query that requests to retrieve models that fulfill a condition verified over all the processes

**Table 1** Occurrence predicates; the predicates are evaluated in the context of a process model

Predicate	Definition
CanOccur (A)	The predicate evaluates to <i>true</i> if the model describes <i>at least one</i> process with <i>at least one</i> occurrence of action A; otherwise, it evaluates to <i>false</i> .
AlwaysOccurs (A)	The predicate evaluates to <i>true</i> if <i>every</i> process the model describes has <i>at least one</i> occurrence of action A; otherwise, it evaluates to <i>false</i> .

of the models. One way to specify a condition is by using behavioral predicates, as detailed in Sect. 3.1.1, or scenarios, as discussed in Sect. 3.1.2.

### 3.1.1 Behavioral Predicates

Process models describe processes composed of actions that can be executed and, thus, observed in the real-world. One way to convey how many occurrences of an action, or pairs of actions in a specific behavioral relationship, can be observed in the executions of processes described by the model is by using predicates with quantifiers.<sup>1</sup> When studying process models, the user may, for instance, be interested in how often certain actions can occur, how often certain actions can cause occurrences of other actions, or how often actions can be executed simultaneously.

The 4C spectrum is a systematically organized repertoire of predicates that assess in how many processes that a model describes how many occurrences of one action are in a specific behavioral relation with how many occurrences of another action [25]. The predicates of the spectrum explore the fundamental *behavioral relations* of co-occurrence, conflict, causality, and concurrency of action occurrences in processes. Hence, we refer to these predicates as *behavioral predicates*.

A PQL query can use predicates of the 4C spectrum as atomic propositions in the propositional logic formula of its WHERE clause. When a model is matched to a query, the value of each predicate is established based on the processes that the model describes. If the formula in the WHERE clause of a SELECT statement evaluates to *true* for a particular model, then the model is included in the result of the query. Additional checks may need to be applied for other PQL statement types to confirm that the model indeed must be manipulated.

To accompany the 4C predicates, all of which are binary predicates, that is, they take two actions as input, PQL supports two unary predicates listed in Table 1. As suggested by their definitions, these predicates allow verifying the frequencies of individual action occurrences, for example, before applying the 4C predicates, which then can explain how these occurrences relate to each other.

<sup>1</sup> A *predicate* is a function that evaluates to either *true* or *false* truth value, while a *quantifier* is an operator that specifies how many elements from the given collection should satisfy an open formula.



**Table 2** Co-occurrence and conflict predicates; the predicates are evaluated in the context of a process model

Predicate	Definition
CanConflict (A, B)	The predicate evaluates to <i>true</i> if the model describes <i>at least one</i> process with <i>at least one</i> occurrence of action A and <i>no</i> occurrences of action B; otherwise, it evaluates to <i>false</i> .
CanCooccur (A, B)	The predicate evaluates to <i>true</i> if the model describes <i>at least one</i> process with <i>at least one</i> occurrence of action A and <i>at least one</i> occurrence of action B; otherwise, it evaluates to <i>false</i> .
Conflict (A, B)	The predicate evaluates to <i>true</i> if the model describes <i>no</i> process with <i>at least one</i> occurrence of action A and <i>at least one</i> occurrence of action B; otherwise, it evaluates to <i>false</i> .
Cooccur (A, B)	The predicate evaluates to <i>true</i> if <i>every</i> process the model describes that has <i>at least one</i> occurrence of action A also has <i>at least one</i> occurrence of action B, and <i>vice versa</i> ; otherwise, it evaluates to <i>false</i> .
Requires (A, B)	The predicate evaluates to <i>true</i> if the model describes <i>no</i> process with <i>at least one</i> occurrence of action A and <i>no</i> occurrences of action B, <i>at least one</i> process with <i>at least one</i> occurrence of action B and <i>no</i> occurrences of action A, and <i>at least one</i> process with <i>at least one</i> occurrence of action A and <i>at least one</i> occurrence of action B; otherwise, it evaluates to <i>false</i> .
Independent (A, B)	The predicate evaluates to <i>true</i> if the model describes <i>at least one</i> process with <i>at least one</i> occurrence of action A and <i>no</i> occurrences of action B, <i>at least one</i> process with <i>at least one</i> occurrence of action B and <i>no</i> occurrences of action A, and <i>at least one</i> process with <i>at least one</i> occurrence of action A and <i>at least one</i> occurrence of action B; otherwise, it evaluates to <i>false</i> .

Table 2 lists six 4C predicates grounded in the conflict and co-occurrence behavioral relations. Note that the CanConflict and CanCooccur predicates are seminal as the remaining four predicates from the table can be expressed as propositional logic formulas over them. Hence, these four predicates can be seen as macros that can simplify the conditions the user may want to express in the WHERE clause of a PQL query. The CanConflict and CanCooccur predicates can be combined into logic formulas to express other conditions that explore conflict and co-occurrence behavioral relations. According to one classification, 63 conflict and 15 co-occurrence properties can be expressed this way [25].

Table 3 lists all the 4C predicates grounded in the causal precedence and concurrency behavioral relations. Given actions A and B, the predicates emerge through universal or existential quantification over three domains, namely the collection of all concurrent processes that the model describes (see column “Pr.” in the table), the collection of all occurrences of action A in a concurrent process the model describes (column “A”), and the collection of all occurrences of action B in the same concurrent process, and the choice of the behavioral relation between the occurrences of actions A and B (column “Rel.”), either causal precedence (“Causal.”) or concurrency (“Concur.”). These configurations lead to eight causality and eight

**Table 3** Concurrency and causality predicates; “Pr.”—concurrent processes of the model, “A” —occurrences of action A in the concurrent process, “B” — occurrences of action B in the concurrent process, “V” —every concurrent process of the model/every occurrence of the action in the concurrent process, “E” —exists a concurrent process of the model/exists an occurrence of the action in the concurrent process, “Rel.”—behavioral relation, “Syntax”—PQL syntax for expressing the predicate, and “Name”—the name of the predicate. The predicates are evaluated in the context of a process model

Pr.	A	B	Rel.	Syntax	Name
V	V	V	Causal.	TotalCausal(A, B)	Total (mutual) causal
V	V	V	Concur.	TotalConcurrent(A, B)	Total (mutual) concurrent
V	V	E	Causal.	TotalFunctionalCausal(A, B)	Total functional causal
V	V	E	Concur.	TotalFunctionalConcurrent(A, B)	Total functional concurrent
V	E	V	Causal.	TotalDominantCausal(A, B)	Total dominant causal
V	E	V	Concur.	TotalDominantConcurrent(A, B)	Total dominant concurrent
V	E	E	Causal.	TotalExistCausal(A, B)	Total existential causal
V	E	E	Concur.	TotalExistConcurrent(A, B)	Total existential concurrent
E	V	V	Causal.	ExistTotalCausal(A, B)	Existential total causal
E	V	V	Concur.	ExistTotalConcurrent(A, B)	Existential total concurrent
E	V	E	Causal.	ExistFunctionalCausal(A, B)	Existential functional causal
E	V	E	Concur.	ExistFunctionalConcurrent(A, B)	Existential functional concurrent
E	E	V	Causal.	ExistDominantCausal(A, B)	Existential dominant causal
E	E	V	Concur.	ExistDominantConcurrent(A, B)	Existential dominant concurrent
E	E	E	Causal.	ExistCausal(A, B)	Existential (mutual) causal
E	E	E	Concur.	ExistConcurrent(A, B)	Existential (mutual) concurrent

concurrency predicates. The syntax of the behavioral predicates in PQL and their names are provided in columns “Syntax” and “Name” of Table 3, respectively.

For example, the total concurrent predicate evaluates to *true* for input actions A and B, if in every (“ $\forall$ ”) concurrent process the model describes that has at least one occurrence of action A and at least one occurrence of action B, it holds that every (“ $\forall$ ”) occurrence of action A is concurrent (“Concur.”) with every (“ $\forall$ ”) occurrence of action B; otherwise, the total concurrent predicate evaluates to *false* for that input. Thus,  $\text{TotalConcurrent}(B, C)$  evaluates to *true* for model 5 in Fig. 1. Indeed, every concurrent process of model 5 contains exactly one occurrence of action B, exactly one occurrence of action C, and these occurrences are concurrent; see three out of infinitely many concurrent processes model 5 describes in Fig. 2. In contrast,  $\text{TotalConcurrent}(D, E)$  evaluates to *false* for model 5 and processes 2 and 3 in Fig. 2 evidence this, as they contain occurrences of D and E that are in the causal precedence relation. However, process 1 in Fig. 2 justifies the fact that  $\text{ExistTotalConcurrent}(D, E)$  holds *true*. This predicate verifies whether there exists a concurrent process described by the model in which all occurrences of actions are concurrent. In process 1, there is exactly one occurrence of action D, exactly one occurrence of action E, and these two occurrences are concurrent. Note, however, that “stronger” concurrency relations also hold between actions D and E in model 5, for instance,  $\text{TotalFunctionalConcur}(D, E)$  and  $\text{TotalFunctionalConcur}(E, D)$ . Indeed, in every (“ $\forall$ ”) concurrent process of model 5, for every (“ $\forall$ ”) occurrence of action D in the process, there exists (“ $\exists$ ”) an occurrence of action E that is concurrent with that occurrence of D, and vice versa.

As examples of the causality predicates, note that  $\text{TotalCausal}(B, D)$  holds, but  $\text{TotalCausal}(C, D)$  does not hold for model 5 from Fig. 1. In every concurrent process in Fig. 2 it holds that the only occurrence of action C is concurrent to one occurrence of action D, invalidating the total causal relation between the actions. In contrast, the only occurrence of action B is in the causal precedence relation with every occurrence of action D in every concurrent process of model 5.

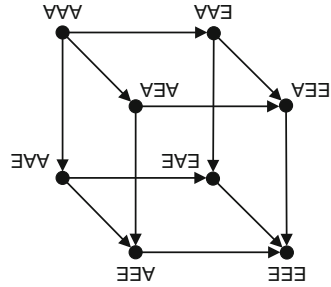
Table 4 lists definitions of all the eight 4C causality predicates. The definitions of the eight concurrency predicates can be obtained by replacing the causal precedence relations with the concurrency relations. Furthermore, Polyvyanyy et al. [25] formalize all the predicates using mathematical notation.

As already mentioned, causality and concurrency predicates can be distinguished based on their “strength.” Fig. 4 summarizes implications between the pairs of causality (or concurrency) predicates from the 4C spectrum; the transitive implications are not shown. The vertices represent causality (or concurrency) predicates, while the labels encode the quantifiers from the first three columns in Table 3. Hence, for example, the fact that the  $\text{ExistTotalCausal}$  predicate holds for a given pair of actions (see the “ $\exists\forall\forall$ ” label in Fig. 4) implies that both  $\text{ExistFunctionalCausal}$  (“ $\exists\forall\exists$ ”) and  $\text{ExistDominantCausal}$  (“ $\exists\exists\forall$ ”) predicates hold and, transitively,  $\text{ExistCausal}$  (“ $\exists\exists\exists$ ”) holds for the same pair of actions; note that the converse implications, in general, do not hold. Consequently,

**Table 4** Causality predicates. The predicates are evaluated in the context of a process model

Predicate	Definition
ExistCausal(A, B)	The predicate evaluates to <i>true</i> if the model describes <i>at least one</i> concurrent process in which <i>at least one</i> occurrence of action A causally precedes <i>at least one</i> occurrence of action B; otherwise, it evaluates to <i>false</i> .
ExistDominantCausal(A, B)	The predicate evaluates to <i>true</i> if the model describes <i>at least one</i> concurrent process with <i>at least one</i> occurrence of action B and, in that concurrent process, there is <i>one</i> occurrence of action A that causally precedes <i>every</i> occurrence of action B; otherwise, it evaluates to <i>false</i> .
ExistFunctionalCausal(A, B)	The predicate evaluates to <i>true</i> if the model describes <i>at least one</i> concurrent process with <i>at least one</i> occurrence of action A and, in that concurrent process, there is <i>one</i> occurrence of action B such that <i>every</i> occurrence of action A causally precedes that occurrence of action B; otherwise, it evaluates to <i>false</i> .
ExistTotalCausal(A, B)	The predicate evaluates to <i>true</i> if the model describes <i>at least one</i> concurrent process with <i>at least one</i> occurrence of action A, <i>at least one</i> occurrence of action B, and, in that concurrent process, <i>every</i> occurrence of action A causally precedes <i>every</i> occurrence of action B; otherwise, it evaluates to <i>false</i> .
TotalExistCausal(A, B)	The predicate evaluates to <i>true</i> if in <i>every</i> concurrent process the model describes that has <i>at least one</i> occurrence of action A and <i>at least one</i> occurrence of action B, <i>at least one</i> occurrence of action A causally precedes <i>at least one</i> occurrence of action B; otherwise, it evaluates to <i>false</i> .
TotalDominantCausal(A, B)	The predicate evaluates to <i>true</i> if in <i>every</i> concurrent process the model describes that has <i>at least one</i> occurrence of action A and <i>at least one</i> occurrence of action B, there is <i>one</i> occurrence of action A that causally precedes <i>every</i> occurrence of action B; otherwise, it evaluates to <i>false</i> .
TotalFunctionalCausal(A, B)	The predicate evaluates to <i>true</i> if in <i>every</i> concurrent process the model describes that has <i>at least one</i> occurrence of action A and <i>at least one</i> occurrence of action B, there is <i>one</i> occurrence of action B such that <i>every</i> occurrence of action A causally precedes that occurrence of action B; otherwise, it evaluates to <i>false</i> .
TotalCausal(A, B)	The predicate evaluates to <i>true</i> if in <i>every</i> concurrent process the model describes that has <i>at least one</i> occurrence of action A and <i>at least one</i> occurrence of action B, <i>every</i> occurrence of action A causally precedes <i>every</i> occurrence of action B; otherwise, it evaluates to <i>false</i> .

**Fig. 4** The 4C spectrum causality/concurrency lattice [25]



we say that `ExistTotalCausal` is stronger than the other existential causality predicates.

In a study with the prospective stakeholders of PQL, all twelve preselected 4C predicates were recognized as suitable for process querying. The `CanOccur`, `AlwaysOccurs`, `Cooccur`, `Conflict`, `TotalCausal`, and `TotalConcurrent` predicates were, in addition, identified as most useful and such that are most likely to be used for solving practical problems [19].

### 3.1.2 Scenarios

Any finite repertoire of behavioral predicates is limited in its expressive power, as it can only express a finite number of conditions over a fixed collection of actions, while the number of process collections that process models can express over the same actions is unbounded [18]. Therefore, in addition to querying based on the 4C predicates, PQL supports scenario-based querying [22].

The concept central to scenario-based querying is the notion of a *trace with wildcards*. A trace with wildcards is a finite sequence in which every element is either a special wildcard element “\*” or a pair composed of an action and a number between zero and one. For example,  $\omega = \langle *, (A, 1.0), (B, 0.8), *, (A, 1.0) \rangle$  is a trace with wildcards composed of five elements.

A trace with wildcards defines a collection of processes. These processes result from the concatenation of collections of sequences defined by the elements of the trace. The concatenation is performed in the order the corresponding elements appear in the trace. The special “\*” element defines the collection of all finite sequences over all possible actions. In turn, an element that is a pair of an action  $x$  and a number  $y$  defines the collection of all sequences composed of one action, where the actions are taken from the set of all actions that are similar with  $x$  to the level of at least  $y$ ; the similarity should be established based on some given similarity function that maps pairs of actions to their similarity scores between zero and one. Different similarity functions can be used. For instance, one such similarity function can be established based on the similarity of action names or labels. Thus,  $\omega$  defines the collection that includes every process in which action A eventually occurs, that occurrence is immediately followed by an occurrence of action B, or an

occurrence of some similar with B action, and then some other actions can occur before the process ends with yet another occurrence of action A.

The `Executes` predicate takes as input a trace with wildcards and verifies, in the context of a given process model, whether the model describes at least one process that is also included in the collection of processes defined by the trace. In other words, it verifies whether the model can execute actions according to the pattern captured by the trace. If so, the predicate returns *true*; otherwise, it returns *false*. The concrete syntax of the `Executes` predicate for the input trace with wildcards  $\omega$  is `Executes (<*, "A", "B" [0.8] , *, "A">)`, or `Executes (<*, "A", ~"B" , *, "A">)` if the process querying tool is configured to use 0.8 as the default action similarity threshold.

The `Executes` predicates can be used, together with the 4C predicates, as atomic propositions in the propositional logic formula of the `WHERE` clause of a PQL query, thus enriching the expressive power of the language. Indeed, by combining `Executes` predicates, one can, for instance, express a condition to check whether a given model describes, or does not describe, some finite collection of processes of interest. Note that, in general, the number of such conditions is unbounded. For more information on the scenario-based querying support in PQL, refer to [22].

### 3.2 Process Manipulation

Process manipulations in PQL are implemented using the concept of an *optimal alignment* between a process and a process model [1, 31]. An alignment is composed of moves. A *synchronous move* is a pair in which both elements are the same action, for example (A, A). In contrast, an *asynchronous move* is a pair in which one element is an action, and the other element is a special “no move” element, denoted by “ $\gg$ ”. An *alignment* is a sequence of synchronous and asynchronous moves for which two conditions hold. First, the first elements from the moves, when positioned in the order the corresponding moves appear in the alignment and all the “no move” elements are skipped, form the process. Second, the second elements from the moves, again positioned as in the alignment and without the “no move” elements, form a process described by the model. Finally, an optimal alignment between a process and model is an alignment between the process and model such that every other alignment between them has more asynchronous moves than an optimal alignment.

An alignment is often summarized as a table. For instance, Table 5 shows an optimal alignment between process  $\langle F, D, G \rangle$  and process model 5 from Fig. 1. It is a sequence of thirteen moves. In the table, moves are encoded as columns, such that two successive columns refer to two successive moves in the alignment. Each column has two rows. The top row of each column specifies the first element in the corresponding move, while the bottom row specifies the second element in the

**Table 5** An optimal alignment between process  $\langle F, D, G \rangle$  and process model 5 from Fig. 1

»	»	»	»	»	F	D	G	»	»	»
A	C	B	E	D	F	»	G	D	E	F

move. Hence, the optimal alignment in Table 5 consists of two synchronous and eleven asynchronous moves.

For instance, PQL relies on the alignment from Table 5 to implement query Q5 discussed in Sect. 2 on model 5 from Fig. 1. Indeed, the alignment demonstrates that the process fragment  $\langle F, D, G \rangle$  requested to be inserted into the model, see “INSERT  $\langle *, "F", "D", "G", * \rangle$ ” in the query, has a “gap” captured by the asynchronous move (D, ») in the processes described by the model, see the move highlighted with gray background in the alignment. This asynchronous move determines the place in the model at which action D can be inserted; after process-prefix  $\langle A, C, B, E, D, F \rangle$  and before process-suffix  $\langle G, D, E, F \rangle$ . The concrete modifications on the model are then implemented using *process repair* techniques [7, 17] from the field of process mining [30]. Recall that model 7 in Fig. 3 is a model that results from executing query Q5 on model 5.

## 4 Process Querying Framework

The *Process Querying Framework* (PQF) is an abstract system of components that can be selectively replaced to result in a new process querying method [21]. In this section, we identify which active and passive components of the framework are supported in PQL. The aim of this exercise is threefold: Tracking the status of the PQL implementation, planning the next design and implementation activities, and preparation of PQL for comparison with other process querying methods positioned within the framework.

Figure 5 shows a schematic view of the framework. In the figure, rectangles and ovals denote active and passive components, respectively. The arcs denote input and output passive components of active components. That is, the passive components are consumed and produced by the active components. Dashed lines encode the aggregation relationships between the passive components. Finally, we use different backgrounds to reflect the different implementation statuses of the components; refer to the legend in the figure. The framework consists of four parts, each responsible for one dedicated function, including managing processes and queries, preparing and executing queries, and supporting the interpretation of querying results. In Fig. 5, each part is enclosed in an area with a dotted border.

The “Model, Simulate, Record, and Correlate” part of the framework is responsible for the management of the process repository and process queries. In general, the repository can comprise different types of models of processes. PQL was initially introduced to address querying of process models, that is, conceptual models that

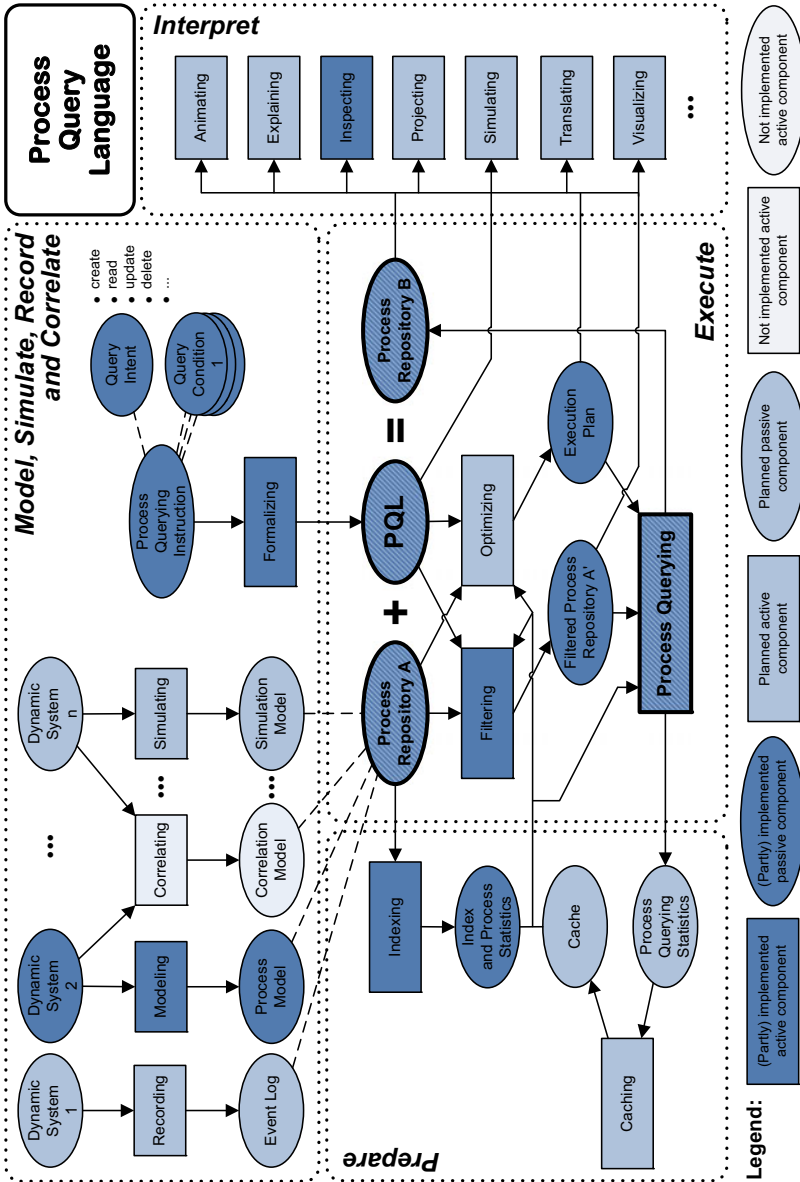


Fig. 5 A schematic view of the components of the Process Querying Framework implemented in PQL; adapted from [21]



describe collections of processes. Examples of process models are Petri nets, BPMN diagrams, Event-driven Process Chains (EPCs), and UML Activity Diagrams. The current implementation of PQL works with process models formalized as Petri nets. Note that for many process modeling notations, the corresponding mappings to Petri nets have been devised. Being able to query process models, PQL can be adapted for querying their recorded executions, also known as *event logs* in process mining [30], and, consequently, to *simulation models*, as combinations of models and their executions. The extension of PQL to support querying over event logs and simulation models is future work. Other models that describe processes, for instance, correlation models that specify relationships between multiple processes, are not currently supported by PQL. A process querying instruction specifies an intent to query or manipulate a process repository utilizing various query conditions.

PQL is a language for formalizing process querying instructions. It supports process querying by means of the *read* intent implemented using `SELECT` statements. In the current version of PQL, process manipulation is implemented using `INSERT` statements that address the *create* and *update* process querying intents. In the future, the support of the *update* intent will be supplemented by `UPDATE` and `DELETE` PQL statements.

The “Prepare” part of the framework, as its name suggests, is responsible for preparing the process repository for efficient querying. The framework offers two types of preparations: indexing and caching. The *Indexing* component takes a process repository as input and constructs its alternative representation, called an *index*, which is then used to optimize computations during the execution of process queries. PQL implements indexing of the 4C behavioral predicates for all the process models in the repository. At runtime, when computing PQL programs, the precomputed behavioral relations are accessed in the index in close to real-time and reused. We plan to implement an additional index based on the special data structures, called *untanglings* of process models [20]. Untanglings can be used to efficiently identify groups of actions that can be executed together in some process. The *Caching* component stores data computed in the previous executions of PQL programs that then gets reused in computations of the future PQL programs. We plan to implement caching in PQL based on the statistics of the past PQL program executions.

The “Execute” part is responsible for executing process queries and comprises components for filtering process repositories and optimizing and executing process queries. For efficiency considerations, before a PQL program is executed, models that clearly should not be included in the result of the program are filtered away. The *Filtering* component of PQL checks whether actions that, according to the PQL program, must or must not be present in the result of the program are indeed described or not described, respectively, by the input model. We will extend this capability with filtering based on the untanglings to detect if combinations of actions can or cannot occur in an execution of the candidate model or process. Design and implementation of comprehensive query optimization mechanisms in PQL is future work. In the current implementation of the language, the execution plan of a PQL program is guided by its parse tree. Basic execution optimizations are supported. For example, when the result of a propositional logic formula is known based on

a subset of its propositions, the other propositions are not computed. Finally, the *Process Querying* component of the PQL method implements the formal semantics of the language; see [19, 22] for details. When a PQL program is executed, it takes as input a process repository and produces another repository consisting of the retrieved and manipulated, as requested by the PQL program, models.

The “Interpret” part of the framework is responsible for communicating the querying results to the user. All the components of this part aim to improve the comprehension of the results. The components are inspired by the various means for improving comprehension of conceptual models [13]. PQL results are encoded as process models or processes. The user can foster their understanding by inspecting, or reading, them. Future work will address the design, implementation, and evaluation of other techniques for explaining, projecting, translating, visualizing, animating, and simulating results of PQL programs for their better comprehension.

## 5 Implementation

The PQL querying method has been implemented in an open-source process repository.<sup>2</sup> Users interact with the repository via command-line interfaces (CLIs) of two utilities: the PQL bot and the PQL tool. The PQL bot prepares models for querying, while the PQL tool executes PQL programs over the models stored in the repository.

PQL programs process only indexed models. The PQL bot systematically indexes models in the repository. One can start multiple bot instances simultaneously to index multiple models in parallel. To construct an index, a bot instance computes all the 4C behavioral predicates over all the actions of the model using three types of analysis over the reachable states described by the model: the *reachability analysis* [9], the *coverability analysis* [26], and the *structural analysis over a complete prefix* [6, 15] of the *unfolding* [16] of the model. PQL bots use the solutions to the reachability and covering problems implemented in the `LOLA` tool version 2.0 [28]. The implementation of the algorithm by Esparza et al. [6], available as part of the `jBPT` library [24], is used to construct finite complete prefixes of unfoldings.

Process models stored in the repository are Petri nets described using the Petri Net Markup Language (PNML) [2]. Many high-level process modeling languages, such as BPMN and EPC, can be translated to Petri nets [4, 29]. As a result, PQL can be used to query and manipulate models developed using a wide range of notations.

The listing below shows a sample output of a PQL bot instance. One can configure a bot instance by specifying its name (option `-n`), time to sleep (i.e., stay idle) between indexing two models (option `-s`), and maximal time to attempt indexing a model (option `-i`). Once started, a bot instance alternates sleeping and indexing phases and sends periodic alive messages to the repository. Before indexing, models are checked for semantic correctness.

---

<sup>2</sup> <https://github.com/processquerying/PQL.git>.

```
>> java -jar PQL.BOT-1.0.jar -n=Brisbane -s=60 -i=3600
>> =====
>> Process Query Language (PQL) Bot ver. 1.0
>> =====
>> Name:                Brisbane
>> Sleep time:          60s
>> Max. index time:    3600s
>> =====
>> 10:45:18.487 Brisbane - There are no pending jobs
>> 10:45:18.487 Brisbane - Sent an alive message
>> 10:45:18.497 Brisbane - Going to sleep for 60 seconds
>> 10:46:18.505 Brisbane - Woke up
>> 10:46:18.525 Brisbane - Retrieved indexing job for the model with ID 1
>> 10:46:18.575 Brisbane - Start checking model with ID 1
>> 10:46:23.506 Brisbane - Finished checking model with ID 1
>> 10:46:23.506 Brisbane - Start indexing model with ID 1
>> 10:47:03.608 Brisbane - Finished indexing model with ID 1
>> 10:47:03.608 Brisbane - Going to sleep for 60 seconds
>> 10:48:03.613 Brisbane - Woke up
>> 10:48:03.623 Brisbane - Retrieved indexing job for the model with ID 2
>> 10:48:03.673 Brisbane - Start checking model with ID 2
>> 10:48:13.248 Brisbane - Finished checking model with ID 2
>> 10:48:13.249 Brisbane - Start indexing model with ID 2
>> 10:49:52.679 Brisbane - Finished indexing model with ID 2
>> 10:49:52.679 Brisbane - Going to sleep for 60 seconds
>> 10:50:52.704 Brisbane - Woke up
>> 10:50:52.704 Brisbane - There are no pending jobs
>> ...
```

Table 6 lists several CLI options of the PQL tool. For example, the PQL tool can be used to store (option `-s`), check (option `-c`), index (option `-i`), and delete (option `-d`) a process model, visualize the parse tree of a PQL program (option `-p`), execute a PQL program (options `-q`), and to reset the repository (option `-r`).

To store models in the repository, the CLI option `-s` of the PQL tool must be accompanied by the `-pnml` option that specifies a path to a single PNML file or to a directory that contains many PNML files. If a path to a single PNML file is specified, the call must include option `-id` to specify a unique identifier to associate with the model; otherwise, the models are attempted to be stored using their file names as unique identifiers. A stored model can be indexed by a PQL bot instance or by the PQL tool using the CLI option `-i` accompanied by option `-id` that specifies the

**Table 6** CLI options of the PQL tool

Option name	Short name	Parameter	Description	Required option
<code>-check</code>	<code>-c</code>		Check if model can be indexed	<code>-id</code>
<code>-delete</code>	<code>-d</code>		Delete model (and its index)	<code>-id</code>
<code>-index</code>	<code>-i</code>		Index model	<code>-id</code>
<code>-identifier</code>	<code>-id</code>	<string>	Model identifier	
<code>-parse</code>	<code>-p</code>		Show PQL program parse tree	<code>-pql</code>
<code>-pnmlPath</code>	<code>-pnml</code>	<path>	Path to a PNML file	
<code>-pqlPath</code>	<code>-pql</code>	<path>	Path to a PQL file	
<code>-query</code>	<code>-q</code>		Execute PQL program	<code>-pql</code>
<code>-reset</code>	<code>-r</code>		Reset repository	
<code>-store</code>	<code>-s</code>		Store model in the repository	<code>-pnml (-id)</code>

unique identifier of the model that should be indexed. When indexing a model, the PQL tool uses the same routines as the PQL bot.

To execute a PQL program, the user can use options `-q` and `-pql` of the PQL tool. The latter specifies a path to a file that contains the program. An example command-line output of executing a PQL program is shown below. Here, the PQL tool is requested to execute the PQL program stored in the `prog.pql` file. The program requests to retrieve every model in the repository in which the “process payment” action, or a similar action, occurs in every execution the model describes; note that two similar actions, “process payment by cash” and “process payment by check”, were found in the repository for the requested similarity threshold of 0.8. The tool retrieved two models that match the query. These are models with identifiers 364 and 778; see the last line of the listing.

```
>> java -jar PQL.TOOL-1.0.jar -q -pql=prog.pql
>> PQL query: SELECT * FROM * WHERE AlwaysOccurs("process payment"[0.8]);
>> Attributes: [UNIVERSE]
>> Locations: [UNIVERSE]
>> Task:      "process payment"[0.8] -> ["process payment by cash",
>>      "process payment by check"]
>>
>> Result:    [364, 778]
```

The PQL tool supports multi-threaded querying. The user can configure the desired number of threads to use for executing PQL programs. As a result of executing a PQL program, the tool returns a collection of matching and augmented models.

## 6 Discussion

The design of PQL aims to maximize the number of supported process querying and process manipulation techniques, as requested by the *process querying compromise* [21], which identifies a concrete process querying method as an intersection of implemented decidable, efficient, and suitable techniques. In this section, we discuss research problems that emerged during the design of PQL, and solutions to these problems that shaped PQL and will inform the future extensions to the language. First, Sect. 6.1 discusses four fundamental problems of process querying that PQL aims to solve. Then, Sect. 6.2 discusses problems that aim to ensure the quality of process querying and manipulation operations performed by PQL. Next, Sect. 6.3 summarizes conducted work to establish the suitability of PQL. Finally, Sect. 6.4 is devoted to the aspects related to the ability to compute PQL queries efficiently.

## 6.1 *Querying and Manipulation*

Given a process model and a process query that describes a collection of processes, the process querying problem is a decision problem that consists in checking whether the model describes processes from the collection.

**Process querying problem.** Given a process model and a description of a collection of processes, check if the model describes processes included in the collection.

PQL can be used to pose and solve process querying problems via `SELECT` statements. One may want to augment a process model so that the collection of processes it describes includes specified processes. This task can be fulfilled by solving the process insertion problem.

**Process insertion problem.** Given a process model and a description of a collection of processes, construct a process model that describes processes captured in the model and included in the collection.

PQL `INSERT` statements can be used to express and solve process insertion problems. In contrast, if a model needs to be augmented to describe processes of the original model without some specific processes, a process deletion problem must be solved.

**Process deletion problem.** Given a process model and a description of a collection of processes, construct a process model that describes processes captured in the model but not included in the collection.

One can use PQL `DELETE` statements to formulate and solve process deletion problems. However, if specific processes must be replaced in the collection of processes described by a model, a process update problem must be solved.

**Process update problem.** Given a process model, a description of a collection of source processes, and a description of a collection of target processes, construct a process model that describes processes captured in the model and included in the target collection but not included in the source collection.

PQL `UPDATE` statements can be used for expressing and solving process update problems. Future solutions to the above four problems will be considered for inclusion in PQL by implementing and offering them to the users via the corresponding PQL statements.

## 6.2 *Quality*

Given a process model and a query, process querying solves a decision problem with a yes-or-no answer that indicates whether the model matches the query or not.

The quality of such a decision is also binary; the decision is either correct or not. Process manipulation is different, as a requested manipulation can be fulfilled to various degrees. To compare methods for manipulating process models, either to select a method to implement as part of PQL or to choose an already implemented method for triggering during PQL query execution, one should be able to measure and compare their quality in terms of the resulting models they produce. The quality of manipulated process models can be compared against different aspects. Several of these aspects are discussed below, giving rise to three research problems.

**Simplicity problem.** A process model that results from a solution to a process insertion, deletion, or update problem should be simple.

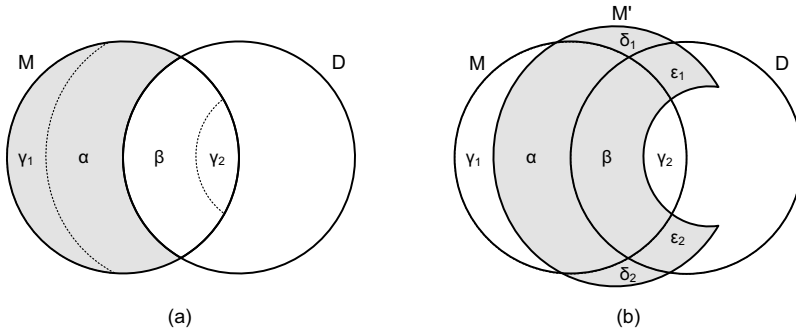
It may be necessary to manually analyze a process model that results from PQL manipulations, for example, to obtain feedback on the model from a process analyst or a domain expert. Hence, the manipulated models must be comprehensible. That is, they should be simple to understand for human readers. Simplicity is the desired quality for many artifacts automatically learned from data using data mining and process mining techniques. The simplicity criteria for learned models are often implemented as realizations of the Occam's Razor principle [8] that states that a model should use as few constructs as possible. Alternatively, this principle can be interpreted as if a model should not be overcomplicated without necessity. Consequently, existing simplicity criteria [10–12] from the field of process mining [30] can be reused to assess the simplicity of the manipulated by PQL models. The model simplicity criteria that will be developed in the future may consider the specifics of the process manipulation problems, refer to Sect. 6.1.

**Resemblance problem.** A process model that results from a solution to a process insertion, deletion, or update problem for a given process model should resemble the original model.

As PQL manipulations are applied over a given process model, it may be desirable that a resulting manipulated model resembles the original model. This desire, again, can stem from the potential necessity to assess manipulated models manually, this time in the context of the original model. Indeed, the user may know the model they request to manipulate and, consequently, expect that the resulting model is not radically different from the model they know, especially if the intended changes to the model are not extensive. This intention to keep resemblance with the original model is similar to the desire of repaired models, studied in process mining [30], to resemble the original models that were repaired. Thus, measures of model resemblance developed in the context of process manipulation can draw inspiration from the corresponding measures studied as part of the process repair problem [7, 17].

**Correctness problem.** A process model that results from a solution to a process insertion, deletion, or update problem should describe the requested processes.

A solution to a process manipulation problem, either an insertion, deletion, or an update problem, should construct a process model that describes a specific,



**Fig. 6** A schematic visualization of the participating process collections in the context of a solution to the process deletion problem: **(a)** the problem definition and **(b)** a possible problem solution

requested collection of processes. However, methods for process manipulation can produce models that do not fulfill this correctness criterion; for instance, to avoid constructing complex models or models that do not resemble the input models. Various measures can be introduced to assess the correctness of manipulated models in terms of the processes they describe. These measures can quantify and compare collections of processes that were requested and appeared, were requested and did not appear, were not requested and appeared, and were not requested and did not appear in the processes described by the manipulated model.

Figure 6a visualizes an example process deletion problem schematically. Concretely, given a model that describes a collection of processes  $M$ , the problem requests to construct a model that describes processes captured in the input model but not in a collection of processes  $D$ . Hence, the resulting model should describe the collection of processes  $M \setminus D$ , denoted by the shaded region in the figure. In turn, Fig. 6b shows a collection of processes  $M'$  described by some model constructed as a solution to the problem superposed on the two process collections from Fig. 6a. Several sets of processes emerge in this situation. Processes  $\alpha$  are the processes that should and are described by the resulting model, while processes  $\beta$  are the processes that should not but are described by the resulting model. The resulting model does not describe processes  $\gamma_1$  and  $\gamma_2$ . However, while processes  $\gamma_2$  were correctly deleted, processes  $\gamma_1$  should be present in  $M'$ . Processes  $\delta_1 \cup \delta_2$  are not participating in the problem definition but are described by the resulting model. Finally, processes  $\epsilon_1 \cup \epsilon_2$  were requested to be deleted and were not described by the input model, but ended up as described by the resulting model. A good solution to the process deletion problem should aim to minimize the sizes of sets  $\gamma_1$ ,  $\beta$ ,  $\delta_1$ ,  $\delta_2$ ,  $\epsilon_1$ , and  $\epsilon_2$ . The measures of the correctness of process manipulations should quantify this intuition to support the design of correct methods. Here, again, we can learn from the subarea of conformance checking [3, 23] in process mining [30], which studies ways to diagnose commonalities and discrepancies between processes.

Consider models 8 and 9 in Fig. 3 that can result from executing query Q6 presented in Sect. 2 on model 3 from Fig. 1. If we apply the reasoning from Fig. 6 to these two models, then for model 8 it holds that  $\beta$  is empty and  $\gamma_1 = \text{AG}(\text{DED})^+$ , where  $\gamma_1$  is specified by a regular expression, while for model 9 sets  $\beta$  and  $\gamma_1$  are both empty. Hence, model 9 can be considered as a more correct, and hence a better, result of query Q6 than model 8.

### 6.3 Suitability

The suitability of a method refers to its quality of being appropriate for a purpose. Conducted empirical studies on the suitability of the current process querying methods guide their design and implementation.

To evaluate the suitability of the 4C behavioral predicates for the purpose of process querying and to identify the most relevant predicates to implement in PQL, we performed a user study [19]. In that study, we conducted semi-structured interviews with business analysts that actively work with process models. In the interviews, besides explaining the high-level design of PQL, we tested the understanding of twelve preselected 4C predicates and asked to evaluate their ability to fulfill the process querying tasks. The twelve predicates were selected to ensure they include, and combine in different ways, all the features of all the 4C predicates. Our questions to the stakeholders probed usefulness, importance, likelihood, and frequency of using the predicates in daily work. All the predicates were identified as suitable, while the six most relevant were implemented in PQL. These are the `CanOccur`, `AlwaysOccurs`, `Cooccur`, `Conflict`, `TotalCausal`, and `TotalConcurrent` predicates.

Process querying grounded in the collection of the 4C predicates, or any other set of similar predicates, has a fundamental limitation. A querying method that relies on a finite number of behavioral predicates can distinguish between a finite number of model classes [18], where any two models from the same class are considered equivalent by every query. The scenario-based process querying facilities of PQL extend its expressiveness [22]. PQL querying based on traces with wildcards, as explained in Sect. 3.1.2, can be used to express an intent to retrieve a model that describes processes that contain, or do not contain, *any* finite collection of processes and, thus, can be used to discriminate infinitely many models.

Future studies will strengthen the current results on the suitability of PQL for fulfilling process querying and process manipulation tasks.

### 6.4 Decidability and Efficiency

Karsten Wolf demonstrated that computations of all the 4C predicates currently implemented in PQL can be reduced, often via exponential space transformations,



to model checking [32]. In that work of Karsten Wolf, the reduction of one 4C predicate, namely the *total existential concurrent* predicate, was left open, and its decidability, for the general case, is currently unknown. In addition, the proposed transformations for four 4C predicates are applicable only in the special case of the absence of auto-concurrency in process models. Note that model checking over infinite-state systems is undecidable and is PSPACE-complete over finite-state systems [5], making it from challenging to impossible to evaluate the predicates at runtime. Hence, we precompute and store values of the predicates we can obtain in an index and access this index in close to real-time during the computation of PQL queries.

To perform scenario-based querying, that is, to check if a model describes a process that matches a sequence of actions with wildcards (see queries Q3 and Q4 in Sect. 2), first, the queried model gets transformed. The size of the transformed model is proportional to the size of the model and the scenario of interest. Then, an *optimal alignment* between the transformed model and a sequence of actions induced by the scenario of interest is constructed, and its cost is analyzed. The problem of computing an optimal alignment is equivalent to the reachability problem [1, 31], which is decidable [27] with the exponential space as the lower bound [14]. Despite its high computational complexity, the proposed method works in close to real-time on industrial and synthetic models [22]. To speed up query processing, we propose to use the untangling-based index [20] that allows identifying models that describe a process in which all actions from the scenario of interest occur. Then, further checks should be applied to verify if the actions occur in a requested order.

PQL queries that solve the process insertion problem are implemented using the impact-driven process model repair method [17]. Similar to scenario-based querying, the method relies on optimal alignments to compute queries. However, in this case, the alignments are used to identify the minimal required changes to the model to fulfill the query.

## 7 Conclusion

This chapter gives an overview of PQL, a domain-specific programming language for process querying and process manipulation. PQL is a declarative language with the SQL-like syntax. It is useful for managing process models stored in process repositories based on the processes that these models describe. Process querying is supported in PQL by means of the `SELECT` statements, while process manipulation is implemented using the `INSERT`, `DELETE`, and `UPDATE` statements. The chapter also discusses the currently supported features of the language, a publicly available implementation of a process repository with PQL support, future design and implementation efforts aimed at shaping the language, and open research problems triggered by the design of the language.

## References

1. Adriansyah, A.: Aligning observed and modeled behavior. Ph.D. thesis, TU/e (2014). <http://dx.doi.org/10.6100/IR770080>
2. Billington, J., Christensen, S., van Hee, K.M., Kindler, E., Kummer, O., Petrucci, L., Post, R., Stehno, C., Weber, M.: The Petri net markup language: Concepts, technology, and tools. In: ICATPN. LNCS, vol. 2679, pp. 483–505. Springer, New York (2003)
3. Carmona, J., van Dongen, B.F., Solti, A., Weidlich, M.: Conformance Checking - Relating Processes and Models. Springer, New York (2018). <https://doi.org/10.1007/978-3-319-99414-7>
4. Dijkman, R.M., Dumas, M., Ouyang, C.: Semantics and analysis of business process models in BPMN. *Inf. Softw. Technol.* **50**(12), 1281–1294 (2008)
5. Esparza, J., Nielsen, M.: Decidability issues for Petri nets—a survey. *Bull. EATCS* **52**, 244–262 (1994)
6. Esparza, J., Römer, S., Vogler, W.: An improvement of McMillan’s unfolding algorithm. *Formal Methods Syst. Des.* **20**(3), 285–310 (2002)
7. Fahland, D., van der Aalst, W.M.: Model repair—aligning process models to reality. *Inf. Syst.* **47**, 220–243 (2015). <http://dx.doi.org/10.1016/j.is.2013.12.007>
8. Grünwald, P.D.: The Minimum Description Length Principle (Adaptive Computation and Machine Learning). The MIT Press, Cambridge (2007)
9. Hack, M.: Decidability Questions for Petri Nets. Outstanding Dissertations in the Computer Sciences. Garland Publishing, New York (1975)
10. Kalenkova, A.A., Polyvyanyy, A., Rosa, M.L.: A framework for estimating simplicity of automatically discovered process models based on structural and behavioral characteristics. In: BPM. Lecture Notes in Computer Science, vol. 12168, pp. 129–146. Springer, New York (2020)
11. Laue, R., Gruhn, V.: Complexity metrics for business process models. In: BIS. LNI, vol. P-85, pp. 1–12. GI (2006)
12. Lieben, J., Jouck, T., Depaire, B., Jans, M.: An improved way for measuring simplicity during process discovery. In: EOMAS@CAiSE. Lecture Notes in Business Information Processing, vol. 332, pp. 49–62. Springer, New York (2018)
13. Lindland, O.I., Sindre, G., Sølvsberg, A.: Understanding quality in conceptual modeling. *IEEE Softw.* **11**(2), 42–49 (1994)
14. Lipton, R.: The Reachability Problem Requires Exponential Space. Research report. Department of Computer Science, Yale University (1976)
15. McMillan, K.L.: Using unfoldings to avoid the state explosion problem in the verification of asynchronous circuits. In: Computer Aided Verification (CAV). Lecture Notes in Computer Science, vol. 663, pp. 164–177. Springer, New York (1992)
16. Nielsen, M., Plotkin, G.D., Winskel, G.: Petri nets, event structures and domains, Part I. *Theor. Comput. Sci.* **13**, 85–108 (1981)
17. Polyvyanyy, A., van der Aalst, W.M.P., ter Hofstede, A.H.M., Wynn, M.T.: Impact-driven process model repair. *ACM Trans. Softw. Eng. Methodol.* **25**(4), 28:1–28:60 (2017)
18. Polyvyanyy, A., Armas-Cervantes, A., Dumas, M., García-Bañuelos, L.: On the expressive power of behavioral profiles. *Formal Asp. Comput.* **28**(4), 597–613 (2016). <http://dx.doi.org/10.1007/s00165-016-0372-4>
19. Polyvyanyy, A., ter Hofstede, A.H.M., Rosa, M.L., Ouyang, C., Pika, A.: Process query language: Design, implementation, and evaluation. *CoRR* abs/1909.09543 (2019)
20. Polyvyanyy, A., La Rosa, M., ter Hofstede, A.H.M.: Indexing and efficient instance-based retrieval of process models using untanglings. In: CAiSE. LNCS, vol. 8484, pp. 439–456. Springer, New York (2014)
21. Polyvyanyy, A., Ouyang, C., Barros, A., van der Aalst, W.M.P.: Process querying: Enabling business intelligence through query-based process analytics. *Dec. Support Syst.* **100**, 41–56 (2017). <https://doi.org/10.1016/j.dss.2017.04.011>

22. Polyvyanyy, A., Pika, A., ter Hofstede, A.H.M.: Scenario-based process querying for compliance, reuse, and standardization. *Inf. Syst.* **93**, 101563 (2020)
23. Polyvyanyy, A., Solti, A., Weidlich, M., Ciccio, C.D., Mendling, J.: Monotone precision and recall measures for comparing executions and specifications of dynamic systems. *ACM Trans. Softw. Eng. Methodol.* **29**(3), 17:1–17:41 (2020)
24. Polyvyanyy, A., Weidlich, M.: Towards a compendium of process technologies: the jBPT library for process model analysis. In: CAiSE Forum, *CEUR Workshop Proceedings*, vol. 998, pp. 106–113. CEUR-WS.org (2013)
25. Polyvyanyy, A., Weidlich, M., Conforti, R., Rosa, M.L., ter Hofstede, A.H.M.: The 4C spectrum of fundamental behavioral relations for concurrent systems. In: *Petri Nets. LNCS*, vol. 8489, pp. 210–232. Springer, New York (2014)
26. Rackoff, C.: The covering and boundedness problems for vector addition systems. *Theor. Comput. Sci.* **6**, 223–231 (1978)
27. Reutenauer, C.: *The Mathematics of Petri Nets*. Prentice-Hall, Inc., Upper Saddle River, NJ (1990)
28. Schmidt, K.: LoLA: A low level analyser. In: *Application and Theory of Petri Nets (ICATPN)*. Lecture Notes in Computer Science, vol. 1825, pp. 465–474. Springer, New York (2000)
29. van der Aalst, W.M.P.: Formalization and verification of event-driven process chains. *Inf. Softw. Technol.* **41**(10), 639–650 (1999)
30. van der Aalst, W.M.P.: *Process Mining—Data Science in Action*, 2nd edn. Springer, New York (2016)
31. van der Aalst, W.M.P., Adriansyah, A., van Dongen, B.F.: Replaying history on process models for conformance checking and performance analysis. *Data Min. Knowl. Discov.* **2**(2), 182–192 (2012). <https://doi.org/10.1002/widm.1045>
32. Wolf, K.: Interleaving based model checking of concurrency and causality. *Fund. Inf.* **161**(4), 423–445 (2018). <https://doi.org/10.3233/FI-2018-1709>

**Part III**  
**Event Log and Process Model Querying**

# Business Process Query Language



Mariusz Momotko and Kazimierz Subieta

**Abstract** Modern Business Process Management systems have to work effectively in a distributed cloud environment and to adapt quickly to dynamic changes. One of the key approaches to increase business process adaptability is to make process definitions more flexible. Usually, this requires to express complex constraints and conditions within a process definition. These complex elements are related to the history of process execution, current organizational and application data. In addition, such complex constraints and conditions should be represented in a standardized and yet simple way. In order to satisfy the above requirements, we need: (1) a business process metamodel that includes proper data structures for process definitions and the history of their execution; (2) a powerful and easy to understand language to query models instantiated from the metamodel; (3) integration of that query language with a widely used business process definition language. In this chapter, we propose Business Process Query Language (BPQL) together with the Business Process Metamodel. BPQL is integrated with the Business Process Model and Notation language increasing significantly its expressiveness and flexibility. We also present results of applying BPQL in the OfficeObjects® WorkFlow system.

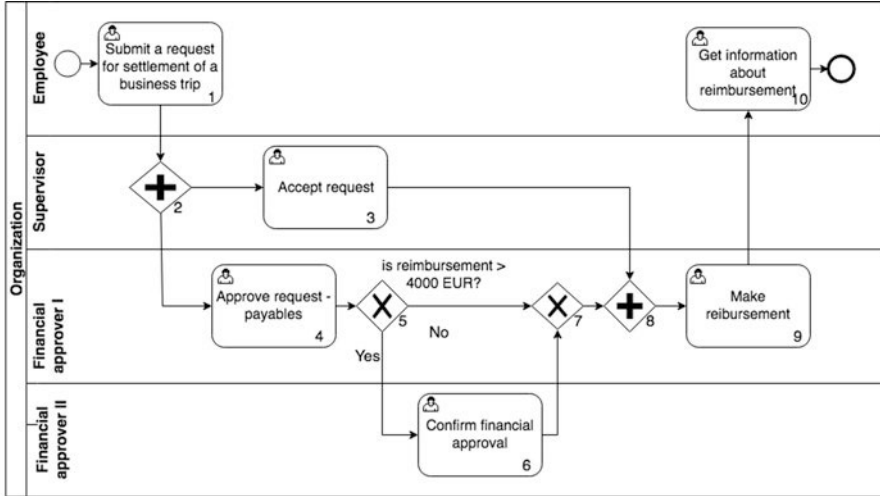
## 1 Introduction

In the last two decades Business Process Management systems (BPM systems) made a successful career. The BPM systems have been used for implementing various types of business processes. Despite many advantages of applying those systems, several significant limitations were observed. One of the major restrictions was an assumption that business processes do not change too often during their

---

M. Momotko (✉)  
Amazon Alexa, Amazon Development Center, Gdansk, Poland

K. Subieta  
Polish-Japanese Academy of Information Technology, Warsaw, Poland  
e-mail: [subieta@pjwstk.edu.pl](mailto:subieta@pjwstk.edu.pl)



**Fig. 1** A simplified process for settlement of travel expenses

execution. While such assumption may be satisfied for majority of production processes, for less rigid processes, such as administration ones, this is not true. Because of the nature of the latter processes, they need to adapt to dynamic changes [10, 14, 15] in business process environment (i.e., services, data, and (human) resources) and benefit from information about their execution.

For example, we can identify such needs in a process for settlement of travel expenses for a business trip. Its simplified definition is presented in Fig. 1. According to this definition, an employee can submit a request for settlement of travel expenses. This request is then verified and accepted by both the employee’s supervisor and a financial approver (level I). If travel costs are significant, the request needs to be approved by a second financial approver (level II). If all approvals are collected, the original financial approver makes a reimbursement. Eventually, the employee gets information<sup>1</sup> about the reimbursement, which was made and completes the process.

The process definition also includes some additional constraints on participant assignment. The employee’s supervisor must be a person who is a direct supervisor or, if there is none, an employee from HR department possessing a role “Travel.Reimbursement.DefaultApprover”. The financial approver (level I) must play the role of “Travel.Reimbursement.FinApprover”. In addition, in order to optimize the overall processing time, this approver must be selected from all the available financial approvers as the one with minimal (current) workload. The most

<sup>1</sup> This activity could be implemented in a different way, for example, as a message sent to the employee. To make it simple we assumed that it is represented by an atomic activity and connected with the rest of the process via control flow.

complex constraint is defined for the second financial approver. He/she must be a person who plays a role of “Travel.Reimbursement.FinApprover” and be not the person who made the first approval. Yet, the second approver must be chosen as the person with minimal workload too.

In order to satisfy the above requirements, we need to make process definition more flexible and include conditions on participant assignment, which query:

- Organizational structure data (e.g., who possesses a given role, who is a supervisor for a given person).
- Process definition data and its execution history (e.g., who performed a first activity, who performed financial approval).
- Current state of execution of all processes in a given organization (e.g., who, of a given role, has minimal current workload).

These conditions must be evaluated during process execution when performers of relevant activities are determined.

There are many ways to represent such conditions in a process definition. Most often, such conditions are hard-coded. This solution is efficient; however, any change to these conditions requires code re-compilation. In addition, readability of such hard-coded solution is limited and, practically, only programmers may modify and deploy such changes.

An alternative approach is based on using a standardized query language that is able to represent the mentioned conditions. As described in [5] and [7] such language must be expressive enough to define queries on process definition data, history of process executions as well as relevant data that come from business process environment (e.g., organizational structure data). Yet, this language must be understandable by process designers and have well-defined meaning of all the constructs it uses.

In this chapter, we propose BPQL—a Business Process Query Language that is able to satisfy the above requirements. In order to achieve this, we first define a business process metamodel. This metamodel is then used to represent models (in terms of data structures) for process definitions and history of their executions. Those models can be queried by BPQL. Next, we specify BPQL syntax and explain its semantics. Then we present how BPQL may be used to define built-in process monitoring functions. We also define how BPQL may be integrated with a standard process definition language and specify an architecture for a BPM system extended by BPQL. Finally, we show real BPQL use cases and summarize the results discussing opportunities for future extensions. In all consecutive sections of this chapter we use the same example process for settlement of travel expenses.

## 2 Business Process Metamodel

According to Business Process Model and Notation of Object Management Group (BPMN v2.0 of OMG, [8]), a *business process* is a sequence or flow of activities in an organization with the objective of carrying out work. Usually, a business

process is modeled (graphical notation) and defined (more detailed specification) in a standard business process language, such as BPMN. Such definition is then used to execute the business process. A single process definition may be used many times for different business process execution.

A *business process metamodel* is a conceptual view (a schema) over data structures, which is able to represent all possible business processes. Usually, definition of data structures for a business process includes structures used for its modeling and definition. However, complete data structures for a business process should also include data structures related to process execution. An example of some data structures on process execution are the process execution state (e.g., to understand if a given process execution is running or is postponed) and the way how activities are executed (e.g., how many performers executed this activity, when it was started, and when it was completed).

In this section, we propose a business process metamodel that is able to represent both definition and execution structures for all possible business processes. Thus, this metamodel includes two parts: process definition part and process execution part, which are related to each other. In addition, the metamodel defines core elements of business process environment that are needed to execute business processes. An instance of the metamodel is a *business process model*. This model consists of instantiation of data structures representing definition of a given business process together with instantiation of data structures representing all performed executions of this process (according to its definition). A single execution of a business process is called a *process instance*. In a given BPM system, the metamodel specification together with all instantiated data structures are stored in a *Business Process Repository*. This is presented in Fig. 2.

The proposed metamodel addresses all entities required to define and execute business processes. The entities managed by a BPM system are explicitly defined within the metamodel while the entities managed by the other IT systems are only referenced within the metamodel. The *process definition* part of the metamodel

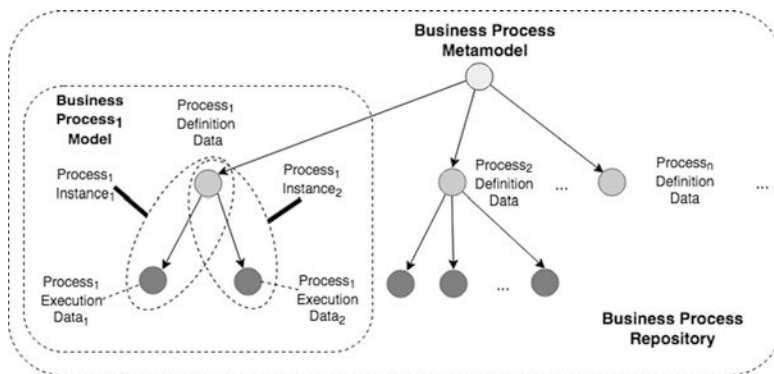


Fig. 2 Business process metamodel and examples of its models



defines the top-level entities, their relationships, and basic attributes. This part is used to design and implement a computer representation of business processes. The *process execution* part of the metamodel defines the top-level entities, their relationships, and basic attributes. This part is used to represent process executions performed according to process definitions. Entities defined within the process definition parts are a subset of entities defined in BPMN. Entities defined within the process execution part have the same execution semantics as defined for BPMN. The entities managed by other IT systems are treated as a BPM environment.

A BPM system uses data on users, roles, and organizational structure to assign potential performers of human tasks during defining a process, as well as to evaluate such assignments and determine actual performers during process execution. For non-human, service tasks, BPM system uses data on IT services to define how to invoke them, especially in terms of input and output parameters. In many IT environments data on services are provided by a dedicated Service Registry. During process execution, services are called directly by the BPM system. A BPM system provides two main services to IT environment: it defines and executes business processes. It also manages a BPM repository that stores all data on process definitions and all their performed executions. An architecture explaining interfaces between a BPM system and IT environment is presented in Fig. 3.

A high-level view of the metamodel is presented as a Unified Modeling Language (UML) class diagram in Fig. 4. The main entity of the definition part of the metamodel is *ProcessDef*. It provides basic information about the computer representation of a business process. For every process a set of *ContainerAttDef* is defined. These attributes are used during process execution for evaluation of conditional expressions, such as transition conditions or pre- and post-conditions. The set of container attributes depends on individual process definitions.

Process definition consists of activities. An *Activity* defines a piece of work that forms a single logical step within a process. There are three types of activities: atomic, compound, and route activities. An *Atomic activity* is the smallest logical step within a process that cannot be divided. For every atomic activity it is possible to define *who* can perform it, *how* it can be executed, and *what* data it can process.

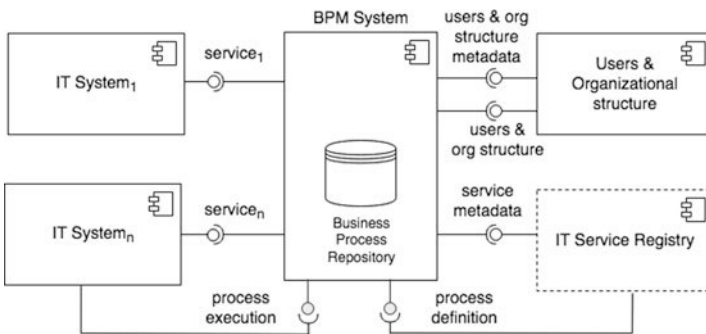


Fig. 3 BPM system as a part of an IT environment

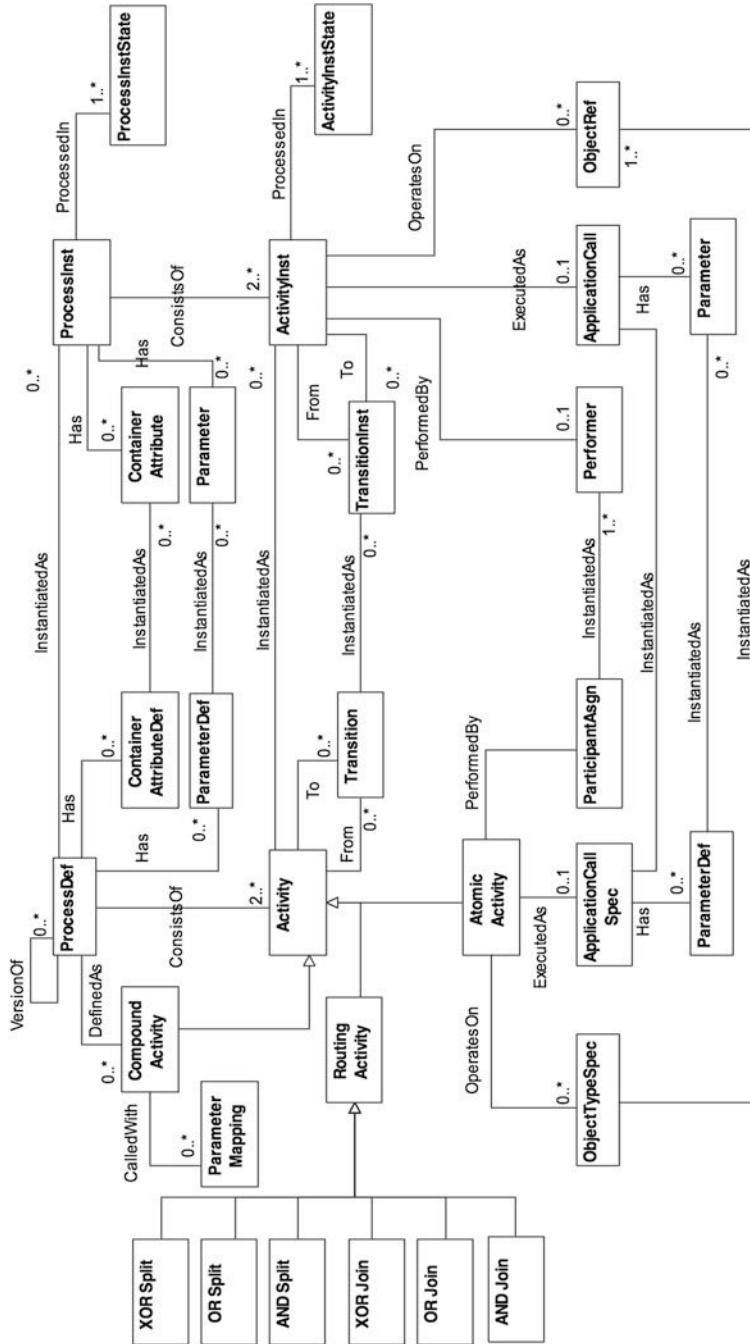


Fig. 4 The business process metamodel

An atomic activity may be performed by zero, one, or more participants. A *participant* is a user (or a group of users), a role, or an organizational unit. Zero participants assigned means that a given activity is executed by a BPM system. Specification of participants that can perform a given activity is called (*workflow*) *participant assignment*. The way how an activity is performed is specified by an *application* that is executed on behalf of the activity. Such specification also includes a set of parameters that are passed to the application. It is represented in the metamodel by *ApplicationCallSpec*. Since the mentioned application operates on data, also object types that will be processed (i.e., created, modified, read, or deleted) by this activity must be defined. Object types are represented in the metamodel as *ObjectTypeSpec*.

The second type of activity is a *compound activity*, which helps in splitting a complex process into smaller parts. Those parts can be modeled and managed separately. A compound activity is represented as a sub-process of a given process. It can be nested and include other compound activities.

The last type of activity is a *route activity*. This type of activity performs no work processing, neither object types nor applications are associated with it. A route activity is used to express two core control flow operators: split and join operators. There are three basic types defined for both operators, namely AND, OR, and XOR. More advanced control flow constructs can be built using those operators. In BPMN specification, splits and joins are called *gateways*.

The order of activities in a process is defined by transitions. A *transition* from one activity to another activity may be conditional (involving expressions that are evaluated to permit or inhibit the transition) or unconditional.

Once a process is defined, it can be executed many times. Execution of a process according to its definition is called a *process enactment*. The context of an enactment includes performers, relevant data, and application call parameters.

The representation of a single enactment is called a *process instance*. Its lifecycle is expressed by states and is described as a UML state chart diagram. The history of states for a given process instance is represented by process instance state entities. Every process instance has its own data container. This container is an instantiation of a data container type defined for a process and includes container attributes used to control process execution (e.g., as a part of transition conditions).

Execution of a process instance may be regarded as execution of a set of activity instances. An activity instance is performed either by a single human performer or by a BPM system by invoking a single application. If for a given activity, its participant assignment evaluates to more than one performer, then execution of this activity is represented by a set of activity instances. In such case, the number of activity instances is equal to the number of the activity performers. In case of an application invocation, its input and output parameters are instances of data object types assigned to the activity during process definition.

A route activity is performed automatically by the system, and there is neither application nor data objects assigned to it. If an activity instance is a compound activity, its execution is represented by a dedicated process instance performed according to the definition of that activity.

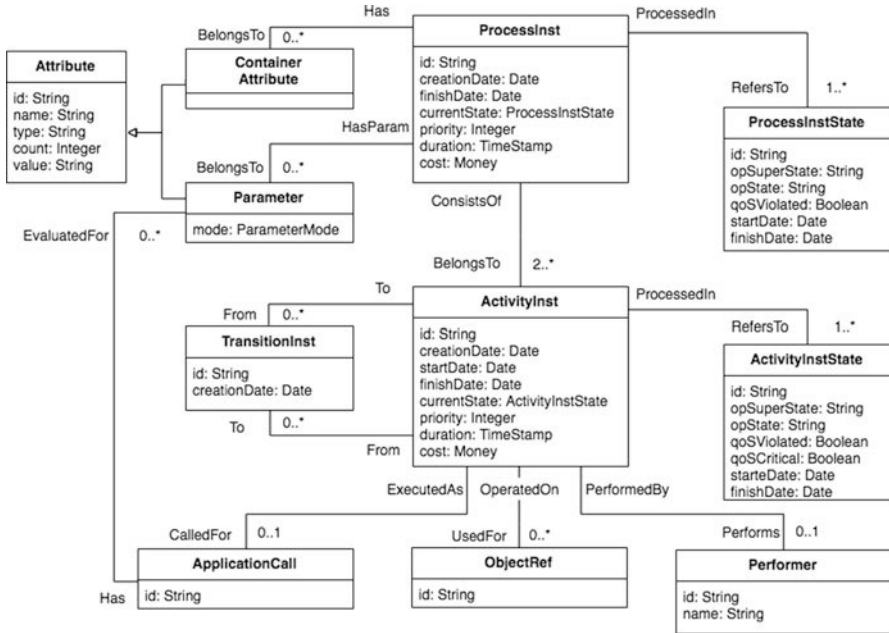


Fig. 5 The business process metamodel—process execution part

Similarly to process instance, the life cycle of an activity instance is represented by a state diagram and is stored as a collection of state entities. Flow between activity instances is represented by *transition instances*. When an activity instance is accomplished, the system checks which transitions that are going from this activity may be instantiated. If a transition has no condition or the transition condition is satisfied, it is automatically instantiated by the BPM system. A transition instance may be considered as a relation “predecessor-successor” between two activity instances.

The above description of the metamodel allows users to ask BPQL queries at the level of entities. However, most of BPQL queries operates at the level of individual attributes of the metamodel. An example of such detailing for the process execution part is presented in Fig. 5. A complete, detailed description of the whole metamodel is included in [5], Chapter 2.

Knowing the metamodel, we define a process model for execution of our example process for settlement of travel expenses (see Fig. 1). For simplicity, presentation of the model is limited to execution of its first four activities as shown in Fig. 6.

The process definition part includes a *ProcessDef* object and *Activity* objects. The latter objects are connected via *Transition* objects. The process execution part includes a *ProcessInst* object and *ActivityInst* objects. The latter object is connected via *TransitionInst* objects. Every object has a unique identifier. The objects from execution part always have references to relevant objects from the definition part via *instantiatedAs* relations. For example *ActivityInst* object with

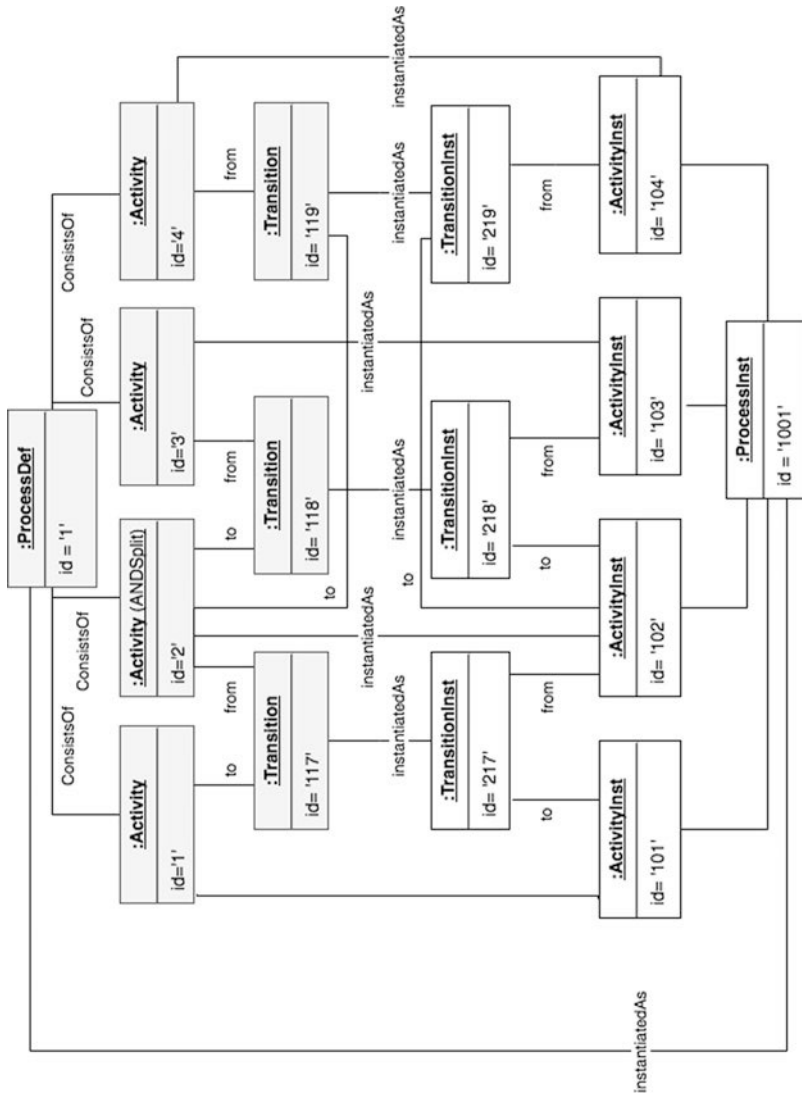


Fig. 6 A fragment of a model instantiated from the metamodel of the process for settlement of travel expenses

$id=“103”$ <sup>2</sup> represents execution of “Accept request” activity with  $activityId=“3”$ . Therefore, it is always clear what is the order of execution and what activities and transitions were instantiated during the execution.

### 3 Query Language

The metamodel described in the previous section specifies business process entities and relationships. In order to extract information from process models derived from this metamodel, we propose BPQL—a business process query language.

Conceptually, BPQL is based on Stack-Based Approach (SBA, [1, 11–13]). SBA allows BPQL to define its operational semantics based on an abstract machine and operations, in which names, their bindings, and scopes defined by query and data structure are central. BPQL, as an object-oriented query language, operates natively on models derived from the Business Process Metamodel.

Before BPQL syntax and semantics is introduced, we explain some example BPQL queries. All these queries use the execution part of the Business Process Metamodel defined in the previous section. To visualize how the queries are constructed, we present a (simplified) execution part of the metamodel together with two example models in Fig. 7.

Below, we define six simple BPQL queries that query the example models.

```

1: ProcessInst
2: ProcessInst.id
3: ProcessInst where (id = '123')
4: (ProcessInst where (id = '123')).ConsistsOf.ActivityInst
5: count((ProcessInst where (id = '123')).ConsistsOf.ActivityInst)
6: ActivityInst where (PerformedBy.Performer.name='johnb')
```

The first query (line 1) returns all *ProcessInst* objects, that is two objects (with id “123” and “124”). The second query (line 2) gets all *ProcessInst* objects and then for every object returns its identifier. Thus the result of the second query consists of two identifiers (i.e., “123” and “124”). The third query (line 3) filters the returned *ProcessInst* objects. Since process instance identifier is unique at the level of the BPM system, only one object (i.e., the one with id=“123”) is returned. The fourth query (line 4) filters *ProcessInst* objects using an identifier and for the filtered objects gets, via *ConsistsOf* relation, all the relevant *ActivityInst* objects. In our example, two *ActivityInst* objects with identifiers “12” and “13” are returned. The fifth query (line 5) counts all *ActivityInst* objects returned by the fourth query. The query returns the value of 2. The last query (line 6) filters all *ActivityInst* objects that are performed by “johnb”. The condition to filter *ActivityInst* objects is defined on *Performer.name* attribute. A *Performer* object is reached from *ActivityInst* object

<sup>2</sup> The identifiers used in BPQL are strings, not integers. Such solution is more flexible and allows to add additional affixes (e.g., to group identifiers by the metamodel object type).

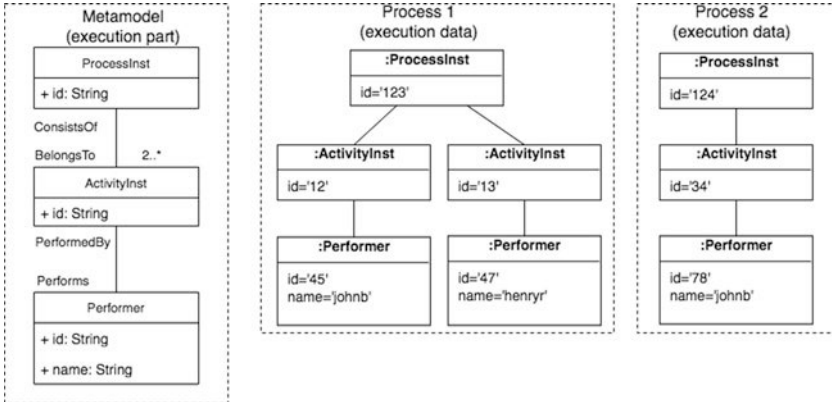


Fig. 7 An extract of the execution part of the Business Process Metamodel with example models

via relation “PerformedBy” and the query returns two *ActivityInst* objects (with id “12” and “34”).

### 3.1 Syntax

Syntax determines how BPQL queries are constructed. BPQL syntax is defined by language tokens and syntactic rules. A BPQL *language token* is a character string that can be included in a BPQL query. The following sets of language tokens are defined in BPQL:

- *L*—set of literals. A literal may be numeric (i.e., an integer or float point number), Boolean (i.e., true or false), or alpha-numeric (i.e., a string of characters surrounded by single quotes). Some literals may have additional meaning such as alpha-numeric literals that are considered as dates.
- *N*—set of entity names (e.g., attribute names, association names, class names). This set is shared by all BPQL queries, which operate over a given process model. The set also includes ad-hoc names that users can use in queries.
- *O*—set of BPQL operator names, imperative construct names, function and procedure names (both built-in and user-defined), and other reserved words. The content of this set is common for all BPQL queries.
- Parentheses and other tokens to determine syntactic structures of queries.

The sets of tokens are used in definition of the *syntactic rules* of BPQL queries:

- R1** Any element of *L* is a query. For instance, 2, 3.14, “alaBama123”, “2004-05-07”, and *true* are queries.
- R2** Any element of *N* is a query. For example: *ProcessInst* (class), *priority* (attribute), and *InstantiatedAs* (association) are queries.

- R3** If  $\Delta$  is an algebraic unary operator and  $q$  is a query, then  $\Delta(q)$  is a query. Examples of such operators are: *sqrt*, *sum*, *avg*, *log*,  $-$ , *not*. For selected operators (e.g.,  $-$ , *not*) the brackets may be omitted.
- R4** If  $\Delta$  is an algebraic binary operator and  $q_1, q_2$  are queries, then  $q_1 \Delta q_2$  is a query. Examples of such operator are: *and*, *or*,  $=$ ,  $+$ ,  $-$ ,  $*$ ,  $/$ ,  $<$ , and  $>$ .
- R5** If  $\theta$  is a non-algebraic binary operator and  $q_1, q_2$  are queries, then  $q_1 \theta q_2$  is a query. Examples of such operator are selection (where), dependent join (join), projection/navigation ( $\cdot$ ), and existential and universal quantifiers.
- R6** If  $q$  is a query and  $n \in N$  then  $q \text{ as } n$  is a query. The “as” operator is used to introduce a temporal name (so called synonym).
- R7** If  $q$  is a query and  $n \in N$  then  $q \text{ group as } n$  is a query.
- R8** If  $q$  is a query, then  $(q)$  is a query.
- R9** If  $n \in O$  and  $q_1, \dots, q_k$  are queries, then  $n(q_1, \dots, q_k)$  is a query.
- R10** If  $n \in O$  without parameters, then  $n()$  and  $n$  are queries.
- R11** If  $q_1, \dots, q_k$  are queries, then  $(q_1, \dots, q_k)$ , *struct*( $q_1, \dots, q_k$ ), *sequence* ( $q_1, \dots, q_k$ ), and *bag*( $q_1, \dots, q_k$ ) are queries.
- R12** If  $q_1, q_2, q_3$  are queries, then *if*  $q_1$  *then*  $q_2$  and *if*  $q_1$  *then*  $q_2$  *else*  $q_3$  are queries.

The BPQL grammar in Extended Backus–Naur Form is presented in Table 1.

## 3.2 Semantics

BPQL semantics is based on an abstract machine that executes a query and delivers its result. In this chapter, we define this machine as a recursive evaluation procedure that operates on a Data Storage (DS), ENVIRONMENT Stack (ENVS), and Query RESULT Stack (QRES). This evaluation procedure is used to define the meaning of the individual BPQL operators (algebraic and non-algebraic) and imperative constructs. The approach is universal, fully precise w.r.t. all BPQL operators, and makes it possible to develop powerful query optimization methods (such as factoring of independent sub-queries, removing dead sub-queries, indexing, etc.).

### 3.2.1 Architecture of the Query Evaluation Mechanism

The architecture of the BPQL query evaluation mechanism is presented in Fig. 8.

During execution of a BPQL query, the query evaluation mechanism operates on business process objects, uses local variables, and calls appropriate procedures. Business process objects that instantiate entities from the Business Process Meta-model are managed by DS. Local variables, function call parameters, and other query processing elements (e.g., parameter name binding) are controlled by ENVS. The entities stored on this stack depend on the data storage content and current query results. During query evaluation, all the intermediate results are managed by QRES.



**Table 1** EBNF notation of BPQL

Rule	Additional assumptions	
<query>	::= <query l>   <algExpr>   <logExpr>	
<query l>	::= <literal>   <name>   <procedureCall>   <query><collectionOp><query>   <collection>(<queryList>)   <query> [ <b>as</b>   <b>group as</b> ] <aliasName>   <query> [ <b>where</b>   <b>.</b>   <b>join</b> ] <query>   <b>for</b> [ <b>some</b>   <b>any</b> ] <query>(<query>)   <query> <b>order by</b> <query>	<name> ∈ N  <aliasName> ∈ N
<algExpr>	::= <algSum>	
<algSum>	::= <algProduct>{ [+   - ] <algSum> }	
<algProduct>	::= <algItem>{ [ *   / ] <algProduct> }	
<algItem>	::= -<algItem>   (<algSum>)   <query l>	
<logExpr>	::= <logSum>	
<logSum>	::= <logProduct>{ <b>or</b> <logSum> }	
<logProduct>	::= <logItem>{ <b>and</b> <logProduct> }	
<logItem>	::= <b>not</b> <logItem>   <logCondition>   (<logSum>)	
<logCondition>	::= <query l><opComp><query l>   <query l>	
<opComp>	::= <   <=   =   >   < >   <b>like</b>	
<CollectionOp>	::= <b>union</b>   <b>intersect</b>   <b>diff</b>   <b>in</b>	
<collection>	::= [ <b>struct</b>   <b>bag</b>   <b>sequence</b> ] (<queryList>)	
<procedureCall>	::= <pName>({ (<queryList> )})	<pName> ∈ N
<procedure>	::= <pName>{ <procParams> } <sBlock>	<pName> ∈ N
<procParams>	::= ({ <param> { , <param> } })	
<param>	::= { <b>out</b> } <parName>	<parName> ∈ N
<sBlock>	::= <statement>;	
<sBlock>	::= '{ <statement>; { <statement>; } }'	
<statement>	::= <create>   <assignment>   <forEach>   <ifThenElse>   <switchCase>   <whileDo>   <doWhile>   <procedureCall>   <b>return</b> <query>   <b>break</b>   <b>continue</b>	
<assignment>	::= <lValue>=<query>	<lValue> ∈ N
<forEach>	::= <b>for each</b> <query> <b>do</b> <sBlock>	
<ifThenElse>	::= <b>if</b> <querySL> <b>then</b> <sBlock>{ <b>else</b> <sBlock> }	
<switchCase>	::= <b>switch</b> (<query>) '{ <case> { <case> } }'	
<case>	::= [ <b>case</b>   <b>default</b> ]:<sBlock>	
<doWhile>	::= <b>do</b> <sBlock> <b>while</b> (<query>)	
<whileDo>	::= <b>while</b> (<query>) <b>do</b> <sBlock>	
<queryList>	::= <query>{ , <query> }	
<querySL>	::= <query>	Result always a Boolean
<literal>	::= <text>   <integer>   <float>   <boolean>	
<boolean>	::= <b>true</b>   <b>false</b>	

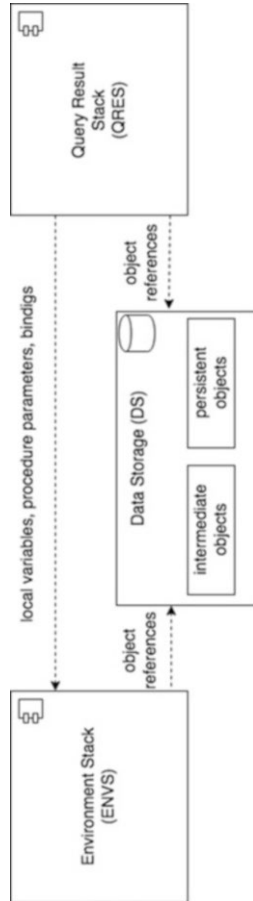


Fig. 8 Architecture of the BPQL query evaluation mechanism

Both stacks can also contain other elements, such as numerical values, string values, binders (named values). ENVs and QRES provide four standard stack operations:

- Push—Add a new element at the top of the stack. The number of the elements in the stack increases by one.
- Pop—Extract the element that is placed on the top of the stack. The number of elements in the stack decreases by one.
- Top—Read the top of the stack.
- Empty—Check whether the stack is empty.

In addition, ENVs provides a search for the required elements from top to bottom.

### 3.2.2 Environment Stack

ENVs (aka call stack) is used to store all local programming entities that are necessary to execute a given procedure call (or function/method/operation call). Examples of such entities are local variables, local constants, and procedure parameters. The content of the call stack strictly depends on the operational context (local and global environment), that is, on the place in the program where a given programming command is executed. The ENVs stack is responsible for:

- Controlling the scope of names occurring in queries and their current bindings.
- Storing local objects or variables for procedures, functions, and methods calls.
- Storing procedure, function, and method arguments (parameters).
- Keeping the return trace for procedures, functions, and methods calls.

The basic assumptions for ENVs are the following:

- Single objects (i.e., one-element collection) and collections of objects stored in ENVs are treated in the same way.
- From the conceptual point of view, the maximal size of ENVs is unlimited.
- ENVs consists of sections. Every section includes information about the environment of a recognizable part of BPQL query. For example, an environment for a function call consists of local variables, local constants, and function parameters.
- The most recent, local section (e.g., section of the procedure which is currently executed) is placed on the top of ENVs. Sections for the functions that were called earlier are located in ENVs lower than the most recent, local section.
- At the bottom of the stack global sections are located. They include common function libraries, global variables, references to database objects, etc.
- ENVs treats persistent and intermediate objects in the same way.
- ENVs stores information about query definitions (e.g., synonyms, quantifier variables).
- ENVs stores references to the objects, not the objects themselves. The objects are stored in DS.

ENVs consists of sections, which are ordered and managed according to the stack principle. Each section is a set of binders. A *binder* is a pair  $\langle n, x \rangle$  (denoted by

$n(x)$ ) where  $n$  is a name (e.g., object name, procedure name, variable name, attribute name, etc.) and  $x$  is a run time entity (usually an object reference) having this name. The role of the  $n(x)$  binder is to bind name  $n$  in the query. The result of binding is  $x$ . For any name occurring in a query, a corresponding binder should exist on ENVs. This stack consists of sections, which are ordered and managed according to the stack principle. Bindings are searched in the following order: first the “youngest” section at the ENVs top is visited, then the section below the top of the stack, etc. up to the global sections on the ENVs bottom. Detailed description of the ENVs mechanism can be found in [13].

### 3.2.3 Query Result Stack

QRES is a generalization of the arithmetic stack known from programming languages. When query evaluation starts, QRES is empty. After the query evaluation, QRES contains the final result. A query result may include one or more objects. If a BPQL function (or operator) is evaluated and it requires  $n$  arguments, then these arguments are taken one after another from the top of the QRES stack (i.e., pop operation on  $n$  stack elements). After evaluation, the function result is stored on the top of the stack (i.e., push operation). QRES can contain other elements required to evaluate queries; see [5] for details.

### 3.2.4 Query Evaluation Procedure

The query evaluation procedure (referred subsequently to as *eval*) takes a BPQL query as input, evaluates it, and returns its result in QRES. The procedure is driven by a parser that constructs the syntactic tree of the query. The tree can be generated earlier and stored in DS. The *eval* procedure in pseudo-code is summarized below:

```

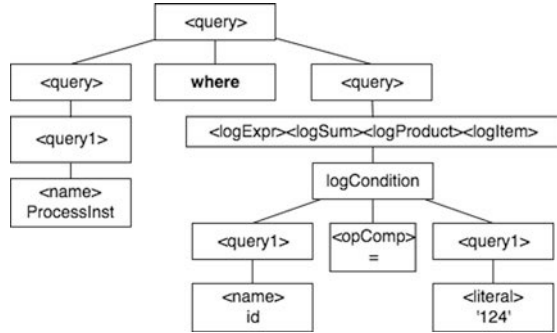
1:   procedure eval(q: BPQL query)
2:   begin
3:       parse(q); // generate or get already generated syntactical tree
4:       case q recognised as literal:
5:         ...
6:       case q recognised as name:
7:         ...
8:   end;
```

During execution, the *eval* procedure operates on process objects stored in DS, ENVs stack, and QRES stack. The final result of the *eval* procedure execution is stored on the QRES top.

The definition of BPQL semantics is driven by its syntax. That means the *eval* procedure is supported by a parser that is able to generate a syntactic tree of the query (i.e., sub-queries, operators). An example of such syntactic tree for a simple query: *ProcessInst where id = "124"* is presented in Fig. 9.

The result of a query evaluation is determined on the basics of the sub-queries evaluation results and appropriate operators. The evaluation process is recursive. First, the leaves of the syntactic tree are evaluated, and then again evaluation results

**Fig. 9** The syntactic tree of a simple query



together with appropriate operators are evaluated. Such evaluation is repeated until the top of the syntactic tree is reached and the final query evaluation result is determined. The intermediate results stored in the QRES stack are consumed by BPQL operators. For example, the equality operator takes two elements from the QRES top (via the pop operation), compares them, and saves the Boolean result back on the QRES top (via the push operation).

### 3.2.5 Collections and Structs

A *collection* is a group of objects or values that need to be processed together. The size of collection is not restricted. Usually collections contain elements of the same type, although this is not always obligatory. BPQL supports two types of collections: bags and sequences. The basic collection type is a *bag*. All BPQL queries results are returned as bags (sometimes as one-element bags). A bag may contain duplicate elements. An example of a query that returns a bag of activity instances that have been performed by “johnb” is given below:

```
ActivityInst where ((PerformedBy.Performer.name) = 'johnb')
```

There are five basic operators on collections, namely union (union), intersection (intersect), difference (diff), subset (in), equality (=). The meaning of these operators is similar as for those defined in the set theory. In addition, BPQL has the *struct* (structure) construct that generalizes the corresponding construct known from C/C++, Java, CORBA, ODMG, etc. A *struct* collects (named) values of different types. Usually the size and type of structures are known during compile time. The order of elements within struct can be important.

### 3.2.6 Literals and Names

An elementary BPQL query is a literal ( $l \in L$ ) or a name ( $n \in N$ ). A fragment of the *eval* procedure for such queries expressed in the pseudo-code is presented below:

```

1:     procedure eval(q: BPQL query)
2:     ...
3:     parse(q); // syntactical tree generation
4:     case q recognised as literal:
5:     push(QRES, literal);
6:     case q recognised as name:
7:     push(QRES, bind(name));
8:     ...

```

If the query is a literal (line 4), the *eval* procedure pushes it on the QRES top (line 5). If the query is a name (line 6), the *eval* procedure calls the *bind* function with this name as a parameter. The function accomplishes binding the name on the ENVStack, and then the binding result is pushed on the QRES top (line 7).

### 3.2.7 Algebraic Operators

In BPQL, there are unary and binary algebraic operators. All algebraic operators operate only on the objects stored in QRES and return new results. Pseudo-code representation of the *eval* procedure is given below:<sup>3</sup>

```

1:     procedure eval(q: BPQL query)
2:     ...
3:     case q recognised as  $\Delta(q_1)$  or  $\Delta q_1$ :
4:     begin
5:     resQ, resQ1: Result;
6:     eval(q1); // evaluate q1, its result is stored on the QRES top
7:     resQ1:= top(QRES); pop(QRES); // pop q1 from the QRES top
8:     resQ:= $\underline{\Delta}$ (resQ1); // run the function
9:     push(QRES, resQ); // store the result on the QRES top
10:    end;
11:    case q recognised as q1  $\Delta$  q2:
12:    begin
13:    resQ, resQ1, resQ2: Result;
14:    eval(q1); // evaluate q1, store its result on the QRES top
15:    eval(q2); // evaluate q2, store its result on the QRES top
16:    resQ2:= top(QRES); pop(QRES); // pop q2 evaluation result
17:    resQ1:= top(QRES); pop(QRES); // pop q1 evaluation result
18:    resQ:= $\underline{\Delta}$ (resQ1, resQ2); // run the function
19:    push(QRES, resQ); // store the overall query evaluation result the QRES top
20:    end;
21:    ...

```

### 3.2.8 Non-algebraic Operators

Formalization of non-algebraic operators is the essence of the stack-based approach. All non-algebraic operators are binary. This group includes selection, dependent join, projection/navigation, existential and universal quantifiers, and transitive closures. As discussed in [7], semantics of non-algebraic operators cannot be expressed using relational or object algebras. To cope with this challenge, non-algebraic operators are parameterized by expressions. For instance, selection

---

<sup>3</sup> The symbol “ $\Delta$ ” denotes an algebraic operator, the symbol “ $\underline{\Delta}$ ”—function that implements this operator,  $q_1$  and  $q_2$  represent BPQL queries.

operator is parameterized by a condition. Such a condition does not belong to the language of algebra but to its not formalized meta language. This mix of formal and informal notions is frustrating and problematic in didactics and implementation. This problem has been solved in SBA in which non-algebraic operators are not parameterized by (informal) meta-language names or conditions. This is the fundamental difference to the ideas provided by relational and object-oriented algebras. In consequence, unlike algebras, in SBA there is no need to subdivide names into “first class” (e.g., object names) and “second class” (e.g., attribute names). Every name occurring in a query is considered on the same semantic level and is processed by the scope and binding rules. This unification has a great advantage for implementation and query optimization.

Semantics of non-algebraic operators may be expressed using the *eval* procedure introduced previously using the same formal pattern. In addition to the functions used for other BPQL operators, the *eval* procedure requires a functionality to traverse via composed object and expose its individual parts for further processing. For that purpose, we define a function named *nested*. For an object provided as in input, this function returns a set of binders for the individual attributes of this object. For example, having an ActivityInst and Performer objects defined for the Process2 model in Fig. 7:<sup>4</sup>

```
<'34', 'ActivityInst', {<i1, 'performedBy', '78'>}>
<'78', 'Performer', {<i2, 'name', 'johnb'>}>
```

The nested function for these objects results in following binders:

```
nested('34') = {performedBy(i1)}
nested('78') = {name(i2)}
```

A generalized algorithm for nested function (*r* means a single result) is defined as follows:

- If *r* is an identifier of a complex object, then *nested(r)* returns binders referring to sub-objects of this object, as shown above.
- If *r* is a binder, then *nested(r)*=*r* (i.e., the result contains this single binder).
- If *r* is an identifier of a linked object, then *nested* returns a set containing a single binder referring to an object pointed to by *r*.
- For structures, *nested(struct{v<sub>1</sub>,v<sub>2</sub>,v<sub>3</sub>,...})*=*nested(v<sub>1</sub>)*∪*nested(v<sub>2</sub>)*∪*nested(v<sub>3</sub>)*...
- For any other cases, *nested* returns an empty set.

In order to evaluate the query  $q_1\theta q_2$ , the following activities have to be performed:

1. Evaluate  $q_1$  and return a collection (bag) of elements.
2. For every element *e* that belongs to the returned collection:

<sup>4</sup> An object is represented by a triple  $\langle i, n, v \rangle$  representing object identifier, name, and value, respectively.

- a. Calculate the value of the function :  $nested(e)$ . The result is a set of binders.
  - b. Insert the calculated set of binders as a new section on the top of ENVs.
  - c. Evaluate  $q_2$  in this new environment.
  - d. Calculate the intermediate result for  $e$  by joining it with every result returned by  $q_2$ . The function join strictly depends on the type of operator  $\theta$ .
  - e. Remove the inserted ENVs section.
3. Aggregate all intermediate results into the final one. The way how the intermediate results are aggregated depends strictly on the type of operator  $\theta$ .

A fragment of the *eval* procedure for non-algebraic operators is presented below. The meaning of  $sum\theta$  function depends on individual non-algebraic operators and is briefly explained below.

```

1:  procedure eval(q: BPQL query)
2:  ...
3:  case q recognised as  $q_1 \theta q_2$  : // non-algebraic binary operators
4:  begin
5:  tmpResBag: bag of Result; // set of all temporal results
6:  tmpRes : Result; // single temporal results
7:  finalRes : Result; // final query results
8:  singleRes: Result; // single result of  $q_1$ 
9:  tmpResBag :=  $\emptyset$ ;
10: eval( $q_1$ ); // evaluate  $q_1$ , store its result on QRES top
11: for each e in top(QRES) do // for each single result of  $q_1$ 
12:   begin
13:    push(ENVs, nested(e)); // creates a new section on ENVs stack
14:    eval( $q_2$ ); // evaluate  $q_2$ , store its result on QRES top
15:    tmpRes := join(e, top(QRES)); // join single result of  $q_1$  with  $q_2$ ;
16:    tmpResBag := tmpResBag union {tmpRes};
17:    pop(QRES); pop(ENVs);
18:   end;
19:   finalRes := sum $\theta$ (tmpResBag); // sum all temporary results
20:   pop(QRES) // remove from QRES the results of  $q_1$  evaluation
21:   push(QRES, finalRes) // store the final results on QRES top
22: end;
23: ...

```

**Selection** is used to extract objects of a given type that satisfy a set of criteria. In BPQL, selection is expressed as the binary *where* operator and has the following syntax:  $q_1$  *where*  $q_2$ . Its left argument is a query to retrieve selection objects. Its right argument is a query that represents a selection criterion. The latter query must return a Boolean value. For example, we may use selection if we want to extract a set of activity instances that had been created before the 1<sup>st</sup> January 2004.

```
ActivityInst where (creationDate < '2004-01-01')
```

According to the *eval* procedure, for every object  $e$  of the result retrieved by  $q_1$  the join function returns: (a) one-element bag if the query  $q_2$  operated on  $e$  returned *true*, (b) empty bag if the query  $q_2$  operated on  $e$  returned *false*. The  $sum\theta$  function aggregates all the intermediate results.

**Projection** (or navigation) is used to extract objects that are related to the other objects in some way (e.g., by association or being a part of). In BPQL, projection is expressed by the “.” operator and has the following syntax:  $q_1.q_2$ . The left argument is a query that defines the way to get to the objects returned by the right



argument. According to the *eval* procedure, the *join* function returns the results retrieved by  $q_2$  ignoring the results retrieved by  $q_1$ . The *sum $\theta$*  function aggregates all the intermediate results of  $q_2$ . For example, we may use several projections (aka path expression) to extract all names of performers that participate in any activity instance.

```
ActivityInst.PerformedBy.Performer.name
```

**Existential quantifier** checks if there is at least one value that satisfies a given condition. In BPQL, existential quantifier is represented by *for some* operator and has the following syntax: *for some*  $q_1(q_2)$  (equivalent to  $q_1 \exists q_2$ ). Its left argument defines a collection of values that are checked. Its right argument is a query representing a condition. We use this operator in the prefix syntax rather than in the infix syntax as for other non-algebraic operators. For example, we may use the existential quantifier to check whether there exists an activity instance started before 1<sup>st</sup> of January 2004, which has not been finished yet (i.e., its current operational superstate is different than “Finished”).

```
for some (ActivityInst as ai)
(ai.creationDate < '2004-01-01' and ai.currState.opSuperState <> 'Finished')
```

According to the *eval* procedure, the *join* function returns all the values returned by  $q_2$  ignoring values returned by  $q_1$ . The *sum $\theta$*  function returns: a) *true* if at least one intermediate result returned from  $q_2$  is true, b) *false* if all the intermediate results returned from  $q_2$  are false.

**Universal quantifier** is used to check whether the given condition is satisfied for all the checked values. In BPQL, the universal quantifier is expressed by *for all* operator and has the following syntax: *for all*  $q_1(q_2)$  (equivalent to  $q_1 \forall q_2$ ). Its left argument defines a collection of values that are checked. Its right argument represents a condition that has to be satisfied for all the values. For example, we may use a universal quantifier to check whether all activity instances started before the 1<sup>st</sup> of January 2004 have already finished (i.e., its current operational superstate is “Finished”).

```
for all (ActivityInst as ai)
(ai.creationDate < '2004-01-01' and ai.currState.opSuperState = 'Finished')
```

According to the *eval* procedure, the *join* function returns all the values returned by  $q_2$  ignoring values returned by  $q_1$ . The *sum $\theta$*  function returns *true* if all the values returned by  $q_2$  are true (or if the number of the values is zero); *false* otherwise.

### 3.2.9 Imperative Constructs

Imperative constructs enable BPM systems to express complex requests that require some programming features. Together with procedures and functions, imperative

constructs can be used to simplify BPQL queries. Below we present some basic imperative constructs available in BPQL.

A *statement* is a programming instruction. Every imperative construct is a statement. The simplest statement is an assignment. A *statements block* is a set of statements that should be considered and executed as a group. A statement block can consist of a single statement. If a statement block includes more than one statement, two additional tokens are used to begin (“{”) and end (“}”) the block.

**Control flow operators** are used to change the sequence of executed statements. In BPQL there are two such operators: *if-then-else* operator and *switch-case* operator. The former operator has the following syntax:

*if query then block-of-statements<sub>1</sub> else block-of-statements<sub>2</sub>.*

The query must return a Boolean value. If the returned value is *true*, then *block-of-statements<sub>1</sub>* is executed. Otherwise, *block-of-statements<sub>2</sub>* is executed. There is also a simplified version of this operator: *if query then block-of-statements<sub>1</sub>*. Its meaning is the same as for the appropriate parts of the previous form. An example of how this operator may be applied is given below. If “johnb” has no task assigned, the query returns a Resource object that represents “johnb”. Otherwise, it returns the resource with the minimal number of tasks currently assigned:

```

if for some (ActivityInst as a)
(a.currState.opSuperState='Running' and a.PerformedBy.Performer.name='johnb')
then return Resource where (
count (isPerformer.Performer.Performs.ActivityInst where
(currState.opSuperState) = 'Running') =
min(Resource.count (IsPerformer.Performer.Performs. ActivityInst where
(currState.opSuperState) = 'Running')));
else return Resource where name = 'johnb';

```

**Switch-case** operator has the following syntax: *switch (query) do list-of-cases*. The query must return a single value *s* of a numeric, Boolean, or string type. The *list-of-cases* includes one or more cases. A single case has the following syntax: *case value: block-of-statements*. If the value is equal to *s*, then *block-of-statements* is executed. After that the other cases are checked against the value. To leave a switch-case operator one has to use the *break* statement.

**For-each operator** is used to process a collection of values returned by a query; one after another. In BPQL, for-each operator is expressed by the *for each* keyword and has the following syntax: *for each query do block-of-statements*. The query returns a collection of values and the statement block is executed for every element of the returned collection. For example, one can use the for each operator to suspend all the activity instances delayed for more than 20 days.

```

for each (ActivityInst where
(DiffDate(CurrDate(),deadline,'d')>20) and (currState.opSuperState)='Running')
do currState.opState:='Suspended';

```

### 3.2.10 Procedures and Functions

BPQL provides mechanisms to define procedures and functions. This mechanism can be used to simplify complex queries and to encapsulate repeatable query parts. A *procedure* is a block of statements. A procedure may have input and output (with *out* keyword) parameters. The former is a call-by-value parameter that can only be read in the procedure body. The latter is a call-by-reference parameter and may be used to modify objects that are kept outside the procedure as local environment. The procedure body consists of one or more statements. Every statement ends with semicolon. The *return* operator may be treated as a special statement that finishes the procedure immediately. A function is a procedure that may return a value compatible with results of queries. In the stack-based approach implemented in BPQL, procedures can be recursive. Semantics for a procedure/function call is defined as follows (*eval* procedure):

1. Bind the procedure name on ENVS and get a reference to the procedure.
2. Initialize a procedure call. ENVS is extended with a new section (an activation record) having the following information: (i) binders storing procedure parameters, (ii) binders to local objects of the procedure, and (iii) a return trace to return to the caller code after terminating the procedure.
3. Evaluate procedure parameters (BPQL queries) and store results on QRES top.
4. Store procedure parameters binders in the activation record according to their values stored at QRES. Remove these values from QRES.
5. If local objects were declared, they are stored as volatile objects. Their binders are inserted into the activation record on ENVS.
6. The procedure body is executed. Queries in the procedure body have access to all ENVS binders, including volatile objects and parameters of the procedure.
7. If the procedure control reaches a return statement, then it finishes. For a return value, it is evaluated and its result is stored on the top of QRES.
8. The procedure is finished. All intermediate objects are being removed. ENVS is cleared from the activation record. The system returns to the point of program according to the return trace. The procedure result is stored at the top of QRES.

### 3.2.11 Predefined Context-Dependent Functions

We also need in BPQL some functions that would extract the context of evaluated queries. For example, if one wants to assign the same performer as for previously executed activity, one needs to know what is the current activity instance and then find its predecessor. Thus in BPQL we introduce *BPQLQueryContext* object with two attributes: *processInstId* and *activityInstId* and propose two BPQL functions:

- *ThisProcessInst* returning a reference to the process instance (its identifier is equal to the *processInstId* attribute).
- *ThisActivityInst* returning a reference to the activity instance (its identifier is equal to the *activityInstId* attribute).

## 4 Monitoring Functions

BPQL constructs can express most, if not all, useful queries over the proposed metamodel. For most real applications, however, these queries are complex and include repeatable fragments. Definition of the same fragments many times may be quite annoying and cause additional mistakes.

To avoid such problems, we propose a set of predefined BPQL functions that can be used to build more advanced BPQL queries or functions. Since these functions mostly query the execution data of the models instantiated from the metamodel, we refer to them as *BPQL monitoring functions*. There are three main types of BPQL monitoring functions: participant assignment functions, process flow functions, and quality of service functions. A list of about 50 BPQL monitoring functions can be found in [5], Appendix A.

**Participant assignment functions** are used for selecting “best” participants to perform given activities. There are various criteria that define what “best” means. The most popular techniques are based on calculation of the number of the assigned tasks, their overall cost, or the total time required to perform them.

An example of such a participant assignment function is a function to calculate the current number of the assigned activity instances or tasks for given participants. An assigned task is defined as an activity instance for which the current superstate is “Created” or “Running”.<sup>5</sup> This example implemented as BPQL function is presented below. The function calculates the number of assigned/performed tasks for every resource passed as its argument.

```

1: procedure GetPerformerWorkload(inputPerfList) {
2:   return (inputPerfList as e).
3:   (
4:     (e as performer),
5:     count(e.Performs.ActivityInst
6:       where (currState.opSuperState) in ('Created', 'Running')) as workload
7:   )
8: };
9: }

```

The function gets *Performer* objects as its input (line 1) and returns relevant (*performer, workload*) structures (lines 4 to 7). In order to generate the structure for every participant provided in the input, we use a BPQL projection operator (lines 2 to 7). The left argument of the operator is represented by the participant objects (line 2) while the right argument is the mentioned structure (line 3 to 7).

**Process flow functions** simplify query operations on business process models. The process flow functions are provided for process definition and process execution parts. Examples of process flow functions include:

- *StartActivity, EndActivity* - start/end activities for given process definitions
- *SuccActivity, PredActivity* - successors/predecessors of an activity

<sup>5</sup> Specification of lifecycle states for an activity instance is provided in [5].

- *StartActivityInst, EndActivityInst* - start/end activity instance (process instance)
- *SuccActivityInst, PredActivityInst* - activity instance successors/predecessors

**Quality of Service (QoS) functions** supports QoS management. This management operates on QoS factors. Following the concept presented in [3, 4], and [16] three main basic QoS factors for business processes include time, cost, and reliability. Usually these factors are set according to the initial business process requirements during the business process definition. Then, the required QoS factors are compared with those retrieved from the business process execution. Finally, the results of the comparison are used to improve business processes and make them possible to meet the initial QoS requirements. These factors are defined on four levels, namely activity instance, process instance, activity, and process levels.

#### 4.1 Settlement of Travel Expenses Example

The process for settlement of travel expenses has been presented in Fig. 1 and described in Sect. 1. Next, we come back to this example and define in BPQL the most advanced participant assignment for the financial approver II re-using one of already defined BPQL monitoring functions. To make this participant assignment more generic, we implement it as a parameterized BPQL function.

```

1: procedure GetPerformerByRoleWithExcludedActPerformer(role, actInstId) {
2:   return
3:   (GetPerformerWorkload(
4:     (Performer where
5:       (id in GetResIdListByRole(role))
6:     )
7:     intersect
8:     (ThisProcessInst.ConsistsOf.ActivityInst where id=actInstId)
9:     .PerformedBy.Performer
10:  )
11:  group as perfList)
12:  .(perfList where workload = min(perfList.workload))
13:  .performer.id;
14: }

```

Line 1 specifies the interface of the BPQL function. It has two input parameters: expected performer role and activity instance identifier executed by a performer who should be excluded from this searching. The function returns an identifier of the performer who has minimal workload. Lines 3 to 10 represent an invocation of *GetPerformerWorkload* BPQL function defined in the previous section. This function gets a collection of *Performer* objects and returns a collection of (*performer, workload*) structures. The input collection of performers (lines 4 to 9) is determined as an intersection (line 7) of employees who play the role provided as the first input parameter (lines 4 to 6) and the employee who performed activity instance provided as the second input parameter (lines 8, 9). To get the collection of performers, we use function *GetResourceIdListByRole* that, for a given role name, returns a collection of employees' identifiers. We assume that this function is provided to the BPM system

via organizational service. To get the performer of the activity instance provided as the second input parameter, we look for an *ActivityInst* object that represents execution of the activity of *id=actInstId* and then via *PerformedBy* relation extracts the relevant *Performer* object. The collection of structures (*performer*, *workload*) returned by *GetPerformerWorkload* function is grouped by the name *perfList* (line 11). Next, this collection is used to find another structure that represents a performer with minimal workload (line 12). To find this performer, we use *min* function that operates on *workload* attribute of all the mentioned structures grouped as *perfList*. Finally, from the structure with minimal workload we get the *performer* attribute representing a *Performer* object and extract its identifier (line 13). An invocation of the above BPQL function for the process for settlement of travel expenses is presented below.

```
1: GetBestPerformerByRoleWithExcludedActPerformer (
2:   'Travel.Reimbursement.FinancialApprover', '4')
```

## 5 Architecture and Standardization

After defining the metamodel and BPQL, in the next step we explain how BPQL can be integrated with existing BPM solutions in order to increase business process adaptability and to make process definitions more flexible. Two requirements must be satisfied to achieve this goal. Firstly, BPQL queries should be a part of a process definition that is evaluated during process execution. Secondly, BPQL as a software component must be integrated with a BPM system that executes such extended process definitions. This section address both these requirements. We present how BPQL may extend BPMN and then show an architecture of a BPM system extended with BPQL. Finally, we compare this tandem (i.e., BPM+BPQL) with Process Querying Framework (PQF, [9]).

### 5.1 BPQL Embedded in BPMN

BPMN is a standardized language to model and define business processes. BPMN definition of a business process can include expressions that are evaluated during process execution. A default language to define those expressions is XPath; however, BPMN allows to use any other business process query language such as BPQL. To use another language, a BPMN definition of a business process must specify *Definitions.expressionLanguage* attribute as it is presented below.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<definitions id="Definition"
...
xmlns="http://www.omg.org/spec/BPMN/20100524/MODEL"
```

```

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
expressionLanguage="http://bpql.org/BPQL/v1">
...
</definitions>

```

Any expression defined in a query language is represented in BPMN by *FormalExpression* object. Such expression may be used to determine performers of a human task or to control flow conditions. In BPMN, a human task is an activity that is performed by one or more people called resources. Using *ResourceAssignmentExpression* object it is possible to determine task performers dynamically. This object has just one attribute: *expression* of *FormalExpression* type. This attribute may be used to specify a BPQL query that will be evaluated during process execution. For example, a participant assignment for “Confirm financial approval” human task can be defined in BPMN in the following way:

```

<userTask id="6" name="Confirm financial approval">
  <humanPerformer>
    <resourceAssignmentExpression>
      <formalExpression>
        GetBestPerformerByRoleWithExcludedActPerformer(
          'Travel.Reimbursement.FinancialApprover', '4')
      </formalExpression>
    </resourceAssignmentExpression>
  </humanPerformer>
</userTask>

```

### 5.2 Architecture

BPQL can be embedded into a BPM system. An architecture of a BPM system extended with BPQL is presented in Fig. 10, a diagram on the left.

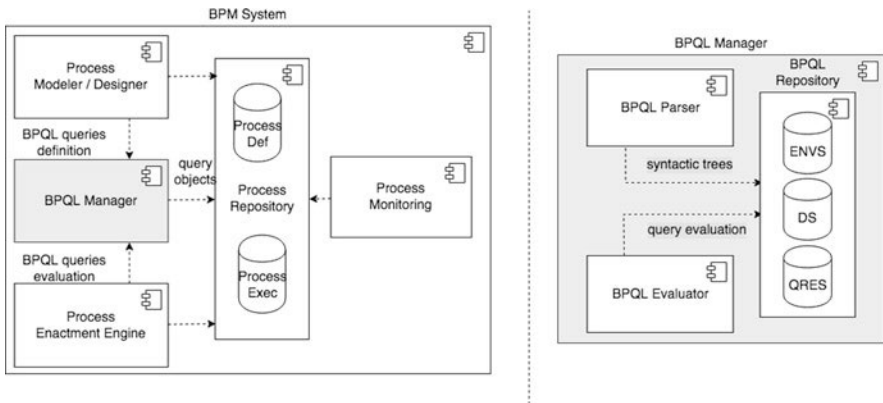


Fig. 10 BPM and BPQL architecture

There are two main use cases of BPQL Manager. In the first use case BPQL Manager provides data about BPQL queries in order to embed them in a process definition. In the second case, BPQL is invoked during process execution to evaluate a given BPQL query. In addition, BPQL Manager, during such evaluation, gets process definition and process execution objects from BPM repository.

A detailed architecture of BPQL Manager is presented in Fig. 10, a diagram on the right. BPQL Manager consists of BPQL Parser, BPQL Evaluator, and BPQL Repository. *BPQL Parser* is responsible for syntactic and (partially) semantic analysis of BPQL queries. During a process compilation (i.e., when a binary representation of a process is generated) Process Modeler/Designer sends all BPQL queries to BPQL Parser, which verifies them, generates their syntactic trees, and returns identifiers of the parsed queries. BPQL Parser stores parsed BPQL queries in DS part of BPQL Repository. *BPQL Evaluator* is responsible for evaluation of BPQL queries.

A request for evaluation of a query is sent by BPM Enactment Engine during process execution. Evaluation is made on the basis of the syntactic tree generated earlier by BPQL Parser. As a result, BPQL Evaluator returns a collection of values that satisfy the query. During evaluation, data stored in ENVIS and QRES are used. These data are managed by BPQL Repository. *BPQL Repository* is responsible for storing BPQL built-in operators and monitoring functions, the current values of ENVIS and QRES stacks, as well as the query evaluation context and other BPQL auxiliary data.

Finally, we discuss how BPQL fits into Process Querying Framework (PQF, [9]). We present how BPQL currently supports active components (Table 2) and passive components (Table 3).

## 6 Case Study

The first version of BPQL has been implemented for participant assignment and integrated in OfficeObjects@WorkFlow (OO WorkFlow) system. This system has been deployed at major Polish public institutions (e.g., Ministry of Labour and Social Policy and Ministry of Transport). First practical verification of BPQL in OO WorkFlow was done for the system of electronic document exchange between Poland and European Council (EWDP system [2]). In this system, OO WorkFlow was used to implement the process for preparation of the Polish standpoint concerning a given case that was discussed at the European Council, EU Committee of Permanent Representatives (COREPER) and COREPER working groups. The process consisted of more than 40 activities and included about 10 process roles. An extract from process execution history is presented in Fig. 11.<sup>6</sup> It was used by all

---

<sup>6</sup> Due to copyrights some elements of the application user interface were hidden.



**Table 2** PQF active components supported by BPQL

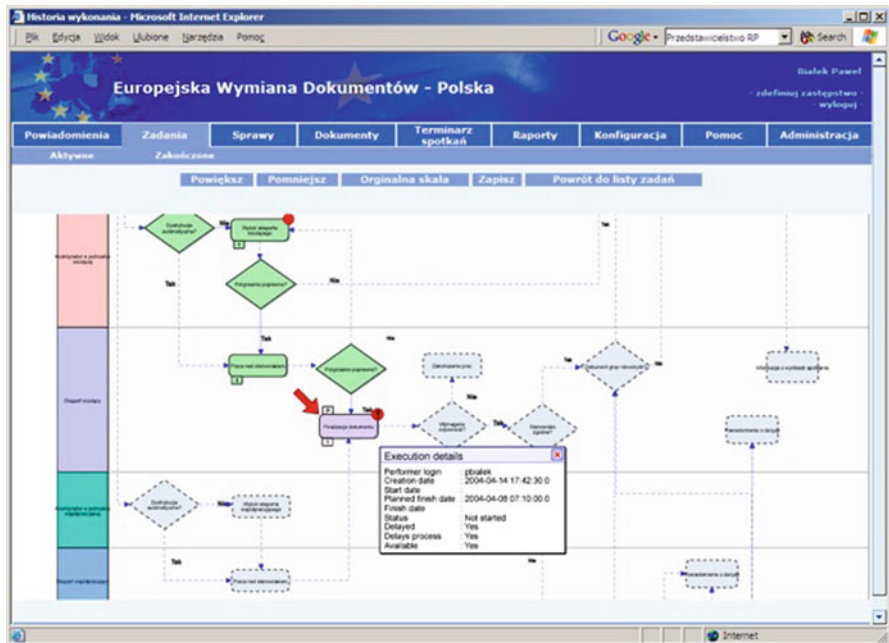
PQF component	BPM/BPQL component
Modeling	Process modeling is performed by Process Modeler/Designer (a part of a BPM system). Definition of the modeled processes are stored in BPM Process Repository. Elements of a business process definition, such as participant assignment, can be represented by BPQL queries
Simulating	In most BPM systems, simulation is a part of business process modeling and built in BPM Modeler/Designer. During simulation, BPQL queries are evaluated by BPQL Manager using data stored in Process Repository
Formalizing	BPQL Parser is responsible for parsing a BPQL query, verifying it, and generating its syntactic tree. The tree is stored in BPQL Repository while its identifier is kept in a process definition within BPM Process Repository
Recording	Process execution is performed by a Process Enactment Engine (a part of a BPM system). All the execution traces are stored in the BPM Process Repository. During process execution, the enactment engine invokes BPQL Manager to evaluate BPQL queries included in the process definition
Indexing	Indexing is a part of the BPM Process Repository where huge amount of data (esp. coming from process execution) is stored. So far, indexing is based on database indexing. BPQL can use indexing to more efficient retrieval of the business process objects
Filtering	All filtering is done by BPQL Evaluator on current basis during a BPQL query evaluation
Process Querying	BPQL Evaluator is responsible for BPQL query evaluation using data from BPM Process Repository (process definitions and history of process execution). All intermediate results during a query evaluation are stored in BPQL repository using DS, ENVS and QRES
Interpreting	Since the result of a BPQL query represents objects defined in the metamodel, those objects are self-describing. So far, BPQL does not provide any tool to analyze how a given query was evaluated

Polish Ministries and central offices with over 10,000 registered users. Daily, there were about 200 documents processed.

Originally, it was planned to have separate processes for every ministry, which could make in total more than twenty processes. These processes differed mainly in the rules to assign individual participants to the same or very similar activities. Using BPQL, it was possible to express the complex assignment rules and unify them. Eventually, EWDP system had just one, unified process for all ministries. Complex rules to assign participants were expressed in BPQL and used for selecting: (1) main coordinator who assigns Polish subjects to individual EU documents, (2) leading and supporting coordinators who assign experts to the processed document, and (3) leading and supporting experts. Participant assignment was based on roles, competences, current workload, process execution history, and availability of the experts. In addition, the pilot implementation phase for EWDP system was intensive and required many refinements of the process. All together, there were more than 80 versions of the process, and a significant part of those was related to updates in

**Table 3** PQF passive components supported in BPQL

PQF component	BPM/BPQL component
Process Model	Business process definitions are stored in BPM Process Repository. The structure of this data is compliant with the metamodel, process definition part
Event Log	Execution traces are stored in BPM Process Repository. The structure of this data is compliant with the metamodel, process execution part
Querying Instruction	BPQL Repository, Data Storage includes definition of BPQL Monitoring Functions and BPQL configuration
Process Query	BPQL Repository, Data Storage includes syntactic trees of BPQL queries
Process Repository	BPM Process Repository plays the role of Process Repository A. There is no need for Process Repository B since all the objects returned by a BPQL query evaluation are directly passed to BPM Enactment Engine. Local variables and intermediate results during query evaluation are stored in ENVS and QRES in BPQL Repository
Execution Plan	Execution plan for a BPQL query is compliant with traversing its syntactic tree. Such trees are stored in BPQL Repository, DS. More advanced execution plans are also possible but not implemented yet



**Fig. 11** A process execution history in EWDP system

process assignment. Thanks to BPQL these updates were made on-the-fly, without re-compiling the code.

## 7 Conclusion

In this chapter, we introduced BPQL together with the Business Process Metamodel. To understand the concept of BPQL, we described its syntax, explained its semantics, and provided several usage examples. We also showed how BPQL can be integrated in a BPM system and embedded in BPMN. Because of the size limit of this chapter, some parts of BPQL description had to be significantly reduced. More detailed description with further examples can be found in [5]. In addition, it is planned a demo project on GitHub (see [6]), which will include examples on how to use BPQL.

Summarizing the results of using BPQL, we observe that it is well suited to make business processes more flexible, especially in terms of dynamic participant assignment. We also believe that using BPQL makes a significant step forward to assure business process adaptability. BPQL can also be used to address other challenges in business process management, such as checking process correctness. However, we do not see immediate advantages of BPQL compared to the other languages dedicated for this particular purpose. There are still open issues in BPQL, such as efficient querying data sources outside of the metamodel, but we believe that flexibility, simplicity, and unambiguity of BPQL is worth a try and we encourage the readers to do that.

## References

1. Adamus, R., Habela, P., Kaczmarek, K., Lentner, M., Stencel, K., Subieta, K.: Stack-based architecture and stack-based query language. In: Object Databases, First International Conference, ICODDB 2008, Berlin, Germany, March 13–14, 2008. Proceedings, pp. 77–96 (2008)
2. Blizniuk, G., Momotko, M., Nowicki, B., Strychowski, J.: The EWD-P system: Polish government - Council of the European Union interoperability achieved. In: 38th Hawaii International Conference on System Sciences (HICSS-38 2005), CD-ROM / Abstracts Proceedings, 3–6 January 2005, Big Island, HI, USA (2005)
3. Cardoso, J.S., Sheth, A.P., Miller, J.A.: Workflow quality of service. In: Enterprise Inter- and Intra-Organizational Integration: Building International Consensus, IFIP TC5/WG5.12 International Conference on Enterprise Integration and Modeling Technique (ICEIMT'02), April 24–26, 2002, Valencia, Spain, pp. 303–311 (2002)
4. Eder, J., Panagos, E.: Managing time in workflow systems. In: Fischer, L. (ed.) Workflow Handbook 2001, pp. 109–132. Future Strategies Inc. (2001)
5. Momotko, M.: Tools for monitoring workflow processes to support dynamic workflow changes. Ph.D. thesis, Polish Academy of Sciences (2005)
6. Momotko, M.: BPQL demo. <https://github.com/mariusz-momotko/BPQL-demo> (2019)

7. Momotko, M., Subieta, K.: Process query language: A way to make workflow processes more flexible. In: *Advances in Databases and Information Systems, 8th East European Conference, ADBIS 2004, Budapest, Hungary, September 22–25, 2004, Proceeding*, pp. 306–321 (2004)
8. OMG: Business process model and notation (BPMN), version 2.0.2. Tech. rep., Object Management Group (2014). <http://www.omg.org/spec/BPMN/2.0.2>
9. Polyvyanyy, A., Ouyang, C., Barros, A., van der Aalst, W.M.P.: Process querying: Enabling business intelligence through query-based process analytics. *Decis. Support Syst.* **100**, 41–56 (2017)
10. Sadiq, S.W.: Handling dynamic schema change in process models. In: *Australasian Database Conference, ADC 2000, Canberra, Australia, January 31–February 3, 2000*, pp. 120–126 (2000)
11. Subieta, K.: Stack-based query language. In: *Encyclopedia of Database Systems, 2nd edn.* (2018)
12. Subieta, K., Beeri, C., Matthes, F., Schmidt, J.W.: A stack-based approach to query languages. In: *East/West Database Workshop, Proceedings of the Second International East/West Database Workshop, Klagenfurt, Austria, 25–28 September 1994*, pp. 159–180 (1994)
13. Syntax and semantics of the stack-based query language (SBQL) (2010). <http://www.ipipan.waw.pl/~subieta/ExtendedWorkflow>
14. van der Aalst, W.M.P.: Generic workflow models: How to handle dynamic change and capture management information? In: *Proceedings of the Fourth IFCIS International Conference on Cooperative Information Systems, Edinburgh, Scotland, USA, September 2–4, 1999*, pp. 115–126 (1999)
15. Weske, M., Vossen, G.: Flexibility and cooperation in workflow management systems. In: *Handbook on Architectures of Information Systems*. Springer (1998)
16. Yang, Y., Dumas, M., García-Bañuelos, L., Polyvyanyy, A., Zhang, L.: Generalized aggregate quality of service computation for composite services. *J. Syst. Softw.* **85**(8), 1818–1830 (2012)

# Celonis PQL: A Query Language for Process Mining



Thomas Vogelgesang, Jessica Ambrosy, David Becher, Robert Seilbeck, Jerome Geyer-Klingenberg, and Martin Klenk

**Abstract** Process mining studies data-driven methods to discover, enhance, and monitor business processes by gathering knowledge from event logs recorded by modern IT systems. To gain valuable process insights, it is essential for process mining users to formalize their process questions as executable queries. For this purpose, we present the Celonis Process Query Language (Celonis PQL), which is a domain-specific language tailored toward a special process data model and designed for business users. It translates process-related business questions into queries and executes them on a custom-built query engine. Celonis PQL covers a broad set of more than 150 operators, ranging from process-specific functions to machine learning and mathematical operators. Its syntax is inspired by SQL, but specialized for process-related queries. In addition, we present practical use cases and real-world applications, which demonstrate the expressiveness of the language and how business users can apply it to discover, enhance, and monitor business processes. The maturity and feasibility of Celonis PQL is shown by thousands of users from different industries, who apply it to various process types and huge amounts of event data every day.

## 1 Introduction

Process mining is a data-driven approach to reconstruct, analyze, and improve business processes using log data recorded by modern IT systems, like enterprise resource planning (ERP) or customer relationship management (CRM) systems [15]. The starting point of process mining is an event log, which is a record of what happens when a business process is performed. Process mining applies special algorithms on the event log data to uncover the actual process execution, to reveal undesired process behavior and to identify inefficiencies [15]. Typical business use

---

T. Vogelgesang · J. Ambrosy · D. Becher · R. Seilbeck · J. Geyer-Klingenberg (✉) · M. Klenk  
Celonis SE, Munich, Germany  
e-mail: [j.geyerklingsberg@celonis.com](mailto:j.geyerklingsberg@celonis.com)

cases for process mining are procurement (e.g., purchase-to-pay), sales (e.g., order-to-cash), accounting (e.g., accounts payable), or production (e.g., make-to-order).

Due to their strong ability to provide transparency across complex business processes, process mining capabilities have been adopted by many software vendors and academic tools. A key success factor for any process mining tool is the ability to translate business questions into executable process queries and to make the query results accessible to the user. To this end, we developed Celonis Process Query Language (Celonis PQL). It takes the input from the user and executes the queries in a custom-built query engine. This allows the users to analyze all facets of a business process in detail, as well as to detect and employ process improvements. Celonis PQL is a comprehensive query language that consists of more than 150 (process) operators. The language design is strongly inspired by the requirements of business users. Therefore, Celonis PQL achieved a wide adoption by thousands of users across various industries and process types.

This chapter is organized as follows: Sect. 2 provides the background knowledge, which is required to understand the specifics of our process query language. Section 3 gives an overview of the various application scenarios of Celonis PQL. Section 4 presents the query language, its syntax, and operators. Section 5 demonstrates the applicability of Celonis PQL to solve widespread business problems. Section 6 outlines the implementation of the query language. Section 7 positions Celonis PQL within the Process Querying Framework (PQF). Finally, Sect. 8 concludes the chapter.

## 2 Background

In this section, we introduce the general concept of process mining and how the Celonis software architecture enables process mining through our query language. We also present the history of Celonis PQL as well as the design goals that were considered during the development of the query language.

### 2.1 *Process Mining*

In the course of digitization, an increasing number of log data is recorded in IT systems of companies worldwide. This data is of high value, as it represents how processes are running inside a company. Process mining technology can be applied to such data to gain insights about business processes, discover inefficiencies, and find potential improvements.

CASE	ACTIVITY	TIMESTAMP	DEPARTMENT	ITEM
1	Create Purchase Order Item	2019-01-23 08:15	D1	Screw
1	Request Approval	2019-01-23 08:20	D1	Screw
1	Grant Approval	2019-01-23 11:00	D2	Screw
1	Send Purchase Order	2019-01-23 11:10	D1	Screw
1	Receive Goods	2019-01-25 10:30	D3	Screw
1	Scan Invoice	2019-01-25 11:30	D3	Screw
1	Clear Invoice	2019-01-28 17:15	D3	Screw
2	Create Purchase Order Item	2019-01-23 13:00	D1	Screw Driver
2	Request Approval	2019-01-23 15:00	D1	Screw Driver
2	Reject	2019-01-23 18:00	D2	Screw Driver

Fig. 1 Example event log

Process mining is based on *event logs*. An event log is a collection of *events*. An event is described by a number of attributes. The following three event attributes are always required for process mining:

**Case.** The case attribute indicates which process instance the event belongs to. A process instance is called a *case*. A case usually consists of multiple events.

**Activity.** The activity attribute describes the action that is captured by the event.

**Timestamp.** The timestamp indicates the time when the event took place.

A sequence of events, ordered by their timestamps, that belong to the same case is called a *trace*. The traces of all the different cases with the same activity sequence represent a *variant*. The *throughput time* between two events of a case is the time difference between the corresponding timestamps. Accordingly, the throughput time of a case is equal to the throughput time between the first and the last event of the corresponding trace.

Figure 1 shows an example event log in procurement. Each case represents a process instance of one purchase order. In the first case, the order item is created in the system, an approval for purchasing is requested and approved. After the approval, the order is sent to the vendor. Two days later, the ordered goods are received, the invoice is registered and eventually paid. In the second case, an order item is created, but the approval to actually order it is rejected. Besides the three required attributes of an event log mentioned above, the example also includes attribute DEPARTMENT, which specifies the executing department for each event, as well as attribute ITEM containing the description of the corresponding order item.

Process mining techniques that are applied on an event log to understand and improve the corresponding process can be assigned to three groups [15]: discovery, conformance, and enhancement. Discovery uses the event log as input and generates a business process model as output. Conformance takes the event log and an a priori process model to detect discrepancies between the log data and the a priori model. Enhancement takes the event log and an a priori model to improve the model with the insights generated from the event log.

## 2.2 Architecture Overview

Celonis PQL is an integral component of the Celonis software architecture, which is shown in Fig. 2. All Celonis applications use this language to query data from a *data model*. The data model contains metadata like schema information and the foreign key relationships between the tables, as well as the actual data from the source systems. Celonis PQL queries are evaluated by the Celonis PQL Engine.

**Source system.** A source system is the system containing the business data to be analyzed by the Celonis applications. ERP systems like SAP, CRM systems like Salesforce, and many other standard systems are supported. Celonis applications can also connect to a variety of database systems on the customer’s premises like PostgreSQL. It is also possible to upload Excel or CSV files. Data from multiple source systems can be combined in one data model.

**Data model.** A data model combines all tables from the source system (or multiple source systems), which contain the data about a process that a user wants to analyze. In the data model, the foreign key relationships between the source tables can be defined. This is performed here because specifying joins is not part of the query language itself. The tables are arranged in a snowflake schema, which is common for data warehouses, and the schema is centered around explicit case and activity tables. Other data tables provide additional context. Figure 3 shows an example data model. It contains the event log of Fig. 1 in the ACTIVITIES table, including the DEPARTMENT column. It is linked to the CASES table, containing information about each order item. The ITEM

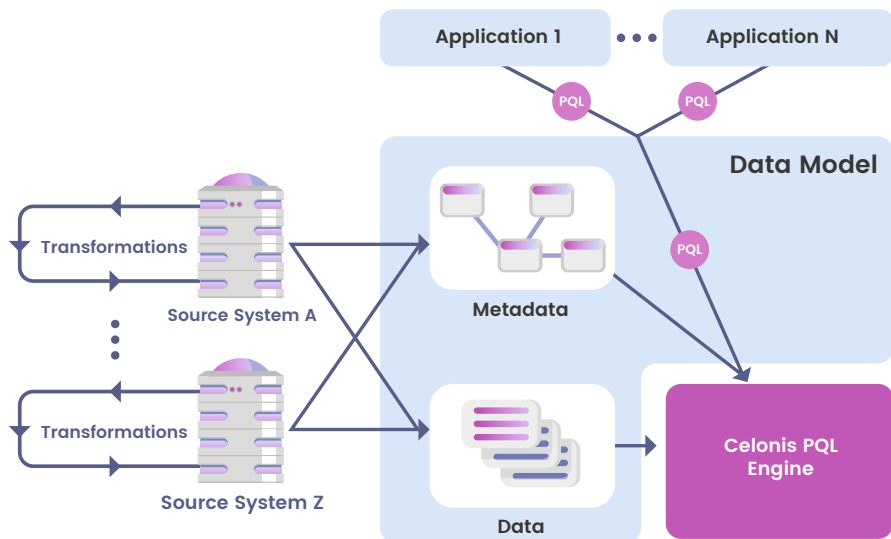


Fig. 2 Celonis architecture overview



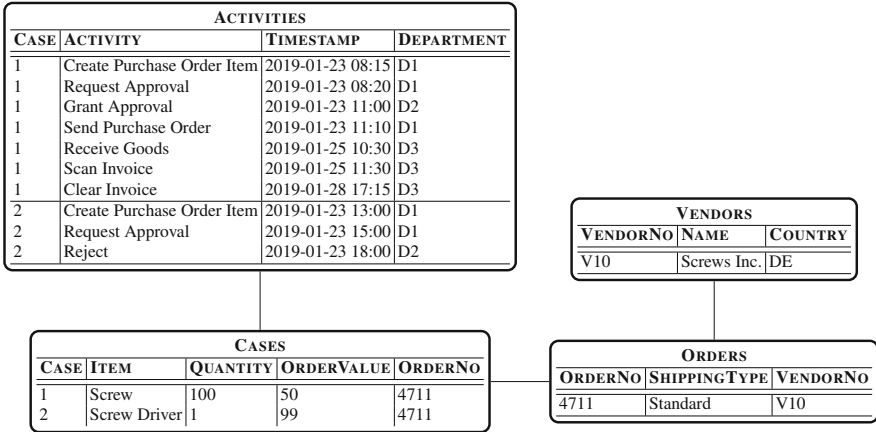


Fig. 3 Example data model with four tables, including activity and case tables

attribute from the example event log of Fig. 1 is contained in the CASES table, as the data model should contain normalized table schemas. Both order items (i.e., both cases) belong to the same purchase order, which is sent to one vendor. Details about the purchase orders and the vendors are available in the ORDERS and VENDORS tables of the data model.

**Activity table.** The data model always contains an event log, which we call the *activity table*. The activity table always contains the three columns of the core event log attributes, while additional columns may be present. Within one case, the corresponding rows in the activity table are always sorted based on the timestamp column.

Usually, the activity table is not directly present in the source systems and therefore needs to be generated depending on the business process being analyzed. Since the source system is a relational database in most cases, this is usually done in SQL in the so-called transformation step. The transformation result can be a database view. However, a persisted table is usually created for performance reasons. This procedure is comparable to the extract, transform, load procedure (ETL) in data warehouses. Like all the other tables, the resulting activity table is then imported into the data model. The user can specify the case and activity table in a graphical user interface (GUI) and mark the corresponding columns of the activity table as the case, activity, and timestamp columns.

**Case table.** The case table provides information about each case and acts as the fact table in the snowflake schema. It always includes the case column, containing all distinct case IDs, while other columns provide additional information on the cases. There is a 1:N relationship between the case table and the activity table. If the case table is not specified in the data model, it will be generated automatically during the data model load. The case table then consists of one column containing all distinct case IDs from the activity table. This guarantees

that a case table always exists, and the Celonis PQL functions and operators can rely on it.

**Celonis PQL Engine.** Celonis PQL Engine is an analytical column-store main memory database system. It evaluates Celonis PQL queries over a defined data model. Section 6 describes the Celonis PQL Engine in more detail.

**Applications.** Celonis applications provide a variety of tools for the business user to discover, monitor, and enhance business processes. All applications use Celonis PQL to query the required data. They include easy-to-use GUIs, providing a convenient way for the users to interact with the process and business data. In the applications, the users can specify custom Celonis PQL queries. There are also many auto-generated queries sent by the applications to retrieve various information, which is then presented to the user in the graphical interface. An overview of the different applications that use Celonis PQL is given in Sect. 3.

### 2.3 History of Celonis PQL

The first version of Celonis PQL was introduced with version 2.4 of Celonis Process Mining, which was released in 2014. Celonis PQL was an extension to SQL, providing commands to query and filter process flows and patterns in addition to the standard SQL commands. For example, the process filter `process# START 'Activity XYZ'` could be used to filter all cases that start with activity “Activity XYZ”. The custom Celonis PQL commands were translated into standard SQL in the background and executed directly on the source database system. Multiple source systems like SAP HANA, MSSQL, and Oracle were supported.

In SQL, different database systems support different SQL dialects. For example, function names and syntaxes for certain functionalities like date and time calculations are database specific. However, Celonis PQL was independent of the SQL dialect of the underlying database system. If necessary, Celonis PQL functions were mapped to equivalent SQL functions based on the appropriate dialect.

For version 4.0 of Celonis Process Mining, which was released in February 2016, the concept of Celonis PQL was fully redesigned based on the experiences gained from the first version of the language. Instead of being an extension to SQL, Celonis PQL became an independent language inspired by SQL. Now, Celonis PQL queries are executed on the custom-built Celonis PQL Engine, which is a query engine that is highly optimized for process mining capabilities. In comparison to the previous approach, this enables much better performance. Also, all functionalities are fully independent of the underlying database dialect, which makes it easier to support a wider range of source systems. Several patents were registered as part of the development efforts.

In October 2018, Celonis Intelligent Business Cloud (IBC) was released. This transition from an on-premises product to a modern native cloud solution provides easier access to process mining and, consequently, IBC increased the number of

Celonis PQL users significantly. Many new applications, all using Celonis PQL to query process data, are included in IBC, as described in Sect. 3.

The language is continuously extended with new functionalities. This is mostly driven by customers who use Celonis PQL on a daily basis to explore their data. Due to the rich functionality and possibility to use it in various Celonis applications, Celonis PQL is used by a high number of users in many production systems.

## 2.4 Design Goals

The first version of Celonis PQL was an extension to SQL. While it had several convenient process mining functions, the actual queries evaluated on the database were complicated. Furthermore, it was difficult to extend the language with new functionality. To overcome these issues, the query language was redesigned based on previous experiences, with the following design goals in mind:

**Simplicity.** The query language should be easy to use for business users. Providing an easy way to translate complex process questions into data queries should make process mining accessible for business users.

**Flexibility.** The query language should not include specialized functions. Instead, the goal is to provide a set of generic functions and operators that can be combined in a wide range of queries. This flexibility is very important, since the users should be able to formulate all their questions in the query language, regardless of the processes they address.

**Event log-centered.** In contrast to SQL, the language should be designed to support dedicated process mining functionality. This should be reflected in the query language by process functions, which operate on the given event log.

**Business focus.** Event data can be augmented with additional business information. It is therefore important to combine process mining and business intelligence (BI) capabilities within one query language. To achieve this, besides specific process mining functionality, the query language should also provide a variety of functions known from SQL, like aggregations, string modifications, and mathematical functions.

**Frontend interaction.** To simplify the use of the query language, the user should be able to formulate queries with support of a GUI. Consequently, the goal is to design a language that provides easy integration via GUI components. The simple query creation using a GUI is a key factor for the usability of a product, which results in high acceptance, usage, and adoption by the users.

### 3 Applications

As a result of emerging technologies, the requirements on tools used for analyzing processes within different business departments go beyond the simple tracking of performance. For this reason, process mining at Celonis is evolving into a holistic approach that serves as a performance accelerator for business processes. Necessary steps that are included within this approach are the discovery, enhancement, and monitoring of processes. Within discovery, process mining can capture digital footprints from all source systems involved in the process, visualize the respective processes, and understand the root causes of deviations between the as-is process and the to-be process. Thus, the discovery step serves as the starting point for process improvements. During enhancement, process mining supports the automation of tasks, proposes intelligent actions, and proactively drives process interventions and improvements. Monitoring allows the user to continuously track the development of key figures that are defined during the discovery step. This enables the ongoing benchmarking of processes—internally, as well as externally. Celonis PQL enables all these activities and tasks, and it is used in all Celonis products as depicted in Fig. 4.

- **Process Analytics** is part of discovery and can be used to visualize and identify the root causes of issues within a process. Furthermore, it identifies the specific actions that have the greatest impact on solving the issue. For that purpose, Celonis PQL is used to obtain performance metrics, such as the average case duration, the change rate in the process, or the degree of process automation. In addition, Celonis PQL enables the user to enrich the event log data with data related to the problematic cases, such as finding the vendor causing the issue, or identifying sub-processes that prolong throughput times.

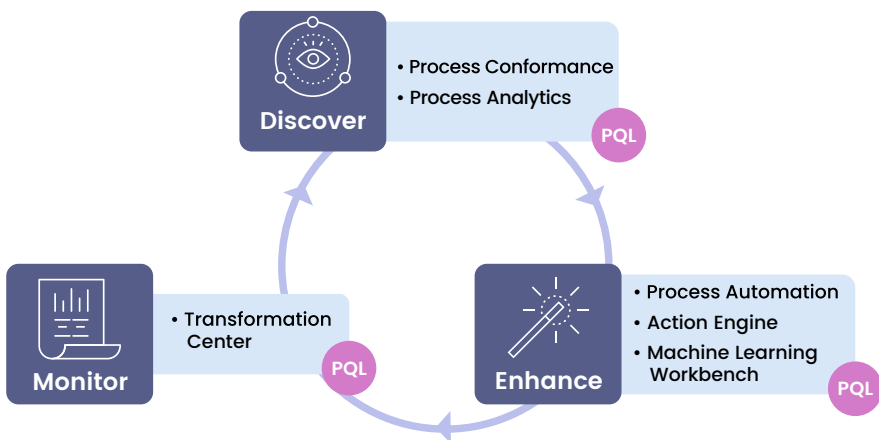


Fig. 4 The Discover, Enhance, Monitor approach

- **Process Conformance** is, as shown in Fig. 4, also part of the discovery step. It is used to identify deviations from the defined to-be process and to uncover the root causes of process deviations. In this context, Celonis PQL allows utilizing the calculated process conformance to obtain metrics leading to the discovery of root causes for deviations [16].
- **Action Engine** is part of the enhancement step and uses insights gained from the discovery step to recommend actions to improve process performance [1]. The foundation to generate these findings is Celonis PQL. The findings can include all the relevant information that impacts a decision about whether or how an action, like executing a task in the source system or triggering a bot, is performed. In addition, Celonis PQL assists in prioritizing necessary actions.
- **Machine Learning Workbench** enables the usage of Jupyter notebooks within Celonis IBC for creating and using Python predictive models. As part of the enhancement step, it supports building predictive models on process data to proactively avoid downstream friction like long running process steps, leading to e.g., late payments in finance. Celonis PQL is crucial for querying the process data necessary as input for the predictive models.
- **Process Automation** allows to automatically trigger specific actions in downstream applications, like SAP, Gmail, and Salesforce, based on predefined process conditions. Therefore, Process Automation is part of the enhancement step. In this context, Celonis PQL is used for process condition querying.
- **Transformation Center** is part of the monitoring step. It measures and monitors the progress of the process metrics defined within the discovery step. Therefore, it assists in achieving Key Performance Indicators (KPIs) and business outcomes. As with Process Analytics, Celonis PQL is used in this context to obtain the performance metrics by querying the underlying event log data and to enrich event log data with relevant business context.

Celonis IBC enables customers to use the entire product range on-demand, containing all products described above. As shown in Fig. 5, Celonis IBC is currently (as of October 2019) managing more than 10,000 users and more than

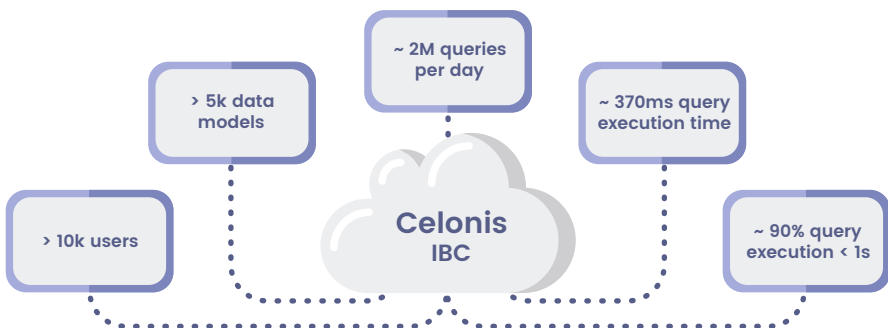


Fig. 5 Celonis IBC statistics (as of October 2019)

5000 data models. Two million queries are executed per day, written in Celonis PQL, and processed by the Celonis PQL Engine with an average execution time of 370 milliseconds per query. In reality, the execution time per query is often much lower, but especially complex Celonis PQL statements on huge datasets lead to outliers in execution time. In around 90% of the cases, the execution time is less than a second. Nevertheless, we still aim for continuous performance improvements in order to further reduce the execution time of complex Celonis PQL statements, like heavily nested queries, in the future.

## 4 The Celonis Process Query Language

The intention of Celonis PQL is to provide a query language for performing process mining tasks on large amounts of event data. As described in Sect. 2.2, it is based on a relational data model. The event and business data as well as all results (including the mined process models) are represented as relational data.

Currently, the supported data types comprise `STRING`, `INT`, `FLOAT`, and `DATE`. Boolean values are not directly supported, but can be represented as integers. Each data type can hold `NULL` values. In general, Celonis PQL treats `NULL` values as non-existing and ignores them in aggregations. Also, row-wise operations like adding the values of two columns will return `NULL` if one of its inputs is `NULL`.

Operators usually create and return a single column that is either added to an existing table (e.g., the case or activity table) or to a new, temporary result table. Only a few operators (e.g., for computing a process graph) create and return one or more tables with multiple columns. However, these operators are only used internally by GUI components and are not exposed to the end-user.

Currently, Celonis PQL provides more than 150 different operators to process the event data. Due to space limitations, we cannot sketch the full language. However, we can offer a brief overview of the major language features before we present selected examples to showcase the expressiveness of the language. Comprehensive documentation of the Celonis PQL operators can be accessed via the free process mining platform Celonis Snap.<sup>1</sup>

### 4.1 Language Overview

Even though Celonis PQL is inspired by SQL, there are major differences between the two query languages. Figure 6 shows these differences by comparing how to query the cases and the number of involved departments for all orders with a value

---

<sup>1</sup> <https://www.celonis.com/snap-signup/> (registration required).

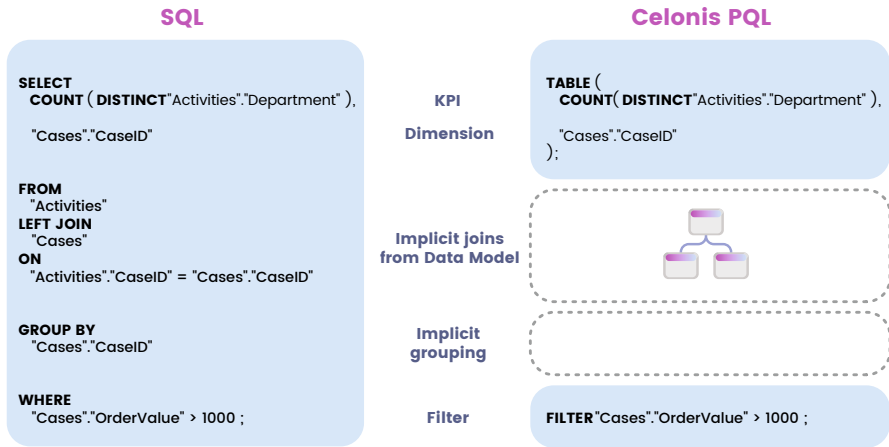


Fig. 6 Comparing SQL and Celonis PQL by an example query

of more than 1000 euros in both languages. Furthermore, it also illustrates the key concepts of Celonis PQL.

Similar to SQL, Celonis PQL enables the user to specify the data columns to retrieve from the data model. This can either be an aggregation, which we call a KPI, or an unaggregated column, which we call a dimension. While the data columns are part of the SELECT statement in SQL, Celonis PQL requires them to be wrapped in the TABLE operator, which combines the specified columns into a common table.

In contrast to SQL, Celonis PQL does not require the user to define how to join the different tables within the query. Instead, it implicitly joins the tables according to their foreign key relationships that have to be defined only once in the data model. Also, the grouping clause is not needed in Celonis PQL as each selected column, which is not aggregated (i.e., a dimension), is implicitly used as a grouper. According to the design goals, implicit joins and groupings significantly reduce the size and complexity of the queries and make it much simpler to formulate them.

Both languages offer the possibility to filter rows. While SQL requires the user to formulate the filter condition in the WHERE clause of the query, Celonis PQL offers the FILTER statements that are separated from the TABLE statements but executed together. Splitting the data selection and the filters into different statements enables the user to define multiple filter statements in different locations inside an application, which then can be combined into the table statement to query the data.

Beyond this simple structure, Celonis PQL provides a wide range of different operators that can be combined to answer complex business questions. The following list gives an overview of the most important classes of operators.

**Aggregations.** Celonis PQL offers a wide range of aggregation functions, from simple standard functions like count and average, to more advanced aggregations like standard deviation and quantiles. Most of the aggregation functions are also

available as window-based functions computing the aggregation not over all values but over a user-defined sliding element window.

**Data functions.** These are operators like `REMAP_VALUES` (see Sect. 4.2) and `CASE WHEN` (see Sect. 5.1), which allow for conditional changes of values.

**Date and time functions.** These functions enable the user to modify, project, or round a date or time value, e.g., add a day to a date or extract the month from a timestamp. There are also functions to compute date and time difference (e.g., between timestamps of events).

**Index functions.** Index functions create indices based on columns. The function `INDEX_ACTIVITY_LOOP`, for example, returns for each activity how many times it has occurred in direct succession in a case. This is useful, e.g., for identifying self-loops and computing their cycle lengths.

**Machine learning functions.** There are various machine learning functions available, e.g., to cluster data using the k-means algorithm or learn decision trees.

**Math functions.** Celonis PQL offers a wide range of mathematical functions, e.g., for arithmetic computations, rounding float numbers, and computing logarithms.

**Predicates and logical operators.** For expressing complex filter conditions, Celonis PQL offers a variety of predicates (like range-based or pattern-based comparison) and standard Boolean operators (`AND`, `OR`, and `NOT`).

**Process functions.** Process functions comprise all process-specific functions that operate on the activity table and take its configuration into account. Examples are pattern-based process filters, `SOURCE` and `TARGET` operators (see Sect. 4.2), and computation of variants (see Sect. 4.3). There are also special process mining operators for discovering process models, clustering variants, and checking the conformance of a process model to the event data (see Sect. 4.4).

**String modification.** These functions enable the user to modify string values, e.g., trimming whitespaces, changing case, and creating substrings.

A major difference between SQL and Celonis PQL is the different language scope. Hence, Celonis PQL does not support all operators that are available in SQL. This is due to the fact that the development of the language is driven by customer requirements, and only operators that are needed for the target use cases are implemented. For example, generic set operators like `UNION` are not supported, as they have not been required so far.

Another major difference to SQL is the missing support of a data manipulation language (DML). As all updates in the process mining scenario should come from the source systems, there is no need to directly manipulate and update the data through the query language. As the data can be considered to be read-only, this also allows for specific performance optimizations during implementation (see Sect. 6).

Furthermore, Celonis PQL does not provide any data definition language (DDL). As the data model is created by a visual data model editor and stored internally, there has not been any need for this so far.

In contrast to SQL, Celonis PQL is domain-specific and offers a wide range of process mining operators that are not available in SQL. Consequently, Celonis PQL



seamlessly integrates the data with the process perspective. In the following, we explain selected process operators like SOURCE and TARGET (Sect. 4.2), VARIANT (Sect. 4.3), and CONFORMANCE (Sect. 4.4) in more detail.

### 4.2 Source and Target Operators

In process mining applications it is often required to relate an event to another event that directly or eventually follows. For instance, this is required to compute the throughput time between two events by calculating the difference between the corresponding timestamps. Due to the relational data model, the timestamp values to subtract are stored in different rows. However, the operators (e.g., arithmetic operations) usually can only combine values from the same row. Therefore, we need a way to combine values from two different rows into the same row for performing such computations.

To overcome this issue, Celonis PQL relies on the SOURCE and TARGET operators. Figure 7 shows an example that illustrates how SOURCE and TARGET can be used to compute the throughput time between an event and its direct successor. While SOURCE always refers to the actual event, TARGET refers to its following event. Consequently, SOURCE and TARGET can be used to combine an event with its following event in the same row of a table. Both operators accept a column of the activity table as input and return the respective value of the referred event, as illustrated in Fig. 7.

For the first event in the ACTIVITIES table, SOURCE returns the activity name “A” of the current event, while TARGET returns the activity name “B” of the following event (refer to ① in Fig. 7). For the second event of the input table, SOURCE returns

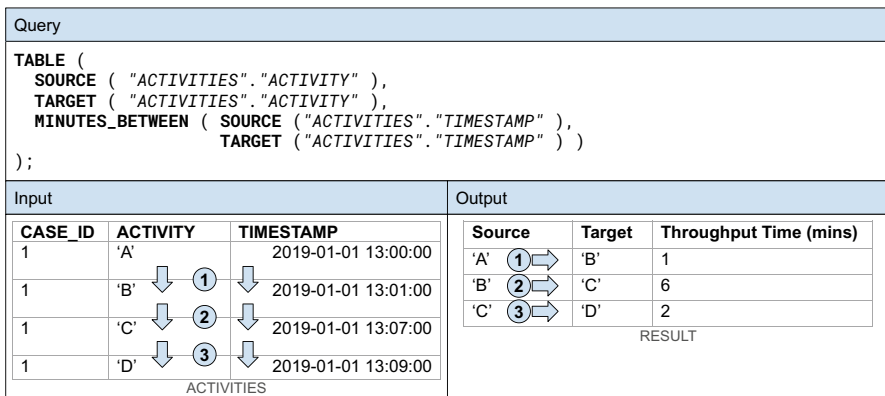


Fig. 7 Example of throughput time computation using SOURCE and TARGET operators

“B” and TARGET returns “C” (refer to ② in Fig. 7) while they return “C” and “D” for the third event (refer to ③ in Fig. 7).

The example also demonstrates how the SOURCE and TARGET operators can be used to compute the throughput time. Instead of the activity column, we can use the column containing the timestamp of the events as input. Consequently, SOURCE and TARGET return the timestamps of the referred events. Then, we can pass the result columns of the SOURCE and TARGET operators to the MINUTES\_BETWEEN operator to compute the difference between the timestamps of an event and its following event in minutes. In the example of Fig. 7, this results in throughput times of 1 minute from “A” to “B”, 6 minutes from “B” to “C”, and 2 minutes from “C” to “D”.

---

### Syntax 1 SOURCE and TARGET operators

---

```
SOURCE ( input_column [, filter_column ] [, edge_config ] )
TARGET ( input_column [, filter_column ] [, edge_config ] )

edge_config ← ANY_OCCURRENCE[] TO ANY_OCCURRENCE[]
              FIRST_OCCURRENCE[] TO ANY_OCCURRENCE[]
              FIRST_OCCURRENCE[] TO ANY_OCCURRENCE_WITH_SELF[]
              ANY_OCCURRENCE[] TO LAST_OCCURRENCE[]
              FIRST_OCCURRENCE[] TO LAST_OCCURRENCE[]
```

---

Syntax 1 shows the syntax of the SOURCE and TARGET operators, which is similar for both operators. The first parameter is a column of the activity table. Its values are mapped to the referred events and returned as result column. The result column is stored in a temporary result table that can be joined with the case table.

To skip certain events, the SOURCE and TARGET operators accept an optional filter column as a parameter. This column must be of the same size as the activity table. The SOURCE and TARGET operators ignore all events that have a NULL value in the related entry of the filter column. Usually, the filter column is created using the REMAP\_VALUES operator.

---

### Syntax 2 REMAP\_VALUES operator

---

```
REMAP_VALUES ( input_column ( , [ string , string ] )+ [, other_value ] )
REMAP_INTS ( input_column ( , [ integer , integer ] )+ [, other_value ] )
```

---

The syntax of the REMAP\_VALUES operator is shown in Syntax 2. The first parameter is an input column of type string that provides the values that should be remapped as input. For creating a filter column for the SOURCE and TARGET operators, this input column is usually the activity column of the activity table. However, REMAP\_VALUES can be generally applied to any column of type string. The second parameter is a list of one or more pairs of string values that describe the

mapping. Each occurrence of the first value of the pair will be remapped to the second value of the pair. Finally, the operator accepts an optional string value that will replace all values that are not remapped within the mapping. If this optional default replacement value is missing, all values not considered in the mapping will remain unchanged. As the REMAP\_VALUES operator is only applicable to columns of type string, REMAP\_INTS provides a similar functionality for columns of type integer.

Figure 8 shows a simple example of the REMAP\_VALUES operator. It takes the activity column of the activity table as input and maps “B” and “C” to NULL. As the optional replacement value is not defined, all the other values (“A” and “D”) remain the same. Figure 9 demonstrates how to use the result of the REMAP\_VALUES as filter column for the SOURCE and TARGET operators by an example query.

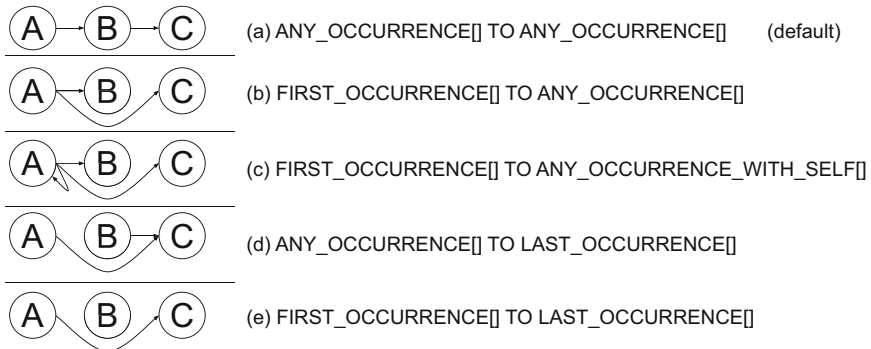
The query returns the activity names of the source and target events given in the input table ACTIVITIES. However, the RESULT table only shows one row relating “A” to “D” because the activities “B” and “C” are filtered out. This is achieved by passing the result of the REMAP\_VALUES operator as shown in Fig. 8 to the SOURCE operator as filter column. As both activities “B” and “C” are mapped

Query			
<pre>TABLE (   REMAP_VALUES ( "ACTIVITIES"."ACTIVITY", [ 'B', NULL ], [ 'C', NULL ] ) ) );</pre>			
Input			Output
<b>CASE_ID</b>	<b>ACTIVITY</b>	<b>TIMESTAMP</b>	<b>Remapped Values</b>
1	'A'	2019-01-01 13:00:00	'A'
1	'B'	2019-01-01 13:01:00	NULL
1	'C'	2019-01-01 13:07:00	NULL
1	'D'	2019-01-01 13:09:00	'D'
ACTIVITIES			RESULT

Fig. 8 Example of REMAP\_VALUES operator

Query			
<pre>TABLE (   SOURCE ( "ACTIVITIES"."ACTIVITY",     REMAP_VALUES ( "ACTIVITIES"."ACTIVITY", [ 'B', NULL ], [ 'C', NULL ] ) ),   TARGET ( "ACTIVITIES"."ACTIVITY" ),   MINUTES_BETWEEN ( SOURCE ( "ACTIVITIES"."TIMESTAMP" ),     TARGET ( "ACTIVITIES"."TIMESTAMP" ) ) );</pre>			
Input			Output
<b>CASE_ID</b>	<b>ACTIVITY</b>	<b>TIMESTAMP</b>	<b>Source</b> <b>Target</b> <b>Throughput Time (mins)</b>
1	'A'	2019-01-01 13:00:00	'A'    'D'    9
1	'B'	2019-01-01 13:01:00	
1	'C'	2019-01-01 13:07:00	
1	'D'	2019-01-01 13:09:00	
ACTIVITIES			RESULT

Fig. 9 Example for omitting activities “B” and “C” in SOURCE and TARGET operators



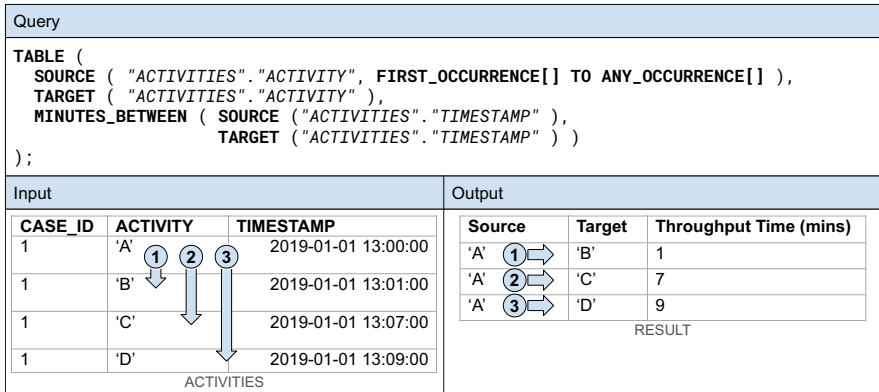
**Fig. 10** Available edge configuration options of the SOURCE and TARGET operators

to NULL, the next subsequent activity of “A” is “D” with a throughput time of 9 minutes.

To define which relationships between the events should be considered, the operators offer the optional edge configuration parameter. Figure 10 illustrates the different edge configuration options. The first option (a) is the default and only considers the direct follow relationships between the events, while option (b) only considers relationships from the first event to all subsequent events. Option (c) is similar to option (b) but also considers self-loops of the first event. Option (d) is the opposite of option (b) and only considers relationships going from any event to the last event. Finally, option (e) only considers the relationship between the first and the last event. The different options enable the user to compute KPIs between different activities of the process. For example, you can use option (b) to compute how many minutes after the start of the process (indicated by the first activity “A”) an activity was executed. This is illustrated in Fig. 11 where SOURCE always refers to the first event of the case (activity “A”) while TARGET refers to any other event (activities “B”, “C”, and “D”). Consequently, MINUTES\_BETWEEN computes the minutes elapsed between the occurrence of “A” and all the other activities of the case. For computing the remaining process execution time for each activity of the process, you can simply adapt the edge configuration in the query from Fig. 11 to option (d).

To simplify the query, the optional edge configuration and the filter column need to be defined in only one occurrence of SOURCE or TARGET per query. The settings are implicitly propagated to all other operators in the same query. This can be seen in the query in Fig. 9, where the TARGET operator inherits the filter column from the SOURCE operator.

Besides the computation of custom process KPIs, like the throughput time between certain activities, SOURCE and TARGET also enable more advanced use cases, like the segregation of duties, as we will demonstrate in Sect. 5.3. A concept similar to the SOURCE and TARGET operators has recently been proposed in [3].



**Fig. 11** Example for computing how many minutes after the start of the process an activity was executed

### 4.3 Variant Computation

The computation of variants is a vital task in process mining. Most process discovery algorithms, like the Inductive Miner [8] or the Heuristics Miner [17], use them as input instead of the raw events and cases to significantly speed up the computation. To compute variants, Celonis PQL provides the `VARIANT` operator that aggregates all events of a case into a string, which represents the variant of the case. The resulting column is added to the case table such that each case is related to its respective variant.

---

**Syntax 3** `VARIANT` operator and `VARIANT` operator with reduced self-loops

---

`VARIANT` ( *input\_column* )

`SHORTENED` ( `VARIANT` ( *input\_column* ) [ , *max\_cycle\_length* ] )

---

The syntax of the `VARIANT` operator is shown in Syntax 3. As input, the operator uses a column of type string from the activity table. The operator concatenates the string values of the given column into a single string delimited by comma and adds the result to the row of the related case. Usually, the activity column is used as input; however, other columns of the activity table, like the name of the executing department or user, can be used.

Sometimes, different cases may have self-loops of the same activity but with a different number of activities. Consequently, these cases are related to different variants. However, in some applications it is not of interest how often an activity is repeated but only if there is a self-loop or not. For such cases, the `VARIANT`

Query																																																												
<pre>TABLE (   "CASES"."CASE_ID",   VARIANT ( "ACTIVITIES"."ACTIVITY" ),   SHORTENED ( VARIANT ( "ACTIVITIES"."ACTIVITY" ) ) );</pre>																																																												
Input		Output																																																										
<table border="1"> <thead> <tr> <th>CASE_ID</th> <th>ACTIVITY</th> <th>TIMESTAMP</th> </tr> </thead> <tbody> <tr><td>1</td><td>'A'</td><td>2019-01-01 13:00:00</td></tr> <tr><td>1</td><td>'B'</td><td>2019-01-01 13:01:00</td></tr> <tr><td>1</td><td>'C'</td><td>2019-01-01 13:02:00</td></tr> <tr><td>2</td><td>'A'</td><td>2019-01-01 13:03:00</td></tr> <tr><td>2</td><td>'B'</td><td>2019-01-01 13:04:00</td></tr> <tr><td>2</td><td>'B'</td><td>2019-01-01 13:05:00</td></tr> <tr><td>2</td><td>'C'</td><td>2019-01-01 13:06:00</td></tr> <tr><td>3</td><td>'A'</td><td>2019-01-01 13:07:00</td></tr> <tr><td>3</td><td>'B'</td><td>2019-01-01 13:08:00</td></tr> <tr><td>3</td><td>'B'</td><td>2019-01-01 13:09:00</td></tr> <tr><td>3</td><td>'B'</td><td>2019-01-01 13:10:00</td></tr> <tr><td>3</td><td>'C'</td><td>2019-01-01 13:11:00</td></tr> </tbody> </table> <p style="text-align: center;">ACTIVITIES</p> <table border="1" style="margin-left: auto; margin-right: auto;"> <thead> <tr> <th>CASE_ID</th> </tr> </thead> <tbody> <tr><td>1</td></tr> <tr><td>2</td></tr> <tr><td>3</td></tr> </tbody> </table> <p style="text-align: center;">CASES</p> <table border="1" style="margin-left: auto; margin-right: auto;"> <thead> <tr> <th>ACTIVITIES.CASE_ID</th> <th>CASES.CASE_ID</th> </tr> </thead> <tbody> <tr> <td style="text-align: center;">1</td> <td style="text-align: center;">1</td> </tr> </tbody> </table> <p style="text-align: center;">Foreign Keys</p>	CASE_ID	ACTIVITY	TIMESTAMP	1	'A'	2019-01-01 13:00:00	1	'B'	2019-01-01 13:01:00	1	'C'	2019-01-01 13:02:00	2	'A'	2019-01-01 13:03:00	2	'B'	2019-01-01 13:04:00	2	'B'	2019-01-01 13:05:00	2	'C'	2019-01-01 13:06:00	3	'A'	2019-01-01 13:07:00	3	'B'	2019-01-01 13:08:00	3	'B'	2019-01-01 13:09:00	3	'B'	2019-01-01 13:10:00	3	'C'	2019-01-01 13:11:00	CASE_ID	1	2	3	ACTIVITIES.CASE_ID	CASES.CASE_ID	1	1	<table border="1"> <thead> <tr> <th>CASE_ID</th> <th>Variant</th> <th>Shortened</th> </tr> </thead> <tbody> <tr> <td>1</td> <td>'A, B, C'</td> <td>'A, B, C'</td> </tr> <tr> <td>2</td> <td>'A, B, B, C'</td> <td>'A, B, B, C'</td> </tr> <tr> <td>3</td> <td>'A, B, B, B, C'</td> <td>'A, B, B, C'</td> </tr> </tbody> </table> <p style="text-align: center;">RESULT</p>	CASE_ID	Variant	Shortened	1	'A, B, C'	'A, B, C'	2	'A, B, B, C'	'A, B, B, C'	3	'A, B, B, B, C'	'A, B, B, C'
CASE_ID	ACTIVITY	TIMESTAMP																																																										
1	'A'	2019-01-01 13:00:00																																																										
1	'B'	2019-01-01 13:01:00																																																										
1	'C'	2019-01-01 13:02:00																																																										
2	'A'	2019-01-01 13:03:00																																																										
2	'B'	2019-01-01 13:04:00																																																										
2	'B'	2019-01-01 13:05:00																																																										
2	'C'	2019-01-01 13:06:00																																																										
3	'A'	2019-01-01 13:07:00																																																										
3	'B'	2019-01-01 13:08:00																																																										
3	'B'	2019-01-01 13:09:00																																																										
3	'B'	2019-01-01 13:10:00																																																										
3	'C'	2019-01-01 13:11:00																																																										
CASE_ID																																																												
1																																																												
2																																																												
3																																																												
ACTIVITIES.CASE_ID	CASES.CASE_ID																																																											
1	1																																																											
CASE_ID	Variant	Shortened																																																										
1	'A, B, C'	'A, B, C'																																																										
2	'A, B, B, C'	'A, B, B, C'																																																										
3	'A, B, B, B, C'	'A, B, B, C'																																																										

**Fig. 12** Example for the VARIANT operator with and without reduced self-loops

operator can be wrapped by the SHORTENED command, which shortens self-loops to a maximum number of occurrences. In this way, it is possible to abstract from repeated activities and reduce the number of distinct variants. The limit for the length of the self-loops can be specified by an optional parameter. The default value for the maximum cycle length is 2.

Figure 12 shows an example query for the variant computation. The input data consists of an activity table and a case table, which can be joined by the foreign key relationship between the CASE\_ID columns of both tables. For each case, the query result shows the variant string (VARIANT column) and the variant string with reduced self-loops (SHORTENED column). Column VARIANT of the RESULT table shows individual variants (with a varying number of “B” activities) for each case, while column SHORTENED shows equal variants for the cases 2 and 3 where the third “B” activity of case 3 is omitted.

## 4.4 Conformance Checking

Besides process discovery, conformance checking is another important process mining technique that relates a process model to an event log [4]. It enables the identification of deviations of the as-is process—as reflected by the data in the event log—from the to-be process as defined by a prescriptive process model. Celonis PQL offers such conformance checking capability via the CONFORMANCE operator.

---

### Syntax 4 CONFORMANCE operator

---

```

CONFORMANCE ( activity_column, model )

READABLE ( CONFORMANCE ( activity_column, model ) )

model      ← places, transitions, flows, mapping, start_places, end_places
places     ← node_list
transitions ← node_list
start_places ← node_list
end_places  ← node_list
node_list  ← [ ( string_id )+ ]
flows      ← [ ( [ string_id string_id ] )+ ]
mapping    ← [ ( [ string string_id ] )+ ]

```

---

The syntax of the CONFORMANCE operator is shown in Syntax 4. It accepts an activity column and a description of a process model, as a Petri net, as input. The first part of the model description is a list containing all places of the Petri net, each specified by a unique string ID. It is followed by a similar list of all transitions. The third part of the model description is a list of flow relations, where each flow relation is specified as a pair of the source place and the target transition or a pair of the source transition and the target place, respectively. After that, a list of value pairs defines the mapping of activity names to the related transitions. The first value in such a pair is an activity name as a string while the second value is the ID of a transition that must be defined in the list of transitions. The last two parts of the model description are the lists of start and end places, respectively. Both lists consist of place IDs that must be specified in the first part of the model description.

The CONFORMANCE operator replays the activities' names from the input column on the process model. As a result, it adds a temporary column of type integer to the activity table. The value of a row in this fresh column indicates if there is a conformance issue or not. Also, the type of violation and the related activities in the process model are encoded in this value.

As the integer encoding is not suitable for the end-user, the CONFORMANCE operator can be wrapped in the READABLE command. If there is a violation, this will translate the encoding into a message explaining the violation.

Figure 13 shows an example query that uses the CONFORMANCE operator, which takes an activity table with three different cases as input. The process model that should be related to the event log is illustrated in Fig. 14. It is a simple Petri net

Query						
<pre>TABLE (   "ACTIVITIES"."CASE_ID",   "ACTIVITIES"."ACTIVITY",   CONFORMANCE ( "ACTIVITIES"."ACTIVITY" , [ "P_0" "P_1" "P_2" ] , [ "T_01" "T_12" ] ,     [ [ "P_0" "T_01" ] [ "T_01" "P_1" ] [ "P_1" "T_12" ] [ "T_12" "P_2" ] ] ,     [ [ 'A' "T_01" ] [ 'B' "T_12" ] ] , [ "P_0" ] , [ "P_2" ]   ) ,   READABLE (     CONFORMANCE ( "ACTIVITIES"."ACTIVITY" , [ "P_0" "P_1" "P_2" ] , [ "T_01" "T_12" ] ,       [ [ "P_0" "T_01" ] [ "T_01" "P_1" ] [ "P_1" "T_12" ] [ "T_12" "P_2" ] ] ,       [ [ 'A' "T_01" ] [ 'B' "T_12" ] ] , [ "P_0" ] , [ "P_2" ]     )   ) );</pre>						
Input			Output			
CASE_ID	ACTIVITY	TIMESTAMP	CASE_ID	ACTIVITY	Conformance	Readable
1	'A'	2019-01-01 13:00:00	1	'A'	2147483647	'Incomplete'
2	'A'	2019-01-01 13:01:00	2	'A'	0	'Conforms'
2	'C'	2019-01-01 13:02:00	2	'C'	-2	'C is an undesired activity'
3	'A'	2019-01-01 13:03:00	3	'A'	0	'Conforms'
3	'B'	2019-01-01 13:04:00	3	'B'	0	'Conforms'
ACTIVITIES			RESULT			

Fig. 13 CONFORMANCE operator example with integer encoding and readable explanation

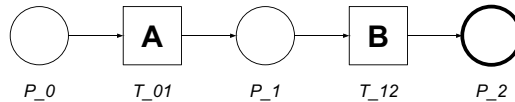


Fig. 14 Simple Petri net used in conformance checking example

consisting of two transitions and three places forming a trivial sequence of two activities “A” and “B”.

The result of the query consists of four columns with CASE\_ID as the first, and ACTIVITY as the second column. The third column (CONFORMANCE) shows the integer encoded result of the CONFORMANCE operator, while the fourth column (READABLE) shows intuitive messages explaining the deviations. Even though activity “A” matches the model, the first row is marked as *incomplete* because it is the last activity of case 1, which does not reach the end of the process due to the missing activity “B”. For case 2, the first activity (row 2) conforms, but the second activity (“C” in row 3) is not part of the process model and, therefore, is marked as an *undesired activity*. In contrast to that, case 3 fully conforms, which is indicated for all its activities (rows 4 and 5) of the output.

As the example illustrates, the model description is quite extensive even for such a small model that seems to contradict the design goal of keeping the language as simple as possible. However, the conformance operator is usually called from a GUI component. Using this component, the user can upload, automatically discover, or manually model a process model in Business Process Model and Notation (BPMN) [12] representation, which is automatically translated into the required



string description. The GUI component can also bind the model description string to a variable. Instead of defining the process model in the query, the user can simply insert the variable, which makes it much easier to use the CONFORMANCE operator in other GUI components. For example, the user can apply the CONFORMANCE operator in a filter in order to restrict a data table or chart only to cases that are marked as *incomplete*.

## 5 Use Cases

This section demonstrates the applicability of Celonis PQL for solving real-world problems of business users. First, we show how Celonis PQL is used to discover working capital optimizations. In our example, we identify early invoice payments to improve the on-time payment rate (Sect. 5.1). Second, we demonstrate how Celonis PQL is used to identify ping-pong-cases in IT service management processes in order to reduce ticket resolution times (Sect. 5.2). Third, we show the application of Celonis PQL for detecting segregation of duties violations to prevent fraud and errors in procurement (Sect. 5.3).

### 5.1 Working Capital Optimization by On-Time Payment of Invoices

Working capital is defined as the difference between a company's current assets and its current liabilities essential for the smooth operation of a business, and is a key figure for measuring a company's liquidity and its short-term financial health. Working capital management aims to optimize liquidity while ensuring sustained operations in the long term. Typical ways to optimize the working capital are inventory reduction, faster collection of receivables, and lengthening of the payable cycle. Activities for optimization within these areas are manifold. One example for lengthening the payable cycle is on-time payment of invoices by avoiding both early and late payments. Eradicating early payments can improve working capital by keeping assets until the day they are due. Preventing late payments can stop late payment penalties and allows to take advantage of cash discounts.

Query 1 shows a Celonis PQL statement for the calculation of the early payment ratio per vendor. Using this query, the user is able to discover the vendors that have the highest ratio of invoices paid more than three days early.

The distinction whether an invoice was paid more than three days before the due date is made within the CASE WHEN statement (lines 7–17) by calculating the throughput time with the CALC\_THROUGHPUT function (lines 9–12). The CALC\_THROUGHPUT operator takes the timestamp of the first occurrence of activity “Clear Invoice” and the timestamp of the first occurrence of activity

---

**Query 1** Average days of early payments per vendor
 

---

```

1 FILTER PROCESS EQUALS 'Clear Invoice';
2 FILTER PROCESS EQUALS 'Due Date passed';
3
4 TABLE (
5   "Invoice"."VendorName",
6   AVG(
7     CASE
8       WHEN COALESCE(
9         CALC_THROUGHPUT(
10          FIRST_OCCURRENCE['Clear Invoice'] TO
11          FIRST_OCCURRENCE['Due Date passed'],
12          REMAP_TIMESTAMPS("Activities"."Eventtime", DAYS)),
13          0
14        ) > 3
15       THEN 1.0
16       ELSE 0.0
17     END
18   ) AS "TooEarlyRatio"
19 ) ORDER BY "TooEarlyRatio" DESC;

```

---

“Due Date passed” and calculates the difference. The second parameter, given as REMAP\_TIMESTAMPS operator (line 12), counts time units in the specified interval DAYS based on the timestamps in the activity table to enable the calculation of the throughput time. As the CALC\_THROUGHPUT operator returns NULL if the end date is before the start date, the result of the calculation is wrapped in the COALESCE (lines 8–14) operator to return 0 in these cases. The result of the COALESCE operator is then compared to the specified three days (line 14). If the result is greater than 3, the CASE WHEN statement returns 1; otherwise 0.

The whole CASE WHEN statement is wrapped in the AVG operator (lines 6–18), allowing to calculate the ratio of invoices paid more than three days early. By specifying the vendor name ("Invoice"."VendorName") as a dimension in the TABLE statement (lines 4–19), the ratio is calculated per vendor. To get the vendors with the highest ratio of early invoice payments, the result of the AVG calculation is sorted in descending order by the ORDER BY statement (line 19). The two FILTER statements (lines 1 and 2) at the beginning of the query ensure that only cases with an already paid invoice and a specified due date are considered within the calculation.

## 5.2 Identifying Ping-Pong-Cases for Ticket Resolution Time Reduction

IT service management (ITSM) refers to the measurements and methods performed by an organization to ensure the optimal support of IT services provided to customers. Service-level agreements (SLAs) between the organization (also referred to as service provider) and the customers (also referred to as service user) define

**Query 2** Direct ping-pong-case ratio per country

---

```

1 TABLE (
2   "Tickets"."Country",
3   COUNT(DISTINCT
4     CASE
5       WHEN "Activities"."Activity" = 'Change Assigned Group'
6         AND "Activities"."Activity" = ACTIVITY_LEAD("Activities"."Activity",2)
7         AND "Activities"."Activity" != ACTIVITY_LEAD("Activities"."Activity",1)
8       THEN "Activities"."TicketId"
9       ELSE NULL
10      END
11    )
12  /
13  COUNT_TABLE("Tickets")
14  AS "DirectPingPongRatio"
15 ) ORDER BY "DirectPingPongRatio" DESC;

```

---

particular aspects of the provided support like availability, responsibility, and most important quality. SLAs are important factors influencing service quality levels and customer happiness. Therefore, compliance with defined SLAs is essential.

Customer support within ITSM systems is usually carried out by creating a ticket for each customer inquiry in the system and solving these tickets. Thus, an important key figure for ITSM is the resolution time of a ticket. A ticket is ideally resolved without the interference of many departments or teams. However, in so-called ping-pong-cases, a ticket is repeatedly going back and forth between departments or teams. This is massively slowing down the resolution time. To prevent this, the identification of ping-pong-cases is crucial.

Query 2 shows a Celonis PQL query to identify direct ping-pong-cases. A case in this context is equivalent to a ticket. Direct ping-pong refers to tickets in which the same activity appears (at least) two times with only one other activity in between, e.g., “Change Assigned Group” directly followed by “Review Ticket” directly followed by “Change Assigned Group”.

The query calculates whether a ticket is a ping-pong-case or not within the CASE WHEN statement (lines 4–10). If the current activity equals “Change Assigned Group”, the second next activity is equal to the current activity, and the next activity is not equal to the current activity, the ticket is classified as ping-pong-case and the CASE WHEN statement returns the ticket ID. The comparison between the current activity, the next, and the second next activity is achieved by using the ACTIVITY\_LEAD operator (lines 6 and 7). In general, the ACTIVITY\_LEAD operator returns the activity from the row that follows the current activity by offset number of rows within a case. As the timestamp column of the activity table is defined in the data model, the ACTIVITY\_LEAD operator can implicitly rely on the correct ordering of events. The CASE WHEN statement is wrapped in a COUNT operator (lines 3–11) to count the total number of ping-pong-cases. By adding DISTINCT (line 3) to the COUNT operator, it is guaranteed that a ticket is only counted once although ping-pong activities can occur multiple times within a ticket. The result of the COUNT operator is then divided by the total number of tickets to get

---

**Query 3** Indirect ping-pong-case ratio per country
 

---

```

1 FILTER "Activities"."Activity" = 'Change Assigned Group';
2
3 TABLE (
4   "Tickets"."Country",
5   COUNT(DISTINCT
6     CASE
7       WHEN ACTIVATION_COUNT("Activities"."Activity") > 1
8         AND "Activities"."Activity" != ACTIVITY_LAG("Activities"."Activity",2)
9         AND "Activities"."Activity" != ACTIVITY_LAG("Activities"."Activity",1)
10      THEN "Activities"."TicketId"
11      ELSE NULL
12     END
13  )
14  /
15  COUNT_TABLE("Tickets")
16  AS "IndirectPingPongRatio"
17 ) ORDER BY "IndirectPingPongRatio" DESC;

```

---

the ratio of ping-pong-cases. Thereby, the total number of tickets is calculated using the `COUNT_TABLE` operator (line 13). `COUNT_TABLE` is a performance-optimized function for counting the number of rows of a specified table. By specifying the country (`"Tickets"."Country"`, line 2) as a dimension in the `TABLE` statement (lines 1–15), the ratio of ping-pong-cases is calculated per country. In order to get the countries with the highest ratio of ping-pong-cases, the calculated ratio is sorted in descending order by the `ORDER BY` statement (line 15).

Query 3 shows a Celonis PQL query to identify indirect ping-pong-cases. Indirect ping-pong refers to tickets in which the activity “Change Assigned Group” appears at least two times with more than one other activity in between, e.g., “Change Assigned Group”, directly followed by “Review Ticket”, directly followed by “Do some work”, directly followed by “Change Assigned Group”.

The query shown in Query 3 calculates whether a ticket is an indirect ping-pong-case or not by using the operators `ACTIVATION_COUNT` (line 7) and `ACTIVITY_LAG` (lines 8 and 9) within a `CASE WHEN` statement (lines 6–12).

`ACTIVATION_COUNT` returns, for every activity, how many times it has already occurred (so far) in the current case. Within the `CASE WHEN` statement, the ticket ID is returned if the `ACTIVATION_COUNT` is greater than 1 and the current activity is not equal to the last and the second last activity. The latter comparison is calculated by the `ACTIVITY_LAG` operator. In general, `ACTIVITY_LAG` returns the activity from the row that precedes the current activity by offset number of rows within a case.

If one of the expressions in the `WHEN`-clause (lines 7–9) is `FALSE`, the `CASE WHEN` statement returns `NULL`. As in the example for direct ping-pong-cases, the `CASE WHEN` statement is wrapped in a `COUNT` operator (lines 5–13) to count the total number of ping-pong-cases. By adding `DISTINCT` (line 5) to the `COUNT` operator, it is guaranteed that a ticket is only counted once as an indirect ping-pong-case. The result of the `COUNT` operator is then, again, divided by the total

**Query 4** Violated SoD ratio per purchase organization

```

1 TABLE (
2   "PurchaseOrders"."PurchaseOrganization",
3   AVG(
4     CASE
5       WHEN SOURCE ( "Activities"."Department",
6         REMAP_VALUES ( "Activities"."Activity",
7           [ 'Request Approval', 'Request Approval' ],
8           [ 'Grant Approval', 'Grant Approval' ],
9           NULL
10      )
11     ) = TARGET ( "Activities"."Department" )
12   THEN 1.0
13   ELSE 0.0
14   END
15  )
16 ) AS "SoDViolationRatio"
17 ) ORDER BY "SoDViolationRatio" DESC;

```

number of tickets to get the ratio of indirect ping-pong-cases. Thereby, the total number of tickets is calculated using the `COUNT_TABLE` operator (line 15). The country ("Tickets"."Country", line 4) is specified as a dimension in the `TABLE` statement (lines 3–17) to calculate the ratio of indirect ping-pong-cases per country. To get the countries with the highest ratio of indirect ping-pong-cases, the calculated ratio is sorted in descending order by the `ORDER BY` statement (line 17). The `FILTER` statement (line 1) at the beginning of the query ensures that the current activity is “Change Assigned Group”.

### 5.3 Fraud Prevention by Identifying Segregation of Duties Violations

*Segregation of Duties* (SoD) is a concept based on shared responsibilities: It ensures that certain activities are not executed by the same person or department. It applies the four-eyes principle and decreases the power of an individual person or department in order to prevent fraud and errors. Therefore, the concept is essential for effective risk management and internal controls. In procurement, unauthorized or unnecessary purchase orders or purchase orders for personal use may occur if duties are not separated properly. With this in mind, it is best practice in procurement to have different people, or departments, for purchase approvals and invoice payment approvals.

Query 4 shows a Celonis PQL query for the calculation of the ratio of purchase orders in which the SoD for the activities “Request Approval” and “Grant Approval” was violated because the same department executed both tasks. The ratio is calculated per purchase organization to discover the ones with the highest violation ratio. Comparing whether the activities “Request Approval” and “Grant Approval”

were executed by the same department is done within the `CASE WHEN` statement (lines 4–14). The statement contrasts the source event department to the target event department by using the `SOURCE` and `TARGET` operators (lines 5–11). A detailed description of these operators can be found in Sect. 4.2. The `REMAP_VALUES` function (lines 6–10) passed as a parameter to the `SOURCE` operator allows to extract the activities “Request Approval” and “Grant Approval” by mapping them to the same name while mapping all the other activities to `NULL`. If the comparison between the source department and the target department returns true, the `CASE WHEN` statement returns 1 (line 12); otherwise 0 (line 13).

The `AVG` operator (lines 3–15) in which the `CASE WHEN` statement is wrapped calculates the ratio of violations of the SoD. By specifying the purchasing organization (“PurchaseOrders”.“PurchaseOrganization”) as a dimension in the `TABLE` statement (lines 1–17), the ratio of violations is calculated per purchase organization. The result of the `AVG` calculation is sorted in descending order by the `ORDER BY` statement (line 17) to get the organizations with the highest violation rate.

## 6 Implementation

Celonis PQL is the basis of a commercial product that promises interactive process mining and business intelligence using datasets with hundreds of millions of events. For this reason, the implementation of the language has to fulfill high requirements regarding performance, scalability, and low latency.

The implementation targets business intelligence and process mining because our experience is that process mining unfolds its full potential in combination with classic BI. Many insights into customer data could only be derived by taking into account further dimensional tables, in addition to the event log. For example, to find the country with the most segregation of duties violations (see Sect. 5.3), information about the countries has to be available.

In the past, different types of software addressed two fields: BI and process mining. BI is the domain of relational database systems. This is also reflected by the TPC-H benchmark [5], which is the de facto standard benchmark for analytical databases. The benchmark portrays a wholesale supplier in a data warehouse and focuses on classic BI questions, but it does not consider any process mining aspects. As a result, databases perform well in answering BI questions, but they are not optimized to answer process mining questions.

Besides the Celonis PQL implementation, process mining is done on relational databases, specialized implementations, or graph databases [6]. Graph databases can be considered as a reasonable choice, because a process instance can be interpreted as a graph. While they deliver a decent performance for process mining, a graph database is not optimized for business intelligence. This is why our objective was not to build upon an existing data processing solution. Instead, we wanted to

design a system from scratch, which combines techniques from relational and graph databases.

Like most state-of-the-art database systems, the Celonis PQL implementation is a main memory database. This means that it uses main memory as primary storage instead of the disk in order to avoid slow disk access. It is implemented in C++ and Java. C++ is used for all software modules in which active control over the main memory is necessary, like the storage layer and the performance-critical process mining algorithms. Java is used for non-performance-critical sections, like the parser, because of its memory safety.

The Celonis PQL Engine uses state-of-the-art techniques from the database research community like just-in-time (JIT) compilation and dictionary encoding. JIT compilation is a technique that generates and compiles code to execute a query. It allows achieving a good cache locality and a low number of CPU instructions resulting in a very high performance as shown by Neumann [11]. Dictionary encoding is a standard compression technique to reduce the memory overhead [10].

Elements that are taken from the graph community are some algorithms and data structures, like the adjacency list. The Celonis PQL Engine thereby exploits features of an event log, like process instances, which in most cases represent graphs with a rather low number of nodes.

The Celonis PQL Engine implementation focuses on analytical queries. It is snapshot-based, which means that the engine answers queries based on the data of the source system at a particular point in time. The data is not constantly updated. Instead, a bulk update mechanism is in place. This avoids some overhead, which would be introduced by a concurrency control mechanism like MVCC [2].

The Celonis PQL Engine implementation is also focused on scaling with the number of CPU cores within one server. The challenge here is that the implementation has to be lightweight enough to run on commodity laptop hardware, while it has to be sophisticated enough to make use of all the power a high-end server provides. This is achieved by parallel intra query execution, refer to Leis et al. [9] for details.

The Celonis PQL Engine implementation, however, is not designed to process queries across multiple servers. This is a conscious decision because the Celonis PQL Engine needs to provide results with low latency. Synchronizing multiple machines across the network to execute a Celonis PQL query adds overhead, which is against the low latency goal. The work of Schüle et al. [14] supports our single-node approach by demonstrating that a single server can handle even large scale applications like Wikipedia. To support such applications, lightweight in-memory compression techniques have to be in place. The Celonis PQL Engine implementation uses an approach for in-memory compression, which is inspired by the work of Lang et al. [7].

## 7 Celonis PQL and the Process Querying Framework

The Process Querying Framework (PQF) [13] is an abstract system consisting of a set of generic components to define a process querying method. Celonis PQL covers many of these components. This section describes the integration of Celonis PQL into the PQF.

Figure 15 illustrates how Celonis PQL instantiates the main components of the PQF. The first part of the framework (*Model, Record, and Correlate*) retrieves or creates the behavioral models and formalizes the business questions into process queries. The event logs are recorded by information systems like ERP or CRM systems, and extracted from these source systems into the Celonis IBC platform. The process models are either manually modeled (e.g., in Celonis IBC or in an external tool) or discovered by process mining techniques, like the Inductive Miner [8]. The correlation models are created by the conformance checking operator (see Sect. 4.4), relating activities of the event log to tasks in the process model. However, the different kinds of model repositories overlap due to their related storage, as relational data within the same data model. For example, the result column of the conformance checking operator (correlation model) is added to the activity table (event log).

The query intent of Celonis PQL is limited to *create* and *read*. While all supported kinds of behavioral models can be read, process models and correlation models can also be created by Celonis PQL queries (e.g., by process discovery and conformance checking). The *update* and *delete* query intents are not included—especially for the event logs—as they should always stem from the source systems. Therefore, event log updates can be achieved by delta loads that regularly extract the latest data from the source systems. The process querying instruction is usually defined by an analyst through a user interface. For example, the user defines the columns to be shown in a table, which can be considered as the query conditions. The selections from the user interface are then formalized into a Celonis PQL query.

The *Prepare* part of the framework focuses on increasing the efficiency of the query processing. The Celonis PQL Engine—that processes the queries—maintains a cache for query results, refer to Sect. 6. After the application starts, it warms up the cache with the most relevant queries derived from the *Process Querying Statistics* to provide fast response times. According to [13], the *Indexing* component does not only include classical index structures but also all kind of data structures for an efficient retrieval of data records. It is covered by the dictionary encoding of columns, as discussed in Sect. 6.

The *Execute* part of the framework combines an event log with an optional process model and a Celonis PQL query into a query result, which can be either a process model, KPIs, filtered and processed event log data, or conformance information. The concrete input and output of the query depend on the selected query intent and the query conditions. The *Filtering* component reduces the input data of the query. This can either be achieved by the `REMAP_VALUES` operator and the filter column of the `SOURCE` and `TARGET` operators, as described in Sect. 4.2,



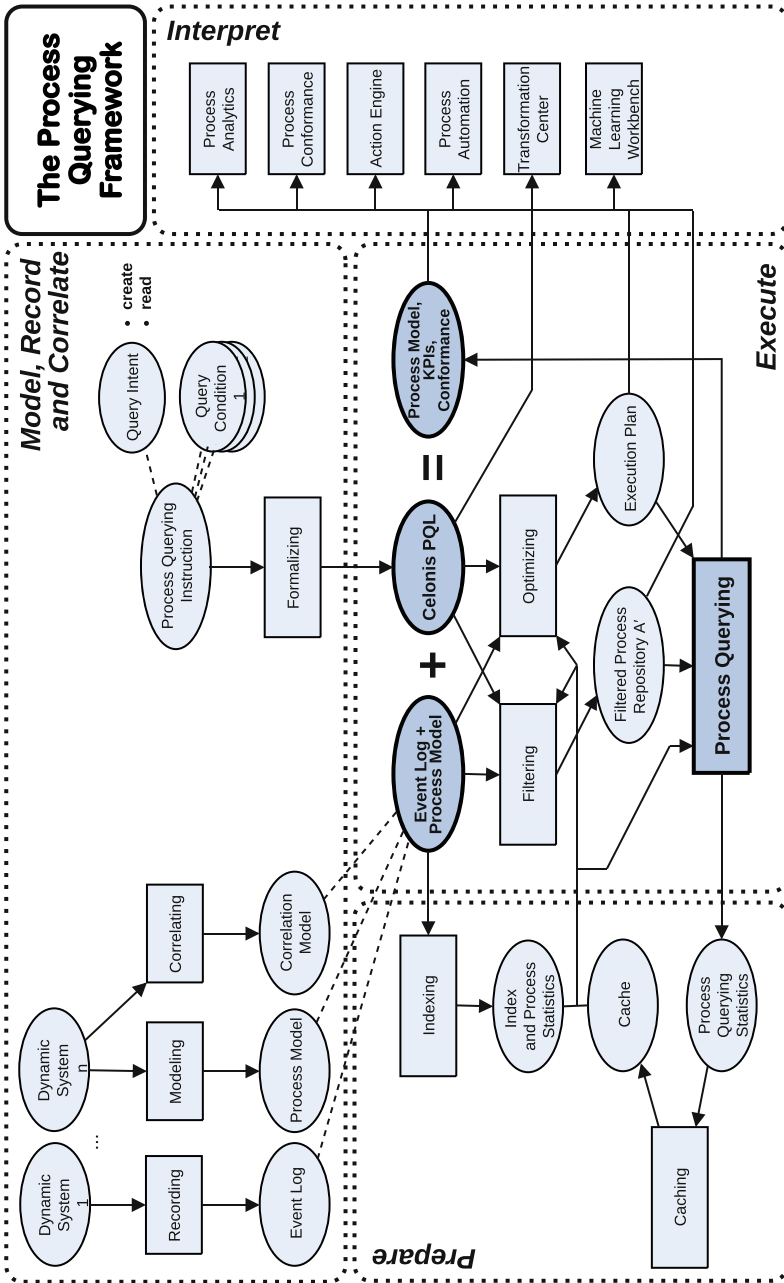


Fig. 15 Celonis PQL in the context of the Process Querying Framework

or by the general filter statement shown in the example in Fig. 6. The *Optimizing* component uses basic database technology to rewrite the query and create the *Execution Plan*, which describes a directed graph of operator nodes. The *Process Querying* component then executes the execution plan on the filtered data. It also retrieves data from the cache to avoid re-computation of either the full query or certain parts of it, which are shared with previous queries.

The *Interpret* part of the framework communicates the query results to the user and improves the user's comprehension of them. The applications in the Celonis IBC platform incorporate Celonis PQL and make the results accessible to the user. The *Process Analytics* presents the query results as process graphs, charts, and tables. Beyond pure visualization, it is highly interactive with dynamic filtering to drill-down the processes to specific cases of interest. This interactivity offered by all GUI components is achieved through the dynamic creation of Celonis PQL queries.

*Process Conformance* [16] shows the deviations between process model and event log in a comprehensive view, including a comparison of KPIs between conforming and non-conforming cases. In contrast to this, the focus of the *Action Engine* [1] is not to present query results, but to trigger user actions, for instance, by informing about deliveries that are expected to be late. The Action Engine can also trigger automated workflows that are executed by the *Process Automation* component. Within this component, the workflows can query data from the event log using Celonis PQL.

*Transformation Center* supports process monitoring. It historicizes the query results to show how the processes evolved over time. Finally, *Machine Learning Workbench* provides a platform for user-defined machine learning analyses over event logs and retrieves the event data using Celonis PQL queries.

## 8 Conclusion and Future Work

In this chapter, we introduced Celonis PQL, which is an independent query language with a custom-built query engine. It is highly optimized for process mining capabilities, and although it was inspired by SQL, the design of Celonis PQL is mostly driven by requirements of business users. A key difference to SQL is that Celonis PQL is a domain-specific language tailored toward a concrete data model. As a consequence, it does not require the user to explicitly define joins of data tables or groupings within the query, which is done implicitly.

As Celonis PQL comprises more than 150 different operators to process event data, we could only provide an overview of the major language features that are currently offered to users and showcase the expressiveness of the language with a few examples. Besides the description of the language, we illustrated the application of Celonis PQL within the various products available in Celonis IBC. Presented statistics show the extensive usage of Celonis PQL within these products. In addition, we presented the applicability of the query language for solving different real-world problems customers are facing, such as fraud prevention with segregation

of duties and speed up of service requests by identifying ping-pong-cases. Finally, we described the position of Celonis PQL within the Process Querying Framework (PQF) [13]. Celonis PQL instantiates all parts of the PQF, except for the capability to *simulate* models. Moreover, *create* and *read* query intents are covered.

Future work on Celonis PQL will focus on the implementation of new operators to further enrich the capabilities and use cases of the query language. Additionally, efforts will be made to improve query performance. New features will be developed in co-innovation projects with academic and commercial partners, and with our customers.

Readers can access a wide range of PQL functionalities for free. Business users can use Celonis PQL in the free Celonis Snap<sup>2</sup> version. Academic users can get free access to the full Celonis IBC technology including the wide range of Celonis PQL capabilities via the Celonis IBC—Academic Edition.<sup>3</sup>

## References

1. Badakhshan, P., Bernhart, G., Geyer-Klingeberg, J., Nakladal, J., Schenk, S., Vogelgesang, T.: The action engine – turning process insights into action. In: ICPM Demos 2019. CEUR Workshop Proceedings, vol. 2374. CEUR-WS.org (2019). <http://ceur-ws.org/Vol-2374/paper8.pdf>
2. Bernstein, P.A., Hadzilacos, V., Goodman, N.: Concurrency Control and Recovery in Database Systems. Addison-Wesley (1987). <http://research.microsoft.com/en-us/people/philbe/ccontrol.aspx>
3. Berti, A.: Increasing scalability of process mining using event dataframes: How data structure matters. CoRR **abs/1907.12817** (2019). <http://arxiv.org/abs/1907.12817>
4. Carmona, J., van Dongen, B.F., Solti, A., Weidlich, M.: Conformance Checking - Relating Processes and Models. Springer (2018). <https://doi.org/10.1007/978-3-319-99414-7>
5. Council, T.P.P.: TPC Benchmark H (Decision Support) Standard Specification Revision 2.18.0 (2018). [http://www.tpc.org/tpc\\_documents\\_current\\_versions/pdf/tpc-h\\_v2.18.0.pdf](http://www.tpc.org/tpc_documents_current_versions/pdf/tpc-h_v2.18.0.pdf)
6. Esser, S., Fahland, D.: Storing and querying multi-dimensional process event logs using graph databases. In: Process Querying (PQ) Workshop 2019, pp. 283–294 (2019)
7. Lang, H., Mühlbauer, T., Funke, F., Boncz, P.A., Neumann, T., Kemper, A.: Data blocks: Hybrid OLTP and OLAP on compressed storage using both vectorization and compilation. In: SIGMOD 2016, pp. 311–326. ACM (2016). <https://doi.org/10.1145/2882903.2882925>
8. Leemans, S.J.J., Fahland, D., van der Aalst, W.M.P.: Discovering block-structured process models from event logs - A constructive approach. In: PETRI NETS 2013. Lecture Notes in Computer Science, vol. 7927, pp. 311–329. Springer (2013). [https://doi.org/10.1007/978-3-642-38697-8\\_17](https://doi.org/10.1007/978-3-642-38697-8_17)
9. Leis, V., Boncz, P.A., Kemper, A., Neumann, T.: Morsel-driven parallelism: a NUMA-aware query evaluation framework for the many-core age. In: SIGMOD 2014, pp. 743–754. ACM (2014). <https://doi.org/10.1145/2588555.2610507>
10. Müller, I., Ratsch, C., Färber, F.: Adaptive string dictionary compression in in-memory column-store database systems. In: Proceedings of the EDBT 2014, pp. 283–294. OpenProceedings.org (2014). <https://doi.org/10.5441/002/edbt.2014.27>

---

<sup>2</sup> <https://www.celonis.com/snap-signup>.

<sup>3</sup> <https://www.celonis.com/academic-signup>.

11. Neumann, T.: Efficiently compiling efficient query plans for modern hardware. *PVLDB* **4**(9), 539–550 (2011). <https://doi.org/10.14778/2002938.2002940>. <http://www.vldb.org/pvldb/vol4/p539-neumann.pdf>
12. OMG: Business Process Model and Notation (BPMN), Version 2.0 (2011). <http://www.omg.org/spec/BPMN/2.0>
13. Polyvyanyy, A., Ouyang, C., Barros, A., van der Aalst, W.M.P.: Process querying: Enabling business intelligence through query-based process analytics. *Decis. Support Syst.* **100**, 41–56 (2017). <https://doi.org/10.1016/j.dss.2017.04.011>
14. Schüle, M., Schliski, P., Hutzelmann, T., Rosenberger, T., Leis, V., Vorona, D., Kemper, A., Neumann, T.: Monopedia: Staying single is good enough - the hyper way for Web scale applications. *PVLDB* **10**(12), 1921–1924 (2017). <https://doi.org/10.14778/3137765.3137809>. <http://www.vldb.org/pvldb/vol10/p1921-schuele.pdf>
15. van der Aalst, W.M.P.: *Process Mining - Data Science in Action*, Second Edition. Springer (2016). <https://doi.org/10.1007/978-3-662-49851-4>
16. Veit, F., Geyer-Klingenberg, J., Madrzak, J., Haug, M., Thomson, J.: The proactive insights engine: Process mining meets machine learning and artificial intelligence. In: *BPM Demos 2017*. CEUR Workshop Proceedings, vol. 1920. CEUR-WS.org (2017). [http://ceur-ws.org/Vol-1920/BPM\\_2017\\_paper\\_192.pdf](http://ceur-ws.org/Vol-1920/BPM_2017_paper_192.pdf)
17. Weijters, A.J.M.M., Ribeiro, J.T.S.: Flexible heuristics miner (FHM). In: *Proceedings of the IEEE CIDM 2011*, pp. 310–317. IEEE (2011). <https://doi.org/10.1109/CIDM.2011.5949453>

**Open Access** This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.



**Part IV**  
**Other Process Querying Methods**

# Process Querying Using Process Model Similarity



Remco M. Dijkman and Rik Eshuis

**Abstract** This chapter describes a specific form of process querying: process querying using process model similarity. This form of process querying can be applied to find one or more process models that are similar to a given query process model. Process querying is useful when searching within a collection of reference process models for a process model that is similar to your own. It is also useful when refactoring a collection of process models, in which case it can be applied to find similar process models and extract the similar parts to create common sub-processes. Process querying consists of a set of measures that can be used to quantify similarity between two process models as well as indexing techniques that can be used to efficiently find a process model within a collection of process models. The chapter shows the applicability of the techniques in a use case.

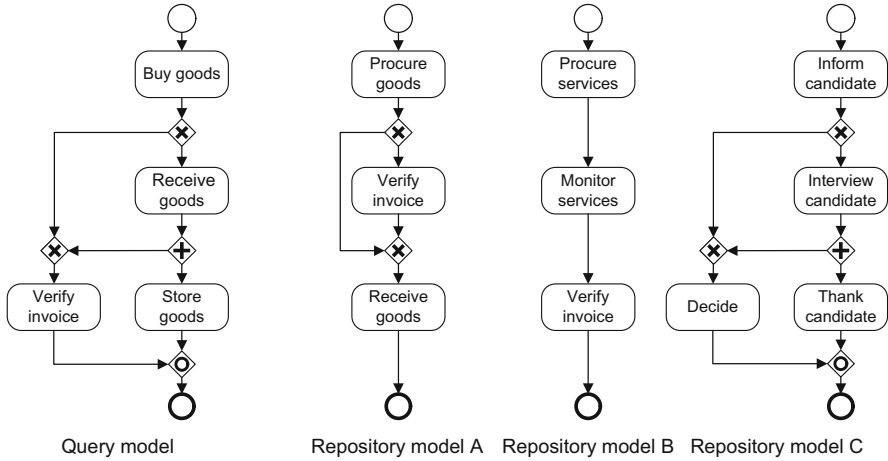
## 1 Introduction

A specific form of process querying is when the query takes the form of a complete process model. There are various use cases in which querying for similar business processes is useful. For example, when merging two organizations or organizational units, it is interesting to determine which process in one organization corresponds to which process in the other organization. This provides a basis for starting to merge the two organizations or organizational units on an operational level. As another example, for a collection of business processes it is interesting to establish where overlap exists in the collection, in order to reduce it.

Figure 1 shows an example, in which there is a simple business process repository, which contains three models (A, B, and C). The models are represented in the Business Process Model and Notation (BPMN). We can query the repository for a model that is similar to the “query model” that is also shown in the figure. In

---

R. M. Dijkman (✉) · R. Eshuis  
Eindhoven University of Technology, Eindhoven, The Netherlands  
e-mail: [r.m.dijkman@tue.nl](mailto:r.m.dijkman@tue.nl); [h.eshuis@tue.nl](mailto:h.eshuis@tue.nl)



**Fig. 1** Running example for process querying using process model similarity

this particular instance, the query should return model A, possibly model B, but not model C. Process similarity querying is the research area that concerns itself with the development of techniques that can do this.

The goal of this chapter is to introduce the topic of process model similarity in such a way that the readers can start to develop their own process model similarity querying techniques. To this end, and in accordance with the Process Querying Framework, the chapter presents measures that can be used to determine process model similarity. It then presents indexing structures that can be used to efficiently query a collection of business process models for models that are similar to a given (query) business process model. Finally, the chapter presents a detailed use case for business process similarity querying.

## 2 Measures of Business Process Similarity

The basic idea behind similarity-driven process querying is to be able to determine the level of similarity between two process models on a scale from 0 to 1. In that way, when a query model is presented to a process repository, that model can be compared to all models in the repository and the repository models that have a similarity to the query model above a certain threshold can be returned (in their order of similarity). In this section, we present various measures of business process similarity, after presenting the preliminary definitions that are required to define the measures.

### 2.1 Preliminaries

There exist many different business process modeling languages. To define business process similarity measures that work for each of these languages, and can even be used to compute the similarity of business processes that are modeled in different languages, we consider each business process as a graph [7].

**Definition 2.1 (Business Process Graph)** Let  $T$  be a set of types and  $\Omega$  be a set of labels. A business process graph is a tuple  $(N, E, \tau, \lambda, \alpha)$ , in which:

- $N$  is a finite set of nodes.
- $E \subseteq N \times N$  is a finite set of edges.
- $\tau : (N \cup E) \rightarrow T$  associates nodes and edges with types.
- $\lambda : (N \cup E) \rightarrow \Omega$  associates nodes and edges with labels.
- $\alpha : (N \cup E) \rightarrow \mathbb{P}(T \rightarrow \Omega)$  associates nodes and edges with attributes, where an attribute is a combination of a type and a label.

Figure 2 shows an example in which the query model from Fig. 1 is transformed into a graph. In this transformation, each of the nodes in the BPMN model becomes a node in a graph and each edge between nodes becomes an edge in the graph. The typing ( $\tau$ ) function can be used to distinguish different types of nodes, such as “tasks” and “gateways”, and possibly different types of edges, such as “flow” and “message”. Types can also be used to represent other relations, such as the

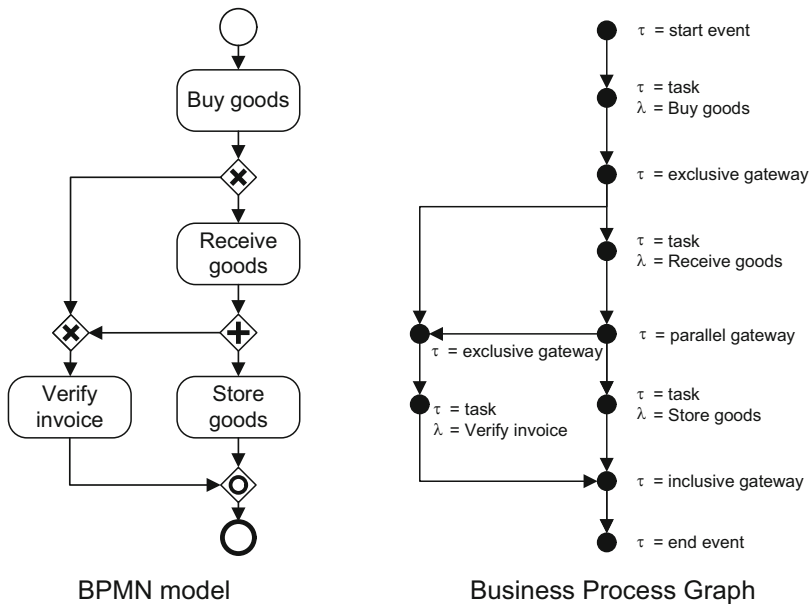


Fig. 2 Transformation of a BPMN model into a business process graph



relation between an embedded sub-process and its contents and the relation between a boundary event and the activity that it can interrupt. In our example, there is only one type of edge in the BPMN model (i.e., “control flow” edge), and since there can be no confusion, for ease of reading, we omit the edge types. The labeling ( $\lambda$ ) function associates nodes and edges with labels. The attribute ( $\alpha$ ) function associates nodes and edges with attributes. In our example, there are no attributes. However, we could, for example, associate the start event with an interarrival time:  $\alpha(s_1) = \{\textit{interarrival time, 5 minutes}\}$ .

In the remainder of this chapter, we use business process graphs constructed from BPMN models as examples, such that the different types of nodes, edges, and attributes are taken from BPMN. To also enable comparison between business process models in different notations, the types of nodes, edges, and attributes must be standardized and mappings between the types from the different notations and the standard types must be defined.

A similarity measure is a measure that, given two business process graphs, returns a score. Often the score is chosen to be between 0 and 1, where 0 means that the graphs are very different and 1 means that the graphs are very similar or even identical. However, other scoring models have also been used.

**Definition 2.2 (Similarity Measure)** Let  $\mathcal{G}$  be the domain of all business process graphs. A similarity measure is a function

$$\textit{sim}: \mathcal{G} \times \mathcal{G} \rightarrow \mathbb{R}.$$

## 2.2 Activity-Based Similarity Measures

Activity-based similarity measures use activities to determine the similarity between business process graphs. These measures require some way to determine the similarity of individual tasks. There are many different ways to do this. In this chapter, we discuss syntactic activity similarity, semantic activity similarity, and attribute activity similarity.

*Syntactic similarity* between two activities is determined based on a string comparison of the labels of the activities. A common way to determine the similarity of two strings is by using string-edit distance [18]. The string-edit distance of two strings is the minimum number of string operations (insert character, delete character, and substitute character) that is necessary to transform one string into another. For example, the edit distance between “alpha” and “beta” is 4: remove the first “a” and substitute the characters “lph” by “bet”. The edit distance of two strings can be transformed into a measure of similarity in various ways, for example by taking the inverse of the edit distance. Here, we will use the measure of edit distance similarity, which has some properties that are interesting from a similarity measure perspective, in particular that it yields a number between 0 and 1 (inclusive), that it yields the value 1 for identical strings, and that it is symmetric.

**Definition 2.3 (Edit-Distance Similarity)** Let  $s, s_1, s_2$  be strings. Furthermore, let  $|s|$  represent the length of the string  $s$  and  $ed(s_1, s_2)$  the edit distance of the two strings  $s_1$  and  $s_2$ . The edit distance similarity of the two strings  $s_1$  and  $s_2$ , denoted by  $es(s_1, s_2)$ , is defined as

$$es(s_1, s_2) = \frac{ed(s_1, s_2)}{\max(|s_1|, |s_2|)}.$$

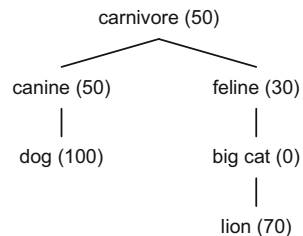
*Semantic similarity* between two activities is determined by comparing the “meaning” of the words in their labels. There exist various measures for the similarity of words, based on their “meaning” (e.g., [17, 20, 28]). Here, we introduce the one by Lin [20]. This measure assumes that words are organized in a tree, such as in the online dictionary WordNet [21]. In such a tree, we can determine the probability that a concept is a member of a class. For example, Fig. 3 shows a part of the WordNet tree of “word meanings.” It shows that a dog IS-A canine IS-A carnivore. The resulting tree must be trained based on a relevant corpus of words. This corpus is used to count the number of times a particular word occurs. For our example in Fig. 3, fictitious word counts are shown between brackets. Assuming that our entire vocabulary is made up of the words from Fig. 3, then the probability that a concept is a member of the class “dog” is  $\frac{100}{300}$  and the probability that a concept is a member of the class “canine” is  $\frac{150}{300}$ . Word similarity can then be defined as follows [20]:

**Definition 2.4 (Word Similarity)** Let  $w_1, w_2$  be two words. Let  $C_1$  be the class in the tree that word  $w_1$  is in and  $C_2$  the class that word  $w_2$  is in, and  $C_0$  the lowest common ancestor of  $C_1$  and  $C_2$ . Furthermore, let  $P(C)$  be the probability that an arbitrary word is a member of class  $C$ . The word similarity between the two words, denoted by  $ws(w_1, w_2)$ , is

$$ws(w_1, w_2) = \frac{2 \cdot \log P(C_0)}{\log P(C_1) + \log P(C_2)}.$$

This definition is based on information content theory. In this theory, the amount of information that is revealed by an observation equals  $-\log p$ , where  $p$  is the probability of that observation occurring. In our case, we model the probability that we encounter a particular word or synonym of that word. Based on that, we calculate the similarity by dividing the amount of information that the words have in

**Fig. 3** An excerpt from the WordNet “word meaning” tree



common by the amount of information that each word reveals individually. We need to multiply this value by 2 to ensure that a value between zero and one is always produced, because the divisor is the sum of two sub-classes. In the example from Fig. 3, for the words  $w_1 = \text{“dog”}$  and  $w_2 = \text{“canine”}$ ,  $P(C_1) = \frac{100}{300}$ ,  $P(C_2) = \frac{150}{300}$ , and  $P(C_0) = \frac{150}{300}$ , such that  $ws(w_1, w_2) \approx 0.77$ .

Word similarity can be further improved using a number of pre-processing techniques. Stemming [27] can be used to reduce different inflections of words to their “stem” form (e.g., to reduce “levels”, “leveling”, and “leveled” to “level”), thus enabling them to be related more easily to their dictionary entry. Part-of-speech tagging [3] can be used to determine which word in the label of a task is a noun, which is a verb, and so on, to ensure that the correct entry in the dictionary is used. It could, for example, be used to distinguish the noun “level” from the verb “to level”. Finally, stop-word removal can be used to remove commonly occurring words, such as “the” and “an”, that do not add to the semantics of the label. For example, from the perspective of semantic similarity, it does not matter if the label contains the text “the client” or just “client”.

Once the similarity between words has been established, the similarity between two labels,  $\lambda_1$  and  $\lambda_2$ , that are composed of these words can be established. A simple way to do that is by determining the similarity between each pair of words consisting of a word from label  $\lambda_1$  and a word from label  $\lambda_2$ , and subsequently keeping the set of pairs that have the highest average similarity and in which each word is in at most one pair. The average similarity is the similarity of the labels. For example, consider two labels “evaluate customer application” and “application assessment” with a similarity of 0.1 between the words “evaluate” and “application”, and between “customer” and “application”, a similarity of 1.0 between “application” and “application”, a similarity of 0.8 between “evaluate” and “assessment”, and a similarity of 0.1 between “customer” and “assessment”, and between “application” and “assessment”. The mapping between words from the first label and words from the second label that produces the highest average similarity is the mapping that maps “application” to “application” with a score of two times 1.0 (one for each word) and “evaluate” to “assessment” with a score of two times 0.8. Consequently, with a total of five words that can potentially be mapped, the score is  $\frac{2 \cdot (1.0 + 0.8)}{5} = 0.72$ .

*Attribute similarity* between two activities determines similarity based not only on the labels, but also on the attributes, of these activities. Attribute similarity can be computed by constructing a “virtual document” [33] for each activity. In this context, a virtual document is a dynamically constructed description of an activity that consists of the activity’s label as a title, as well as document sections for each attribute, where each section has the name of the attribute as its title and the value of the attribute as its content. Once these documents have been constructed, typical text-search techniques can be used to compare the documents. For example, for an activity that has the label “evaluate application”, input data “application”, and output data “evaluation”, the following virtual document can be constructed: “Label: evaluate application. Input data: application. Output data: evaluation”.

Syntactic activity similarity, semantic activity similarity, and attribute activity similarity lead to similarity of pairs of activities. This still must be reworked into similarity of complete models. This can be done by determining an optimal mapping between activities from the query model and activities from the document model, based on their similarity. More precisely:

**Definition 2.5 (Activity-Based Similarity)** Let  $B_1 = (N_1, E_1, \tau_1, \lambda_1, \alpha_1)$  and  $B_2 = (N_2, E_2, \tau_2, \lambda_2, \alpha_2)$  be two business process graphs. Furthermore, let  $lsim$  be a measure to determine the similarity of activity labels as defined above, and  $ts \subseteq T$  be the set of node types that represent activities. An optimal similarity mapping is a mapping  $M_{opt} : N_1 \rightarrow N_2$ , such that there exists no  $M' : N_1 \rightarrow N_2$  with  $\sum_{(n_1, n_2) \in M'} lsim(n_1, n_2) > \sum_{(n_1, n_2) \in M_{opt}} lsim(n_1, n_2)$ . The activity-based similarity of  $B_1$  and  $B_2$  is the similarity induced by the optimal similarity mapping  $M_{opt}$  between the tasks of the business process graphs  $T_1 = \{n_1 \in N_1 | \tau(n_1) \in ts\}$  and  $T_2 = \{n_2 \in N_2 | \tau(n_2) \in ts\}$ :

$$\frac{2 \cdot \sum_{(n_1, n_2) \in M_{opt}} lsim(n_1, n_2)}{|T_1| + |T_2|}.$$

The intuition behind this measure is that divisor represents the theoretical maximum similarity, which is reached when each task from both labels is mapped with a similarity score of 1. The dividend represents the actual similarity score of mapped tasks. This is multiplied by 2, because each mapping maps two tasks. It is important to note that the optimal mapping can assign a task to at most one other task.

For example, consider the query model and repository model A from our running example in Fig. 1. Let the similarity between identical labels be 1, the similarity between “Buy goods” and “Procure goods” be 0.9, the similarity between “Store goods” and “Receive goods” be 0.4, and the similarity between the other labels be 0. It is easy to see that the optimal similarity mapping maps “Buy goods” to “Procure goods”, “Receive goods” to “Receive goods”, and “Verify invoice” to “Verify invoice”. “Store goods” is not mapped because the mapping ( $\rightarrow$ ) requires that each task is mapped at most once and mapping “Verify invoice” to one of the tasks from the other model would lower the similarity. Consequently, the overall similarity of the two models is  $\frac{2 \cdot 2.9}{4+3}$ .

### 2.3 Structure-Based Similarity Measures

One common way to compare two (graph-based) models is by using the notion of graph-edit distance [4]. The idea behind graph-edit distance is similar to the idea behind string-edit distance. It is the minimum number of graph-edit operations that is required to change one graph into another. The basic graph-edit operations that are considered are:

- Insert a node into or remove a node from a graph.

- Insert an edge into or remove an edge from a graph.
- Substitute a node for another node.

Each of the edit operations has a certain cost associated with it. We can, for example, impose a cost of 1 for insertions and deletions. Substitution of nodes is used to replace a node by another node. It would typically be used for nodes in one model for which there exists a node in the other model with a similar label, type, and attributes. The cost of substituting a node for another node could then be related to that similarity (e.g., cost = 1 - similarity). However, these costs can be parameterized and set to yield the best possible result, based on experimentation.

Against this background, we define graph-edit distance as follows.

**Definition 2.6 (Graph-Edit Distance)** Let  $B_1 = (N_1, E_1, \tau_1, \lambda_1, \alpha_1)$  and  $B_2 = (N_2, E_2, \tau_2, \lambda_2, \alpha_2)$  be two business process graphs. Furthermore, let  $lsim$  be a measure to determine the similarity of activity labels as defined above. A mapping  $M : N_1 \rightarrow N_2$  imposes an edit distance between the two graphs, denoted by  $ed_M(B_1, B_2)$ , which is equal to the sum of:

- The number of deleted nodes:  $|\{n_1 \in N_1 | \neg \exists (n_1, n_2) \in M\}|$
- The number of inserted nodes:  $|\{n_2 \in N_2 | \neg \exists (n_1, n_2) \in M\}|$
- The number of deleted edges:  $|\{(n_1, n'_1) \in E_1 | \neg \exists (n_1, n_2), (n'_1, n'_2) \in M, \text{ such that } (n_2, n'_2) \in E_2\}|$
- The number of inserted edges:  $|\{(n_2, n'_2) \in E_2 | \neg \exists (n_1, n_2), (n'_1, n'_2) \in M, \text{ such that } (n_1, n'_1) \in E_1\}|$
- The distance between substituted nodes:  $2 \cdot \sum_{(n_1, n_2) \in M} 1 - lsim(\lambda_1(n_1), \lambda_2(n_2))$

The graph-edit distance between the two graphs, denoted by  $ged(B_1, B_2)$ , is the distance that is defined by a mapping  $M_{opt} : N_1 \rightarrow N_2$  that is optimal, in the sense that there exists no  $M' : N_1 \rightarrow N_2$  with  $ed_{M'}(B_1, B_2) < ed_{M_{opt}}(B_1, B_2)$ .

The distance between the substituted nodes is multiplied by two because it applies to two nodes instead of one (inserted or deleted) node. This ensures that the technique will always prefer to substitute nodes, even in the case of a bad match.

The graph-edit distance similarity can now easily be defined by observing that the maximum distance between two graphs is the distance that is computed when the mapping that induced the edit distance is empty. In that case the distance is equal to the number of nodes and edges in both graphs. Consequently, the graph-edit distance similarity is equal to 1 minus the fraction between the actual distance and this theoretical maximum distance.

**Definition 2.7 (Graph-Edit Distance Similarity)** Let  $B_1 = (N_1, E_1, \tau_1, \lambda_1, \alpha_1)$  and  $B_2 = (N_2, E_2, \tau_2, \lambda_2, \alpha_2)$  be two business process graphs. The graph-edit distance similarity is defined as

$$1 - \frac{ged(B_1, B_2)}{|N_1| + |N_2| + |E_1| + |E_2|}.$$

As an example, consider the query model and repository model A from the running example in Fig. 1. The optimal mapping for graph-edit distance maps the start events to each other (with a similarity of 1), “Buy goods” to “Procure goods” (with a similarity of 0.9), the XOR-split from the query model to the first XOR-split from repository model A (with a similarity of 1), the OR-join from the query model to the second XOR-split from the repository model (with a similarity of 1), “Verify invoice” to “Receive goods” (with a similarity of 0), “Receive goods” to “verify invoice” (with a similarity of 0), and the AND-join to the end event (with a similarity of 1). For this mapping, there are 3 deleted nodes: the AND-split, the “end event”, and “Store goods”. There are 0 inserted nodes, 5 deleted edges, 1 inserted edge, and a substitution score of  $2 \cdot 2 \cdot 1$ . In this way, the similarity becomes  $1 - \frac{4 \cdot 2 + 3 + 0 + 5 + 1}{10 + 7 + 11 + 7} \approx 0.62$ . Admittedly, this is not a very good mapping. The quality of mappings can be fine-tuned by assigning weights to the different parts of the mapping. For example, if the weight for inserted and deleted nodes is much higher than that of inserted and deleted edges, the graph-edit distance would prefer to map “Buy goods” to “Procure goods”, “Receive goods” to “Receive goods”, and “Verify invoice” to “Verify invoice”. In addition, we can prevent that different types of nodes are matched to each other. In prior work [7], we have experimented with different settings to determine settings that yield good similarity scores. We refer to that work if the reader is interested in creating an optimized similarity measure by setting weights.

A challenge that remains is determining the optimal mapping. A simple strategy to determine an optimal mapping is by exhaustively trying all possible mappings and keeping the one with the lowest edit distance. However, since the number of possible mappings is exponential over the number of nodes, this would be a very computationally expensive strategy. There exist many alternative algorithms [6] that find an optimal mapping that is the same as, or close to, the one found by the exhaustive strategy but have much better computational performance.

Additional operations can be envisioned and have indeed been used to improve results. In particular, “grouping” and “ungrouping” operations have been used [34]. These operations are used to acknowledge that processes are not always modeled at the same level of detail. If a part of one process is modeled in more detail than the related part in the other process, its nodes can be grouped to bring it to the same level of detail. Similarly, if the process is modeled in less detail, its nodes can be ungrouped.

## 2.4 Behavior-Based Similarity Measures

The third class of measures not only takes the structure of a business process, but also the behavior into account. In particular, these measures consider that different combinations of gateways may lead to similar behavior and can therefore be considered behaviorally, but not structurally, similar. This is a property that is not captured in purely structural similarity measures.

We use the notion of causal footprint [8] to capture and compare the behavior of two processes. A causal footprint is an approximation of the behavior of a business process, which represents the tasks that must succeed the occurrence of another task and the tasks that must precede the occurrence of another task. This is represented by look-ahead links and look-back links. The look-ahead link of a task  $t$  is the set of tasks  $la$ , such that after the occurrence of  $t$  at least one of the tasks from  $la$  must occur at a later stage in the process. The look-back link of a task  $t$  is the set of tasks  $lb$ , such that before the occurrence of  $t$  at least one of the tasks from  $lb$  must have occurred. In the query model of our running example in Fig. 1, “Buy goods” will always be succeeded by “Verify invoice” and sometimes by “Receive goods” and “Store goods”. This leads to the look-ahead link (“Buy goods”, {“Verify invoice”}) and the look-back links ({“Buy goods”}, “Verify invoice”), ({“Buy goods”}, “Receive goods”), and ({“Buy goods”}, “Store goods”). More precisely, the causal footprint of a business process can be defined as follows.

**Definition 2.8 (Causal Footprint)** The causal footprint of a business process is a tuple  $(T, L_{lb}, L_{la})$ , in which:

- $T$  is the set of tasks in the business process.
- $L_{lb} \subseteq \mathbb{P}(T) \times T$  is the minimal set, such that for each  $(lb, t) \in L_{lb}$ , it holds that for each execution trace of the process, if  $t$  appears in the execution trace, some  $t' \in lb$  must appear before  $t$  in the execution trace.
- $L_{la} \subseteq T \times \mathbb{P}(T)$  is the minimal set, such that for each  $(t, la) \in L_{la}$ , it holds that for each execution trace of the process, if  $t$  appears in the execution trace, some  $t' \in la$  must appear after  $t$  in the execution trace.

We define the similarity between processes in a vector space that is determined by their causal footprints.

**Definition 2.9 (Model Vector Space)** The vector space for a set of causal footprints is the vector space that has one dimension for each task in the causal footprints, one dimension for each look-ahead link, and one dimension for each look-back link.

For example, focusing on the tasks “Buy goods”, “Procure goods”, and “Verify invoice” for the query model and repository model A from Fig. 1, the vector space would be made up of the dimensions: “Buy goods”, “Procure goods”, “Verify invoice” (“Buy goods”, {“Verify invoice”}), ({“Buy goods”}, “Verify invoice”), ({“Procure goods”}, “Verify invoice”).

Subsequently, we can determine the value of each model in each dimension and with that the vector that represents each model in the vector space. This requires that an optimal (activity-based) similarity mapping between the tasks of the models has been established. The value in each dimension is then established as follows.

**Definition 2.10 (Model Vector)** Let  $M_{opt} : T_1 \rightarrow T_2$  be the optimal similarity mapping (conform Definition 2.5) that is established between the tasks of two causal footprints  $(T_1, L_{lb_1}, L_{la_1})$ ,  $(T_2, L_{lb_2}, L_{la_2})$  and let  $lsim$  be the label similarity

measure that was used to establish the mapping. The value of a causal footprint in a dimension  $d$  is

- 1, If  $d$  is a dimension that represents a task, look-ahead link, or look-back link that is contained in the process model.
- Otherwise it is:
  - $lsim(t, t')$ , if  $d$  is a dimension that represents a task  $t$ , and the process model contains a task  $t'$ , such that  $(t, t') \in M_{opt}$  or  $(t', t) \in M_{opt}$
  - $\frac{lsim(t, t')}{2^{|lb|}}$ , if  $d$  is a dimension that represents a look-back link  $(lb, t) \in L_{lb_1}$  or  $(lb, t) \in L_{lb_2}$ , and the process model contains a task  $t'$ , such that  $(t, t') \in M$  or  $(t', t) \in M$
  - $\frac{lsim(t, t')}{2^{|la|}}$ , if  $d$  is a dimension that represents a look-ahead link  $(t, la) \in L_{la_1}$  or  $(t, la) \in L_{la_2}$ , and the process model contains a task  $t'$ , such that  $(t, t') \in M$  or  $(t', t) \in M$
  - 0, otherwise

Now that a vector is established for each causal footprint, and therewith for each model, the similarity between the models can be established using typical vector distance measures. In this chapter, we use the Euclidian distance.

**Definition 2.11 (Behavior-Based Similarity)** Let  $M_1$  and  $M_2$  be two process models with model vectors  $\vec{m}_1$  and  $\vec{m}_2$  as defined in Definition 2.10 in the vector space defined in Definition 2.9. The behavior-based similarity of the two models is

$$\frac{\vec{m}_1 \times \vec{m}_2}{|\vec{m}_1| \cdot |\vec{m}_2|}.$$

### 3 Indexing Structures for Business Process Similarity

To apply the measures from the previous section to search for a model in a repository, the similarity of the query model to each of the models in the repository must be computed. Subsequently, the repository models that have a “sufficient” match to the query model are returned. The drawback of this approach is that it requires a large number of comparisons. Indexing structures can be used to reduce the number of comparisons. In this section, we discuss two possible indexing structures: a tree-based index and feature nets (or F-Nets).

#### 3.1 Tree-Based Index and Proper Metrics

It is possible to organize models in a tree structure. The tree structure can be set up in such a way that measuring the similarity between a model and multiple other models



can be done efficiently because the comparison can start at the root model and then be recursively continued with only the child models that are “sufficiently similar.”

To be able to efficiently measure similarity with multiple models in this manner, the so-called proper distance metric must be used [15].

**Definition 3.1 (Proper Distance Metric)** Let  $\mathcal{G}$  be the domain of all business process graphs. A distance measure is a function  $d : \mathcal{G} \times \mathcal{G} \rightarrow \mathbb{R}$  that represents the distance between two business process graphs. A distance measure is a proper distance metric, if the following conditions hold:

- Symmetry:  $\forall g_1, g_2 \in \mathcal{G} : d(g_1, g_2) = d(g_2, g_1)$
- Non-negativity:  $\forall g_1, g_2 \in \mathcal{G} : d(g_1, g_2) \geq 0$
- Identity:  $\forall g_1, g_2 \in \mathcal{G} : g_1 = g_2 \Rightarrow d(g_1, g_2) = 0$
- Triangle inequality:  $\forall g_1, g_2, g_3 \in \mathcal{G} : d(g_1, g_3) \leq d(g_1, g_2) + d(g_2, g_3)$

Note that we measure distance rather than similarity here. However, a similarity measure can easily be transformed into a distance measure, for example, by taking the inverse of the similarity or 1 minus the similarity. Indeed, some of the similarity measures that are introduced in the previous section are based on distance measures.

There exist different tree structures that can be used to build an efficient index for similarity search based on proper metrics. One of the possible tree structures is the metric tree [31]. A metric tree is constructed in the following manner. First, an arbitrary business process graph  $r \in \mathcal{G}$  is selected as the root node. Subsequently, the median distance,  $m$ , of all business process graphs  $g \in \mathcal{G}$ ,  $g \neq r$  is determined. Then, all models with  $d(g, r) \leq m$  are put in the left subtree of  $r$ , and all models with  $d(g, r) > m$  are put in the right subtree of  $r$ . Next, the tree is recursively built further for the left subtree and for the right subtree, using the set of models that is placed in the respective subtree.

The metric tree can be searched efficiently for a query graph  $q$  and a threshold  $t$ , where each graph  $g \in \mathcal{G}$  with  $d(g, q) \leq t$  must be returned. To obtain the resulting set of graphs, the query is recursively executed on a node of the metric tree that represents a graph  $g$  and a median similarity  $m$  of its children, starting from the root node, as follows:

1. If  $d(q, g) \leq t$ , include  $g$  in the set of results.
2. If  $d(q, g) \leq m + t$  and the node has a left subtree, traverse the left subtree.
3. If  $d(q, g) > m - t$  and the node has a right subtree, traverse the right subtree.

It is easy to prove that this algorithm does not skip graphs that should be included in the set of results because they are above the threshold distance (i.e., a graph  $g' \in \mathcal{G}$  is skipped iff  $d(g', q) > t$ ) [31]. For the left side of the tree, we can prove this as follows. None of the graphs  $g'$  in the left side of the tree are included, if  $d(q, g) > m + t$ . We know based on triangle inequality (Definition 3.1) that  $d(q, g) \leq d(q, g') + d(g', g)$ , such that we can conclude:  $d(q, g') + d(g', g) > m + t$ . We know from the construction of the left subtree that any  $g'$  that is part of the subtree of  $g$  has  $d(g', g) \leq m$ . This means that  $d(g', g) = m - \delta$  for some  $\delta \geq 0$ . Using this equation in  $d(q, g') + d(g', g) > m + t$ , we can conclude that  $d(q, g') > t + \delta$ . Consequently,  $d(q, g') > t$ , so we were right to exclude the graphs

from this subtree, because their distance to the query was above the threshold. The proof for the right side of the tree is analogous.

### 3.2 F-Net

An alternative indexing structure for similarity-driven process querying is the feature net (or F-Net) [35]. An F-Net works by extracting small parts from the models in the repository, which we call features, organizing them in an index and keeping pointers to the models of which they are a part. A similarity query model can also be decomposed into features, and via the index, the models can be returned that have a sufficient number of features that overlap with the query model.

Figure 4 shows an example of an F-Net. It is a part of the F-Net that can be constructed for the models of the running example. The features that are included here are word, label, and composite features, where we use the term composite feature to represent features that involve multiple tasks. Each feature references the models that contain it, and the features that can be constructed from it. For example, the word feature “procure” references the models *A* and *B* from Fig. 1, which contain that feature, and the label features “Procure goods” and “Procure services” that can be constructed from it.

To traverse an F-Net to find the models that should be returned as the result of a given query, we assume that a matching function has been defined that returns, given two features, if they are sufficiently similar to be called “matching features.” This matching function can typically be defined using the similarity functions that are

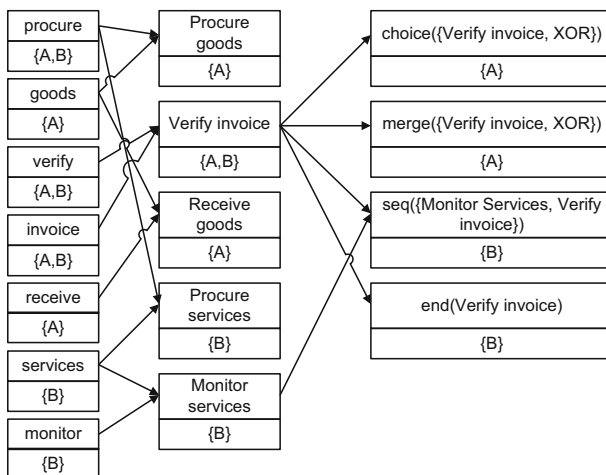


Fig. 4 A part of the F-Net for the running example of Fig. 1

defined in the previous section, and some threshold value that defines how similar two features need to be, to be called “matching features.”

**Definition 3.2 (Feature Matching)** Let  $f_1$  and  $f_2$  be two features. We say that two features match, denoted as  $match(f_1, f_2)$ , if and only if they are of the same type and have a similarity level, according to a selected similarity function, that exceeds a given threshold.

The F-Net can be traversed as follows. First, we decompose the query model into features. For each feature of the query model, we can find matching features in the F-Net, as well as the repository models that have these features. For example, the search for matching word features can be done efficiently by using a text-based index. For each pair of matching features, we can determine the sets of features that are constructed from it in the query model and in the F-Net. These two sets can be compared for matching features again. The benefit is that the sets of features that must be compared can be small. We repeat the process of finding matching features, until the F-Net no longer points to new features.

An F-Net traversal, given a query model, produces two results: (1) a set of models; (2) for each model a set of features that matches the features of the query model. We can use that to compute a similarity estimate of the query model to each of the repository models from the result set. We compute the similarity estimate as the fraction of matching features.

**Definition 3.3 (F-Net-Based Similarity Estimate)** Let  $F_1$  and  $F_2$  be two sets of features. Furthermore, let  $match : F_1 \times F_2 \rightarrow \{\text{True}, \text{False}\}$  be the feature matching function as it is defined in Definition 3.2. The similarity estimate of the two sets of features  $F_1$  and  $F_2$  is

$$\frac{|\{f_1 \in F_1, f_2 \in F_2 | match(f_1, f_2)\}|}{|F_1| + |F_2|}.$$

While the estimate can be used as a similarity measure, it is also possible to use it as a pre-processing step for similarity search. To use it as a pre-processing step, the F-Net-based similarity estimate is used to categorize the repository models into three classes:

- Models that are judged as not similar enough based on the estimate and will not be returned as an answer to the similarity query
- Models that are judged as similar enough based on the estimate and will be returned as an answer to the similarity query
- Models for which it is still uncertain if they are similar enough and that still have to be compared to the query model, using one of the computationally more expensive similarity measures described in the previous section.

In various experiments [35], we have shown that the F-Net-based similarity estimate can be used as a pre-processing step at no penalty or at a minor penalty to the quality of the search results; search result quality is defined in terms of models that are returned as answers to a similarity query, but should not be, or models that

are not returned but should have been. However, while the penalty to quality is minor, the search can be executed up to 10 times faster than when no pre-processing step is used.

## 4 Use Case: Finding Optimal Outsourcing Partners

So far, the focus has been on technical definitions of process similarity. We conclude this chapter by illustrating the practical use of similarity measures in the context of business process outsourcing, in which a provider performs a process on behalf of a client. In establishing an outsourcing relation between client and provider, similarity measures play an essential role to determine whether or not there is a match between the requested process of the client and the offered process of the provider. To paint a complete picture of business process outsourcing, we first sketch different outsourcing scenarios and requirements on matching relations. Next, we introduce different types of matching relations for process outsourcing. Next, we elaborate the different matching relations and explore how they connect with the different types of similarity measures as they are defined in the previous sections. Finally, we discuss the post-matching phase, in which similarity measures can also play a role, and we reflect on other uses of similarity measures for business process outsourcing.

### 4.1 *Scenarios and Requirements for Business Process Outsourcing*

Several scenarios for process matching in the context of business process outsourcing have been proposed in the literature [9, 12, 13], partly inspired by matching relations for semantic Web services [16, 19, 23]:

1. An Original Equipment Manufacturer (OEM) of high-tech products outsources the production of a high-precision component to a supplier. The OEM requires the just-in-time delivery of the component to assemble it into a custom-made production unit. The supplier must comply with the specified request of the client, i.e., it must offer a process that is the same as the requested processes.
2. An organization outsources the logistics of shipping products to consumers to a logistics service provider. The logistics service provider must comply with the process request of the client organization but may offer additional tasks in the process that the client can choose to ignore. In other words, the offered process delivers what is requested, but it is not the same as the requested process.
3. An organization outsources part of its business process to a service provider. To find an optimal match, for instance the service provider offering the lowest price, the organization is accepting a match that is close but not exact. Thus, the

offered and requested processes may not be the same. After selecting a provider, the client organization aligns its process to the process offered by the provider.

From these scenarios, the following requirements on matching can be inferred:

- *Relevance*: Matching relations should ignore irrelevant differences in syntax and structure. At the level of activities, an irrelevant difference can be two activities that are labeled differently even though are in an ontological sense equivalent, for instance Assess claim versus Assess case. At the level of structure, languages like the Business Process Execution Language (BPEL) [1] allow the use of different constructs to express the same concept: for instance, a sequence between two atomic tasks can be expressed in BPEL using either a sequence node as parent of both tasks or by putting the tasks inside a flow node and defining a link between the tasks. At the level of behavior, an irrelevant difference can be for instance between a process that executes two tasks in parallel versus a process that executes two equivalent tasks in arbitrary, serial order, so one by one. If the tasks have short durations, both processes can be accepted as equivalent, so the difference in structure is irrelevant. Comparing the state spaces of both processes is the most reliable way to decide relevance. A wide range of similarity relations have been defined for comparing state spaces of processes [32], from trace equivalence to bisimulation.
- *Efficiency*: To establish an outsourcing relation, many possible providers may need to be compared. In addition, outsourcing can be done dynamically, meaning that a client wishes to establish very quickly (within seconds or milliseconds) an outsourcing relation with a client, for instance for the second scenario. Comparing many providers in a quick way implies that checking matching relations should be done efficiently. Comparing the state spaces of all processes, which results in relevant matches, is therefore typically not feasible. The size of the state space is in the worst case exponential in the size of the process model structure due to parallelism. This trade-off between quality (relevance) and efficiency in checking for a match is well known from other application domains [30].
- *Diversity*: The scenarios illustrate that in practice different types of matching relations can be used to find and establish a business process outsourcing relation between a client organization requesting a process and a provider organization offering a process.

Next, we define different matching relations and explore how the different similarity measures can be used to define these relations.

## 4.2 Matching and Similarity Measures

As explained in Sect. 4.1, in business process outsourcing, a process requested by a client needs to be matched to a process offered by a provider. Given that there

are various scenarios and requirements, we define the following types of matching relations [9]:

- *Exact* matching means both the requested process and the offered process are identical, so the provider process strictly complies with the process request. Exact matching supports the first scenario.
- *Plug-in* matching means the offered process can replace the requested process but may offer additional process behavior like extra tasks that were not requested. Plug-in matching supports the second scenario.
- *Inexact* matching means the offered process contains the main features of the requested process but is not exactly the same. The matched processes differ so much that a collaboration between the client and the provider is in general not possible. Therefore, an additional step is taken: either one of the organizations changes its process to ensure that at run time no logical errors like deadlock result, or one of the organizations uses a process adaptor [11] to compensate the differences in behavior. Thus, an inexact match assumes collaboration is possible. Inexact matching is useful in the third scenario.
- *Failure* occurs when no exact, plug-in, or inexact matching is possible, even though the two processes may share some tasks. Since no collaboration is possible, there is no match.

The latter relation is a possible output of a matching check. However, in the sequel, we ignore it, since it actually denotes the absence of the three other matching relations.

In the previous work [9], we have formalized these matching relations in terms of process trees, which abstract BPEL process models. The matching relations employ behavioral relations that abstract from irrelevant details from process trees. The matching definitions can be checked efficiently. Therefore, the three requirements in Sect. 4.1 are satisfied.

To elaborate these different types of matching relations in more detail and explore their connections with the similarity measures defined in Sect. 2, we consider a concrete outsourcing example. The process to be outsourced handles job applications in an organization (Fig. 5). The process has two stages. In the first stage, applicants are screened or rejected immediately. In the second stage, an applicant is selected and either hired or rejected. All rejected candidates, including the ones rejected in the first stage, are informed by letter in the second stage. The organization that employs the applicants wishes to outsource this job application process for

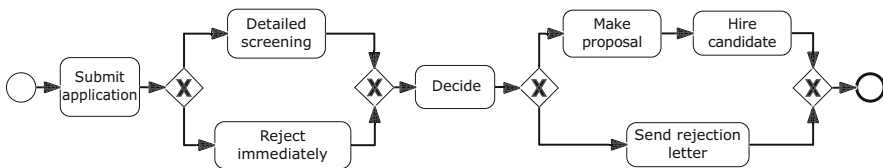
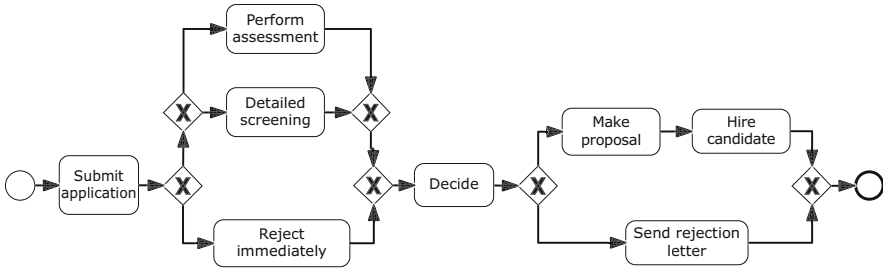


Fig. 5 A job application process



**Fig. 6** Another job application process

certain types of jobs. Consequently, it contacts potential providers of such a process, for instance headhunter companies, and it needs to compare the processes they offer to the process that is requested.

Different types of matching relations may be useful in the following situations.

*Exact Matching* The provider may exactly offer the process requested in Fig. 5. This does not imply the provider implements the actual tasks as specified in Fig. 5. The provider may in practice decompose certain tasks into more concrete tasks, for instance by interviewing an applicant as part of a Detailed screening task. This means that the processes shown in this subsection are actually *process views* that abstract private internal details from the actual performed processes [9]. Process views can be viewed as mutually agreed contracts in the context of business process outsourcing [10]. Process views have also been proposed to offer users personalized visualizations of a process [2].

A minor variation could be that both processes use slightly different labels to express the same task, for instance **Send application** instead of **Submit application**. This means there is a full match on the structure and behavioral measures, but a partial match on the activity (task) measure. In practice, activity mismatches are typically prevented by using a common domain ontology like the Supply Chain Operations Reference (SCOR) Model [29].

*Plug-In Matching* The provider process may offer tasks that are not requested, next to all the requested tasks. For instance, compared to the requested process in Fig. 5, the process model in Fig. 6 offers an extra task **Perform assessment** that can be chosen instead of task **Detailed screening**. Since the requested process is supported by this offered process, there is a plug-in match, since task **Detailed screening** can be always chosen. But, there is no exact match because of the extra task **Perform assessment**.

In this case, there is a full match on the activity measure for those tasks that are shared between both processes. However, there is a partial but relatively high match on the behavioral measures. The match on structural measures is high in this case. However, since similar behavior can be expressed in different ways in a process model, for instance using different combinations of gateways, the match on structural measures could also be low for plug-in matching.

**Table 1** The relation between matching relations for process outsourcing and different types of similarity scores

	Similarity scores		
	Activity-based	Structure-based	Behavior-based
Exact matching	High	Full	Full
Plug-in matching	Full	High, ..., low	High
Inexact matching	Full	High, ..., low	Low

*Inexact Matching* As a small variation on Fig. 6, consider that Perform assessment is preceded and succeeded by parallel instead of exclusive gateways. In that case, Perform assessment needs to be done always and cannot be bypassed by choosing Detailed screening. Consequently, there is an inexact match: the behaviors of the modified process and the original requested process are so much different that the original process cannot be realized by the provider process, since the provider performs an additional task.

For this example, and in general for inexact matching, there is a full match on the activity measure for both processes, but a lower match on the behavioral measure. The scores on structural measures can vary from low to high.

*Relation with Similarity Measures* In explaining the different types of matching, we have mentioned several types of similarity measures and used qualitative values such as full match, high match, and low match. However, as explained in Sect. 2, a similarity measure returns a quantitative similarity score for two process graphs expressed as a real (cf. Definition 2.2), which is often a value in the closed interval between [0..1]. If a score  $s$  is not in this interval, then  $s$  can be normalized into a value in the closed interval [0..1] by the formulae  $s - \min / \max - \min$ , where  $\min$  is the minimal and  $\max$  the maximal similarity score. We relate the qualitative values as follows to the (normalized) quantitative similarity scores: a normalized similarity score of exactly 1 is full, a score close to 1 is high, and a score close to 0 is low.

Using this convention, Table 1 summarizes the discussion of the different matching relations, thus making explicit what is the relation between the different types of matching on the one hand, and the different types of similarity measures and their possible scores on the other hand.

Based on the table and the discussion on which it is based, we draw the following conclusion. The activity-based measures require full or high similarity scores, since a business process outsourcing relation assumes that both organizations have a common understanding of the work that has to be done in the process. The behavior-based measure scores have a strong relation with the different types of matching. Structure-based measures focus on syntactic similarity of process models. They are mainly useful to distinguish different inexact matches.



### 4.3 *Post-Matching*

As mentioned in Sect. 4.2, matching the requested and offered business processes is only one of the steps toward the goal of establishing an agreement between a client requesting a process and a provider offering a process. After a match has been established, the following steps could be performed.

If an exact or plug-in match is established, still the compatibility of the process request and process offer needs to be checked. All matching relations are heuristics that give an answer that may not be accurate. To check accuracy for an exact match, formal verification tools like model checkers can check the state spaces of both processes for equivalence (exact) or simulation (plug-in) relations. Note that such verification tools are not efficient enough to replace matching relation altogether, since equivalence checking is computationally expensive due to the large state spaces of process models containing parallelism. Checking for exact or plug-in matching helps to filter the processes that need to be compared in a verification tool, so the use of heuristics and exact checks is complementary.

If the match is inexact, the process request and process offer need to be aligned in order to establish a viable outsourcing relation. A relaxed alignment relation between the two processes may already exist, for instance an isotactics relation [26]. An alternative way to achieve alignment is that one of the parties changes its process. Typically, this is the client, if the provider offers a standardized process. However, such a change may be difficult, since then most likely other internal processes that are connected to the changed process should be changed as well.

Another way to achieve alignment is to use an adaptor [11], which is a process that compensates differences in behavior between other processes. A client and provider process communicate by exchanging messages that are consumed and produced by tasks. An adaptor can intercept these messages and reorder them before sending. This way, the incompatibility between client and provider processes can be compensated. Adaptors can also be used to compensate differences in message structure [22]. Adaptors are somewhat similar to correlation models in the Process Querying Framework [24], since both aim to achieve alignment. However, communication aspects of processes are not considered by correlation models.

These post-matching steps do not belong to any component of the Process Querying Framework; in particular, they are different from the functionality that is covered by the interpretation component, which focuses on user understanding of the output of process querying. Post-matching focuses on establishing a runtime collaboration between processes, which is not the aim of the Process Querying Framework. This shows that for some application scenarios additional steps outside the framework are needed to actually put the results of process querying to practice.

#### ***4.4 Similarity Measures in Business Process Outsourcing***

Finally, we briefly reflect on the additional application of process similarity measures to business process outsourcing.

One of the trends in the area of business process outsourcing is that processes are treated as commodities [5], meaning they are highly standardized. This increases the likelihood of exact matching relations between clients and providers. In domain such as logistics (cf. the second scenario in Sect. 4.1) and supply chain management, ontologies like SCOR [29] play a key role to achieve this commoditization. Ontologies not only standardize tasks, but also the processes referencing these tasks. If there are minor variations of a standardized process, the different process similarity measures proposed in this chapter can be useful to rank the variations.

Other process similarity measures than the ones presented in this chapter can be used for process outsourcing. Often, BPEL [1] is used in this context. BPEL process models have a tree structure. Process trees are a suitable formalism to represent BPEL processes. Measures on BPEL process trees have been proposed to support efficient matching in the context of outsourcing [9]. These measures can be viewed as specializations of the behavior-based measures discussed in Sect. 2. Also, structure-based measures, based on edit distance, for BPEL have been used to rank different BPEL services that are retrieved for a requested BPEL process [14].

### **5 Process Similarity Querying and the Process Querying Framework**

The measures and indexing techniques that are part of process querying using process model similarity can be incorporated into the Process Querying Framework [25]. In the Process Querying Framework, process similarity querying is part of three components: the “indexing” component, the “filtering” component, and the “process querying” component. At this time, no optimization techniques are applied to process similarity querying.

The “indexing” component can be implemented using the two indexing techniques that are introduced: a tree-based index and an F-Net. Both indices introduce an indexing structure, a tree, and a network, respectively, in which the nodes point to process models from the process model repository. Given a query process model, the indexing structure can be traversed to find the node that represents the process model or process models that are most similar to the given query process model.

The “filtering” component can be implemented as part of a pre-processing step. In that case the indexing technique quickly traverses the repository and creates an “estimate” of process models that are potentially similar to the given query process model, in addition to process models that are certainly similar and process models that are certainly not similar to the given query process model. The benefit of this pre-processing step is that a process querying technique, which is computationally

more expensive than traversing an index, only has to compare the query process model to the filtered potentially similar process models and not to all process models.

The “process querying” component can be implemented for process similarity querying by implementing one or more of the process similarity measures that are introduced in this chapter. A process similarity measure compares two process models and returns a value between zero and one to indicate the similarity of those process models. Querying then works by iterating over all the process models in the repository and returning those process models that have a similarity to a given process query model that is above a certain threshold. The repository models are returned in descending order of their similarity to the query process model. Optionally, only filtered models have to be processed.

## 6 Conclusion

This chapter introduced techniques for process querying using process model similarity. Those techniques include process model similarity measures and indexing techniques. The process model similarity measures can be used to search within a process model repository for process models that are similar to a given query process model. The indexing techniques can be used to efficiently find those similar process models.

Three types of process similarity measures were introduced: activity-based measures, which are purely based on measuring the similarity of the activities that constitute the process models; structure-based measures, which also look at the arcs that connect the activities; and behavior-based measures, which look at the behavioral semantics of the process models. Previous research [7] has shown that structure-based measures have a slightly better performance than activity-based measures in terms of the quality of the search results, while behavior-based measures do not provide an additional benefit but are computationally expensive.

Two types of indexing techniques are introduced: a tree-based and an F-Net-based technique. Both techniques have been shown in previous research to substantially reduce the computational time that is required to perform a process similarity search. The tree-based technique has the benefit that it returns exact results. However, it does place constraints on the process similarity measures with which it can be used. The F-Net-based technique is more flexible but only does pre-processing, after which a selection of the returned process models may still have to be compared to the query process model in the non-indexed manner. Consequently, it is slightly slower than the tree-based technique.

We showed an application of the introduced techniques to a business process outsourcing scenario. In such a scenario, process similarity querying can be used to efficiently find potential business partners.

Previous research [7] has shown that process querying using process model similarity performs well when applied to a business process model repository that

uses relatively uniform terminology to label its activities and defines the process models at the same level of granularity. Terminology is considered uniform when the same word is used to describe the same concept in different process models. When non-uniform terminology is used or when process models are specified at different levels of granularity, performance drops dramatically. More research is required to develop techniques that also work in such situations.

Since a large number of different techniques for measuring process similarity already exist, in our opinion, the community would benefit most from a standardized corpus of data that can be used to evaluate these techniques and other techniques that may be developed in the future. Such a corpus would consist of a large collection of process models and query process models, for which it is known which process model is considered to be similar to which query model. The development of such a collection represents a lot of work but is very important for the research community to test and benchmark the various alternative techniques.

## References

1. Alves, A., Arkin, A., Askary, S., Barreto, C., Bloch, B., Curbera, F., Ford, M., Golland, Y., Guízar, A., Kartha, N., Liu, C.K., Khalaf, R., Koenig, D., Marin, M., Mehta, V., Thatte, S., Rijn, D., Yendluri, P., Yiu, A.: Web Services Business Process Execution Language Version 2.0 (OASIS standard) (2007)
2. Bobrik, R., Reichert, M., Bauer, T.: View-based process visualization. In: Business Process Management, 5th International Conference, BPM 2007, Brisbane, Australia, September 24–28, 2007, Proceedings. Lecture Notes in Computer Science, vol. 4714, pp. 88–95. Springer (2007). [https://doi.org/10.1007/978-3-540-75183-0\\_7](https://doi.org/10.1007/978-3-540-75183-0_7)
3. Brill, E.: A simple rule-based part of speech tagger. In: Proceedings of ANLC, pp. 152–155 (1992)
4. Bunke, H.: On a relation between graph edit distance and maximum common subgraph. *Pattern Recogn. Lett.* **18**(8), 689–694 (1997)
5. Davenport, T.H.: The coming commoditization of processes. *Harv. Bus. Rev.* **83**(6), 100–108 (2005)
6. Dijkman, R., Dumas, M., García-Bañuelos, L.: Graph matching algorithms for business process model similarity search. In: Proceedings of the International Conference on Business Process Management (BPM), vol. 5701, pp. 48–63 (2009)
7. Dijkman, R., Dumas, M., van Dongen, B., Kaarik, R., Mendling, J.: Similarity of business process models: Metrics and evaluation. *Information Systems* **36**(2), 498–516 (2011). <https://doi.org/10.1016/j.is.2010.09.006>
8. Dongen, B., Mendling, J., Aalst, W.: Structural patterns for soundness of business process models. In: Proceedings of EDOC, pp. 116–128 (2006)
9. Eshuis, R., Norta, A., Kopp, O., Pitkanen, E.: Service outsourcing with process views. *IEEE Trans. Serv. Comput.* **8**(1), 136–154 (2015). <https://doi.org/10.1109/TSC.2013.51>
10. Eshuis, R., Norta, A., Roulaux, R.: Evolving process views. *Inf. Softw. Tech.* **80**, 20–35 (2016). <https://doi.org/10.1016/j.infsof.2016.08.004>
11. Eshuis, R., Seguel, R., Grefen, P.: Synthesizing minimal protocol adaptors for asynchronously interacting services. *IEEE Trans. Serv. Comput.* **10**(3), 461–474 (2017). <https://doi.org/10.1109/TSC.2015.2467395>

12. Grefen, P., Aberer, K., Hoffner, Y., Ludwig, H.: CrossFlow: cross-organizational workflow management in dynamic virtual enterprises. *Int. J. Comput. Syst. Sci. Eng.* **15**(5), 277–290 (2001)
13. Grefen, P., Eshuis, R., Mehandjiev, N., Kouvas, G., Weichhart, G.: Internet-based support for process-oriented instant virtual enterprises. *IEEE Internet Comput.* **13**(6), 65–73 (2009)
14. Grigori, D., Corrales, J.C., Bouzeghoub, M., Gater, A.: Ranking BPEL processes for service discovery. *IEEE Trans. Serv. Comput.* **3**(3), 178–192 (2010). <https://doi.org/10.1109/TSC.2010.6>
15. Kunze, M., Weidlich, M., Weske, M.: Behavioral similarity – a proper metric. In: *Proceedings of Business Process Management*, pp. 166–181. Springer, Berlin, Heidelberg (2011)
16. Lécué, F., Delteil, A.: Making the difference in semantic Web service composition. In: *Proc. AAAI Conference on Artificial Intelligence*, pp. 1383–1388 (2007)
17. Lee, J., Kim, M., Lee, Y.: Information retrieval based on conceptual distance in is-a hierarchies. *J. Doc.* **49**(2), 188–207 (1993)
18. Levenshtein, I.: Binary code capable of correcting deletions, insertions and reversals. *Cybern. Control Theory* **10**(8), 707–710 (1966)
19. Li, L., Horrocks, I.: A software framework for matchmaking based on semantic Web technology. In: *Proc. International World Wide Web Conference (WWW) 2003*, pp. 331–339. ACM (2003)
20. Lin, D.: An information-theoretic definition of similarity. In: *Proceedings of ICML*, pp. 296–304 (1998)
21. Miller, G.: WordNet: A lexical database for English. *Commun. ACM* **38**(11), 39–41 (1995)
22. Motahari Nezhad, H.R., Benatallah, B., Martens, A., Curbera, F., Casati, F.: Semi-automated adaptation of service interactions. In: *Proceedings of the 16th International Conference on World Wide Web, WWW '07*, pp. 993–1002. ACM (2007). <https://doi.org/10.1145/1242572.1242706>
23. Paolucci, M., Kawamura, T., Payne, T., Sycara, K.: Semantic matching of Web services capabilities. In: *Proc. International Semantic Web Conference (ISWC'02)*. *Lecture Notes in Computer Science*, vol. 2342, pp. 333–347. Springer (2002)
24. Polyvyanyy, A., Ouyang, C., Barros, A., van der Aalst, W.M.P.: Process querying: Enabling business intelligence through query-based process analytics. *Decis. Support Syst.* **100**, 41–56 (2017). <https://doi.org/10.1016/j.dss.2017.04.011>
25. Polyvyanyy, A., Ouyang, C., Barros, A., van der Aalst, W.: Process querying: Enabling business intelligence through query-based process analytics. *Decis. Support Syst.* **100**, 41–56 (2017)
26. Polyvyanyy, A., Sürmeli, J., Weidlich, M.: Interleaving isotactics - an equivalence notion on behaviour abstractions. *Theor. Comput. Sci.* **737**, 1–18 (2018). <https://doi.org/10.1016/j.tcs.2018.01.005>
27. Porter, M.: An algorithm for suffix stripping. *Program* **14**(3), 130–137 (1980)
28. Resnik, P.: Using information content to evaluate semantic similarity in a taxonomy. In: *Proceedings of IJCAI*, pp. 448–453 (1995)
29. Stephens, S.: Supply chain operations reference model version 5.0: A new tool to improve supply chain efficiency and achieve best practice. *Inf. Syst. Front.* **3**(4), 471–476 (2001). <https://doi.org/10.1023/A:1012881006783>
30. Sycara, K.P., Widoff, S., Klusch, M., Lu, J.: Larks: Dynamic matchmaking among heterogeneous software agents in cyberspace. *Auton. Agent. Multi-Agent Syst.* **5**(2), 173–203 (2002). <https://doi.org/10.1023/A:1014897210525>
31. Uhlmann, J.: Satisfying general proximity/similarity queries with metric trees. *Inf. Proc. Lett.* **40**, 175–179 (1991)
32. van Glabbeek, R.J.: The linear time-branching time spectrum (extended abstract). In: Baeten, J.C.M., Klop, J.W. (eds.) *CONCUR '90, Theories of Concurrency: Unification and Extension*, Amsterdam, The Netherlands, August 27–30, 1990, *Proceedings*. *Lecture Notes in Computer Science*, vol. 458, pp. 278–297. Springer (1990). <https://doi.org/10.1007/BFb0039066>

33. Watters, C.: Information retrieval and the virtual document. *J. Am. Soc. Inf. Sci.* **50**(11), 1028–1029 (1999)
34. Weidlich, M., Dijkman, R., Mendling, J.: The ICoP framework: Identification of correspondences between process models. In: Pernici, B. (ed.) *Proceedings of the International Conference on Advanced Information Systems Engineering (CAiSE)*. *Lecture Notes in Computer Science*, vol. 6051, pp. 483–489 (2010). [https://doi.org/10.1007/978-3-642-13094-6\\_37](https://doi.org/10.1007/978-3-642-13094-6_37)
35. Yan, Z., Dijkman, R.M., Grefen, P.W.P.J.: Fast business process similarity search. *Distrib. Parallel Databases* **30**(2), 105–144 (2012). <https://doi.org/10.1007/s10619-012-7089-z>

# Logic-Based Approaches for Process Querying



Ralf Laue, Jorge Roa, Emiliano Reynares, María Laura Caliusco,  
and Pablo Villarreal

**Abstract** Today, logic-based formalisms are supported by mature languages, tools, and technologies for querying formal models. In this chapter, we show how process querying can be achieved using these technologies. The main idea is to transform the information of a business process model into a logic-based formalism for which existing query languages can be used. More specifically, we show how Prolog and ontologies together with SPARQL can be used to query BPMN process models.

## 1 Introduction

The main advantage of using visual process query languages is that users can define a query in a formalism that is similar to the modeling language they use for depicting the business process models. This means that a user can make queries without having to learn a new language. However, this simplicity has limits. For example, it can be difficult to express a combination of positive scenarios (what must happen) and negative scenarios (what must not happen) using standard modeling languages. It can also take some effort to develop tool support for specific visual query languages.

In this chapter, we show how process querying can be achieved using languages, tools, and technologies that already exist. The main idea is to transform the information that is contained in a business process model into a formalism for which existing query languages can be used. Those formalisms and corresponding query languages (we discuss Prolog, SPARQL, and graph databases) are grounded in formal logics. All information that is contained in a process model can be expressed in terms

---

R. Laue

Department of Computer Science, University of Applied Sciences Zwickau, Zwickau, Germany  
e-mail: [ralf.laue@fh-zwickau.de](mailto:ralf.laue@fh-zwickau.de)

J. Roa (✉) · E. Reynares · M. L. Caliusco · P. Villarreal

Facultad Regional Santa Fe, Universidad Tecnológica Nacional, Santa Fe, Argentina  
e-mail: [jroa@frsf.utn.edu.ar](mailto:jroa@frsf.utn.edu.ar); [ereynares@frsf.utn.edu.ar](mailto:ereynares@frsf.utn.edu.ar); [mcaliusc@frsf.utn.edu.ar](mailto:mcaliusc@frsf.utn.edu.ar);  
[pwillarr@frsf.utn.edu.ar](mailto:pwillarr@frsf.utn.edu.ar)

of logic facts. To be more specific: As business process models can essentially be described as graphs, we can use existing query languages that work well for graph-based structures. One such language is the logic programming language Prolog [17], which will be discussed in Sect. 3. To use this language, the information contained in the models is represented as logic predicates. An alternative way to represent information is the graph-based RDF language, which uses subject–predicate–object triples. For querying, we can use the *SPARQL Protocol And RDF Query Language* (or simply *SPARQL* for short) [40]. Established in 1997, it is widely used in semantic Web and knowledge management applications. We discuss model querying using SPARQL in Sect. 4.

The advantage of using an existing formalism is that we can re-use existing tools for process querying. Sophisticated algorithms (working with techniques such as caching intermediate results) are included out-of-the-box in existing Prolog systems and SPARQL implementations. A limitation is that the queries are based solely on the graph structure. Querying models according to their behavior (i.e., related to possible activity traces) given only their graph structure can be a hard problem.

In Sect. 2.1, we explain basic concepts of the modeling language BPMN and the soundness property. Section 3 shows how different kinds of queries can be expressed in Prolog. Section 4 discusses the same for semantic technologies, in particular OWL 2 and SPARQL. Sections 5 and 6, respectively, shows how logic-based technologies can be integrated into the Process Querying Framework [30] and provides a conclusion.

## 2 Background

In this section, we introduce basic concepts of Business Process Model and Notation [18] and the soundness property [38] as an example correctness criterion for business process models.

### 2.1 Business Process Model and Notation

We use the process modeling language Business Process Model and Notation (BPMN) [18] for illustrating the ideas of our approach, because this is a widespread standard for business process modeling. The most basic elements of this language are tasks, events, and gateways. Directed edges (arcs) depict the control flow between them. **Events** (something that happen during the lifetime of a business process) are represented as circles. An event is called a start event if it has no incoming edges. It is called an end event if it has no outgoing edges. **Tasks** are represented by rectangles with rounded corners. The control flow (called sequence flow in BPMN terminology) between the nodes (flow elements in BPMN terminology) is depicted by directed edges (arcs). The direction of such an edge



shows in which order the nodes have to be executed. **Gateways** can be used for forking and joining paths that have to be performed in parallel or (based on certain conditions) alternatively. There are two kinds of gateways: *Splits* have more than one outgoing edge, and *Joins* have more than one incoming edge. A gateway is represented by a diamond shape.

When used as a split, an *exclusive gateway* directs the sequence flow to exactly one of its outgoing branches. When used as a join, it awaits one incoming branch being completed before triggering the outgoing flow. A gateway of this kind (which we call *XOR-gateway*) is depicted by the  $\diamond$  symbol.

When splitting, a *parallel gateway* activates all outgoing branches; the activities on these branches are executed in parallel. When used as a join, a parallel gateway waits for all incoming branches to complete before triggering the outgoing flow (this behavior is called *synchronization* of the incoming flows). A parallel gateway (also called *AND-gateway*) is depicted by the  $\blacklozenge$  symbol.

Tasks may have one or more incoming and outgoing sequence flows. A task with more than one incoming sequence flow works as a join exclusive gateway. A task with more than one outgoing sequence flow works as a parallel gateway, activating all outgoing branches in parallel.

An inclusive gateway is something in-between an exclusive and a parallel gateway. When used as a split, some of the outgoing branches (but at least one) are activated. When merging, an inclusive gateway waits until all *active* incoming branches have been completed before triggering the outgoing flow. An inclusive gateway (or *OR-gateway*) is depicted by the  $\blacklozenge$  symbol.

Figure 1 shows a simple BPMN model depicting a process of organizing the participation at a trade show. The first task, after the start event (circle on the left), indicates that the deposit has to be paid to the show organizer. The  $\blacklozenge$ -gateways denote that tasks “Sign contract for booth space” and “Determine visual booth design” can be executed in parallel. When both tasks are completed, it will be decided how the product should be presented at the show. This decision is depicted by the  $\diamond$  symbol. If the product should be presented by means of a video, it is necessary to produce the video and then the process ends. If the product should be presented by means of a live demonstration, the human resources department must recruit part-time workers who will do the demonstration and then the process ends.

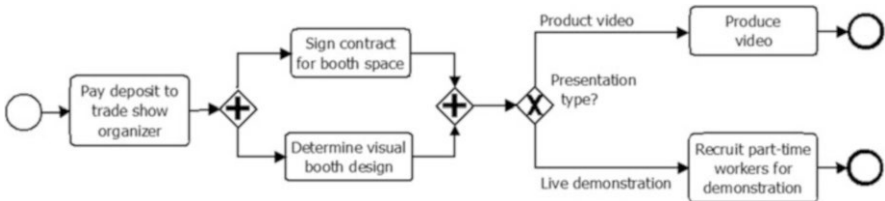


Fig. 1 Example BPMN model

BPMN has several more constructs, but those explained so far are sufficient for explaining our approach.

### 2.2 The Soundness Property

The diagram in Fig. 1 shows a correct BPMN model. Technically speaking, it fulfills the soundness property. The formal definition of this property can be found in [38]; for the sake of our discussion, it is sufficient to describe it informally by the following three properties:

1. *Option to complete:* When the process has been started from a start event, it is possible that it reaches an end event.
2. *Proper completion:* If an end event is reached, there are no remaining join gateways waiting for synchronization.
3. *No dead elements:* Each task in the model can potentially be executed.

Now let us assume that the company wants to change the process: The decision about the product presentation should be made directly after signing the contract for the booth space. The process modeler comes up with the diagram shown in Fig. 2.

Unfortunately, this diagram is wrong—the soundness property is violated: If it will be decided that the product is to be presented by means of a product video, the joining AND-gateway *pj* cannot synchronize. Such a situation—the process cannot make any progress because an element (in this case, the joining AND-gateway) is indefinitely waiting for incoming sequence flow (in this case, waiting for the upper incoming path to complete)—is called a *deadlock*.

Another way to introduce an error into a process model is shown in Fig. 3. Accidentally, the modeler used the wrong type of gateway for joining both paths together (⊗ instead of ⊕). The meaning of such a model is that no synchronization takes place to guarantee that the process does not proceed until *both* “Sign contract for booth space” and “Determine visual booth design” have been executed. Instead, the decision on how the product should be presented at the show will be executed

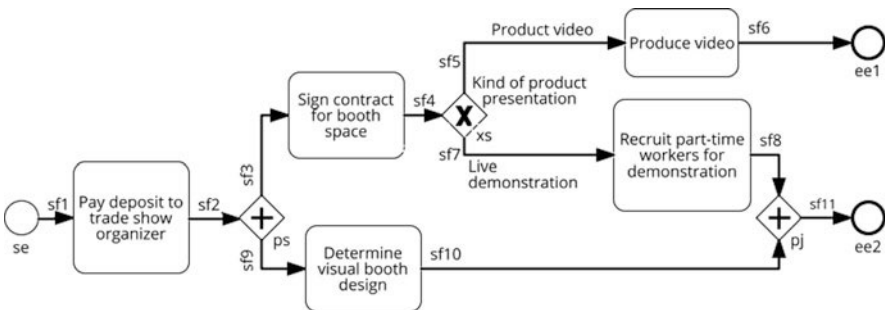


Fig. 2 Unsound BPMN model (deadlock)

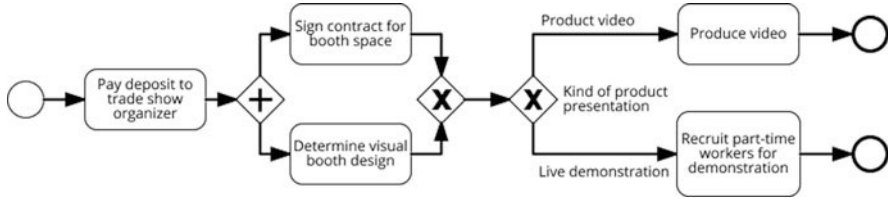


Fig. 3 Unsound BPMN model (lack of synchronization)

after completion of “Sign contract for booth space” (assuming that this is the first of both parallel tasks to be completed) and once again after the completion of “Determine visual booth design”. As a result, the decision of how the product should be presented at the show would be executed twice, which is not the desired behavior. This kind of control-flow problem is called *lack of synchronization*.

In the further course of this chapter, we will discuss how process querying for typical patterns of such soundness violations can help to identify (and hence to correct) this kind of modeling problems.

### 3 Process Querying Using Prolog

#### 3.1 Expressing the Model as Logic Facts

The information contained in the diagram in Fig. 1 can be expressed as a set of logic facts. Hereafter, we list those facts in the syntax of the logic programming language Prolog. Prolog has its roots in First-Order Logic. In Prolog syntax, the information from Fig. 1 is expressed by facts (statements that are known to be true) in the format *predicatename(argument1,argument2,...)*. Each node or edge in the graph can be represented by a unique id. For example, the fact that there is a task with id 1 can be expressed by the unary predicate *task(nodeid\_1)*.

Thus, the model from Fig. 1 can be represented by logic predicates in Prolog syntax as follows:

```

task(nodeid_1).
elementname(nodeid_1,'Pay deposit to tradeshow organizer').
task(nodeid_2).
elementname(nodeid_2,'Sign contract for booth space').
task(nodeid_3).
elementname(nodeid_3,'Determine visual booth design').
task(nodeid_4).
elementname(nodeid_4,'Produce video').
task(nodeid_5).
elementname(nodeid_5,'Recruit part-time workers
for demonstration').
and(nodeid_6).
and(nodeid_7).
    
```

```

xor(nodeid_8).
event(nodeid_9).
event(nodeid_10).
event(nodeid_11).
arc(edgeid_1,nodeid_9,nodeid_1).
arc(edgeid_2,nodeid_1,nodeid_6).
arc(edgeid_3,nodeid_6,nodeid_2).
arc(edgeid_4,nodeid_6,nodeid_3).
arc(edgeid_5,nodeid_2,nodeid_7).
arc(edgeid_6,nodeid_3,nodeid_7).
arc(edgeid_7,nodeid_7,nodeid_8).
arc(edgeid_8,nodeid_8,nodeid_4).
arc(edgeid_9,nodeid_8,nodeid_5).
arc(edgeid_10,nodeid_4,nodeid_10).
arc(edgeid_11,nodeid_5,nodeid_11).
condition(edgeid_8,'product video').
condition(edgeid_9,'live demonstration').

```

Those facts contain all the information that is described in Fig. 1.<sup>1</sup> Other BPMN elements (such as data objects, boundary events, etc.) can be expressed in a similar way. It is not difficult to write a transformation from the XMI-based serialization format of a BPMN diagram to the Prolog facts, e.g., by using an XSLT transformation. Note that in our example, we have dealt with a single model (more precisely: a single diagram) only. In the same way, we can get facts for a set of models (where one model can be depicted as a sub-process of another model). In this case, each fact should in addition contain a model identifier. For example, the first fact would read as `task(modelid_1,nodeid_1)`, etc.

Once we have transformed the information in the model into Prolog facts, we can use Prolog rules and queries for reasoning and querying. For example, it would be easy to find all models of which a given model is a sub-process.

Of course, the person who executes such queries is required to be familiar with Prolog syntax. Some will see this as a disadvantage. However, it is possible (and very advisable) to use a set of pre-defined rules, which simplify the compilation of Prolog queries. For example, we can introduce `startevent` as a new name of a predicate by defining that a start event is an event that does not have any incoming sequence flow:

```
startevent(X) :- event(X), not(arc(_,_,X)).
```

This Prolog rule is interpreted as follows: The unary predicate “X is a start event” holds true if X is an event and (denoted by the comma) there is no arc (with any id originating from any node) that leads to X.

This way, we can build a set of similar predicates that are close to the modelers’ language. These predicates can then be used in queries.

---

<sup>1</sup> With the exception of the information expressed by the layout of the diagram.

### 3.2 *Checking Syntactical Correctness*

A rather simple, but important, type of queries would be those that ensure the syntactic correctness of models by searching for models that violate the syntax rules of the modeling language. In the case of BPMN, one such rule requires that while in general start events are not mandatory, if the process model contains an end event, then there must also be at least one start event. To find violations of this rule, we would just have to look for models that contain an end event, but no start event with a query such as:

```
endevent_without_startevent :- endevent(_),not(startevent(_)).
```

In the same way, we can search for models that do not adhere to organization-wide style rules. In the case of BPMN such a style rule could stipulate, e.g., that only a subset of the BPMN notational elements is allowed or that always an AND-gateway must be used for depicting a parallel split instead of using multiple outgoing sequence flows from a flow element.

We included such Prolog-based semantic checks into the *Eclipse*-based modeling tools *bflow\** *Toolbox* (for Event-Driven Process Chains [EPCs]) [10] and *openOME* (for *i\** models) [23]. In the case of EPC models, we observed in an experiment that novice modelers who could make use of the checks had on average 1.7 syntax errors in their models, while novice modelers without such support had on average 8.8 syntax errors in the same task [24].

### 3.3 *Checking for Proper Layout*

Another type of queries can identify violations of layout conventions. This turned out to be useful for *i\** models. This language has been introduced for reasoning about the goals of process participants and the various ways how to achieve them [41]. *i\** uses the concept of a task decomposition. The decomposition of tasks into sub-tasks that can again have sub-tasks is depicted in a tree-like manner. A task that is decomposed into sub-task(s) should always be located above its sub-tasks(s); otherwise, it is very likely that the reader will misinterpret the model. Similar layout guidelines exist for other modeling languages as well. Some are language independent (such as “avoid parallel lines/arrows that are too close together”), and others refer to a given language (such as “end events should be the rightmost symbols in a diagram” in BPMN models).

For the purpose of checking layout-based rules, we need to have layout-related information stored in the Prolog fact base. In particular, the horizontal and vertical positions as well as the size of an element (such as a task) can be stored as facts:

```
task(id_1).
shape(id_1, 180, 250, 200, 100).
```

In this example,  $id\_1$  is the identifier of a task. The center of the shape depicting this task has the (x,y)-coordinates (180,250), a height of 100 units, and a width of 200 units. Based on this, one can write layout-related queries. For example, it is possible to find out by simple calculation whether two task boxes overlap in the diagram.

It must be mentioned that when it comes to layout-based queries, we face a difficulty. Standards for diagram interchange in languages such as BPMN define serialization aspects only for basic layout information (in particular, the size and position of the elements). However, other aspects such as text size or color are not considered in the standards. If we want to define queries related to those aspects, the vendor-specific extensions from the different modeling tools would have to be used. This means that such queries cannot be used independently of a modeling tool.

### 3.4 Locating Patterns Indicating a Soundness Violation

Prolog-based queries can be used far beyond of identifying violations of syntax, style, and layout rules. In [12], we show how pattern-based heuristics can identify control-flow problems such as deadlocks. Compared with model checkers, the pattern-based analysis was much faster and almost as exact as the results of model-checking tools that explore the state space of the models. Of course, a downside of a pattern-based approach is that (other than by using model checkers) we are unable to locate 100% of the control-flow problems. On the other hand, other than model checkers, the pattern-based approach does not suffer from the state-space explosion problem.

In an analysis of 984 business process models [12], we found that the most frequent patterns of control-flow problems were (i) a deadlock resulting from the combination of an (exclusive or inclusive) OR-split gateway with an AND-join gateway (see Fig. 2) and (ii) loop entries that were wrongly modeled by using an AND-gateway instead of an exclusive gateway, resulting in a deadlock at the loop entry.<sup>2</sup> For identifying those patterns quickly, we made use of Prolog rules. For example, in order to detect the combination XOR-split/AND-join, we aim to find an XOR-split gateway  $s$  and an AND-join gateway  $j$  with the property that there are two paths  $p_1$  and  $p_2$  from  $s$  to  $j$  such that  $s$  and  $j$  are the only nodes that are both in  $p_1$  and  $p_2$ . We call this a *match* between  $s$  and  $j$ , expressed by the Prolog predicate `match(s, j)`. This predicate is used not only for finding the deadlock pattern in Fig. 2, but also in the definition of various other patterns. A detailed discussion of such patterns indicating a soundness violation is done in Sect. 4.

For finding instances of patterns indicating a soundness violation in a model, the predicate `match` is executed often. As this means that paths from a split gateway

---

<sup>2</sup> The Prolog rules can be found in the source code of the open-source modeling tool *bflow\* Toolbox*, which can be downloaded from <http://www.bflow.org>.

to a join gateway have to be explored, this can take a lot of time in large models. We achieved a remarkable speed-up by two means. The first is the application of reduction rules. “Well-structured” parts of the model (such as simple sequences of tasks or, e.g., a combination of a parallel split [AND-split] and a synchronization [AND-join]) are known to be sound. For the purpose of detecting patterns as those shown in Figs. 2 and 3, such model fragments can be treated in the same way as a single task. For this reason, those fragments have been reduced to such a single task by repeatedly applying soundness-preserving reduction rules [22, 26]. The second measure to speed up the process is caching: All `match`-relations are calculated once and stored together with the model as additional Prolog facts. Of course, a recalculation can be necessary if a model changes. We found that both reducing and caching were very helpful for locating control-flow anti-patterns quickly.

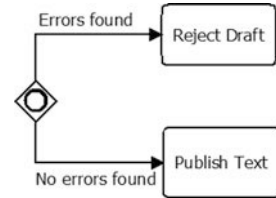
In a similar way, it is possible to define patterns for locating dataflow anti-patterns [37] such as data that are written, but never read.

### 3.5 Locating Incorrect and Ambiguous Labels

Prolog is very well-suited for natural language processing [28], and therefore we can also use it for analyzing the labels of tasks, events, and conditions that are placed at sequence flow arrows. The BPMN standard does not restrict these labels in any way. In most cases, they are written in unconstrained natural language. In [13, 25], we describe how possible ambiguities and problems in business process models can be located by searching for certain patterns in the labels of the model elements. Those patterns either indicate an error, a violation of a style rule, or a “bad smell” (a situation that could be problematic and needs further inspection). In order to find those patterns, we reason about the labels in the business process model. Three relations proved to be particularly useful for this purpose: the synonymy-relation, the antonymy-relation, and the happens-before-relation.

Two words or phrases are in the synonymy-relation if they have the same meaning. In fact, we defined this relation less strictly. For example, “to accept”, “to grant”, and “to approve” were considered as synonyms because each of these verbs refers to a positive decision. While synonymity has been defined for verbs as well as for noun phrases and adjectives, the other relations were defined only between two verbs: Two verbs are in the antonymy-relation if one of them has the opposite meaning of the other one (e.g., *forbid* / *permit*). Two verbs are in the “happens-before”-relation if the activity denoted by the first verb has to happen before the activity denoted by the second verb when both verbs refer to the same object. For example, “to produce” happens before “to ship”. For the purpose of defining the mentioned predicates, we compiled our own list of word relations after realizing that the well-known lexical database WordNet [7] does not meet our requirements. One important reason for this was that the antonymy relation in WordNet does not distinguish between different variants of antonymy such as complementary verbs

**Fig. 4** This situation should be modeled by using an XOR-gateway



(to accept / to reject) and pairs of verbs that show the same event from another perspective (to borrow / to lend) [8].

Once we have defined those relations between single words (or groups of words), it was possible to define Prolog predicates that work on the activity labels. The following predicates, which all evaluate to `true`, should be self-explanatory and can serve as an illustration of possible propositions that can be calculated for business process activity labels.<sup>3</sup>

```

synonym('The letter has been sent.', 'Letter sent').
synonym('Code contains errors', 'Code includes faults').
antonym('Test succeeded', 'Test failed').
antonym('customer record found', 'customer record not found').
happens_before('Pretest', 'Test').
happens_before('copy form', 'recycle form').
  
```

Using these predicates (and a few others), we can define interesting patterns for finding modeling problems related to the labels in natural language. An instance of one such pattern is shown in Fig. 4: Obviously, the inclusive OR-gateway (◇) should be replaced by an exclusive gateway (◇).

Other patterns that can be identified by inspecting the labels of the model elements deal with violations of naming conventions. A special case are anti-patterns that result from mixing natural language and control-flow logic [5]. Such labels contain phrases such as “as long as” or “if ...”. Such situations should be expressed explicitly by means of BPMN constructs instead of natural language [29]. Another type of linguistic anti-patterns in element labels deals with the use of weak words such as “to assist”. As it may be unclear what the process participant has to do in order to “assist” somebody, such words should be replaced by specific ones, exactly describing the actions to be performed.

All the label-related queries mentioned before were aimed to identify erroneous models. In a similar way, one can formulate queries for identifying correct, but incomplete models. Incomplete models can successfully be used for communication with domain experts who want to discuss the most important paths through a process, ignoring possible exceptional cases. Using such models that show only the so-called sunshine paths can be effective for the purpose of discussing about a

<sup>3</sup> In fact, our tool works for business process models with German labels. The English examples have been constructed for illustration purposes.



process. However, when the process needs to be understood in detail, the exceptional cases should be considered as well.

For the purpose of locating models that neglect the possibility of a negative outcome, we identified verbs and adjective phrases that usually indicate an activity with at least two possible outcomes. Examples for such verbs are “to test”, “to check”, or “to decide”. Actions described by those verbs can have a positive outcome (test successful / positive decision) and a negative one. In many cases, the same is true for other verbs as “to search” (what happens if nothing is found?) or “to compare” (what are the consequences if the objects being compared agree to each other / do not agree to each other?). With such a list of verbs, it is not difficult to build queries for finding activities that are not followed by an XOR-gateway depicting the necessary decision among two paths. We show a query of this type by using the predicate *is\_a\_test*, which concludes whether a task label describes a test or decision. After such a task, a branching should occur. A potential problem can be found by looking for tasks of this type that are not followed by an (exclusive or inclusive) OR-split gateway:

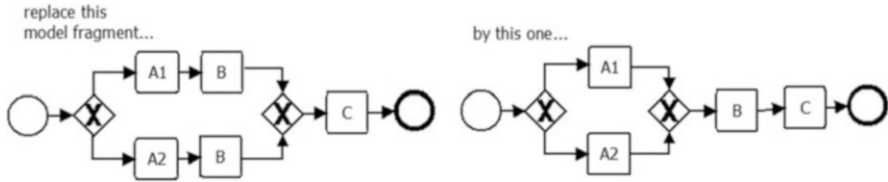
```
task(T), elementname(T, TName),
is_a_test(TName),
arc(T, Next), not((orsplit(Next); xorsplit(Next))).
```

In a similar way, labels of events can be analyzed for finding models in which errors have been disregarded. If the model contains an event with “positive” adjective phrases such as “successful”, “in time”, “without errors”, etc., it is reasonable to assume that the negative counterpart of this event should be modeled as well. By using a list of verbs that should lead to a decision and another list of “positive” adjective phrases, we are able to construct queries to find situations in which only the good case has been modeled.

### 3.6 Suggesting Process Model Refactoring

In addition to queries that identify actual modeling errors, we used the method for constructing queries suggesting a model refactoring. The aim of such a refactoring is to reduce the number of elements in the model, which improves the model’s understandability [11]. An example is shown in Fig. 5. A query to find such a situation could look for at least two tasks with the same label, which precede an XOR-join gateway.

```
xor(Xor),
task(Task1), task(Task2),
arc(_, Task1, Xor), arc(_, Task2, Xor),
Task1 @< Task2,
elementname(Task1, Name), elementname(Task2, Name).
```



**Fig. 5** Suggesting a model refactoring

The last line in this query ensures that *Task1* and *Task2* share the same label and the predicate  $\text{Task1}@\text{Task2}$  essentially guarantees that *Task1* refers to a different element than *Task2*.

### 3.7 Suggesting Process Improvements

In addition to locating problems in the *models*, queries can also be used for locating possibilities for improving the actual *process*. Once again, the verb classification is helpful. It can be used for identifying patterns of potential process weaknesses, for example, those discussed in [3]. However, a substantial difference between our approach and those in [3] is that we do not require to provide additional information about the type of an activity (such as “print something” or “document goes out”) at modeling time. Instead, the type of a task can be concluded from the verb (at least, in most cases). This way, it is possible to construct queries for patterns that point to possible process improvements. For example, one can construct queries for the following situations:

- An object is tested with a negative outcome. In the next step, some rework at the object is done—but it is not tested again.
- A document is sent by more than one mean (such as by fax as well as by letter).
- An activity of the type “receive document” is directly followed by another activity “forward document” (i.e., the model does not contain an activity where the document is read or the decision to whom it should be forwarded is made).
- A document is printed and later the printed-out document is scanned.

As can be seen from the examples, a model that matches the query does not necessarily have to be wrong, but the presence of one of the patterns is a strong indicator for a potential need of process improvements.

## 4 Process Querying Using Semantic Technologies

An ontology is a formal and explicit specification of a shared conceptualization that defines concepts used to represent knowledge and the relationships between the

concepts. It provides a formalized vocabulary of terms, specifying terms' definitions by describing their relationships with other terms in the ontology. An ontology covers a specific domain and it is shared by a community of users [14, 36].

The Web Ontology Language (OWL 2) is an ontology language with formal semantics that is designed to facilitate ontology development and knowledge sharing.<sup>4</sup> The semantics of OWL 2 is based on Description Logic, a fragment of First-Order Logic that has useful computational properties [2].

Semantic Web Rule Language (SWRL) is a language that extends OWL 2 with Horn-like rules. An SWRL rule is of the form of an implication between an antecedent (body) and a consequent (head). The intended meaning can be read as: whenever the conditions specified in the antecedent hold, then the conditions specified in the consequent must also hold.<sup>5</sup>

OWL 2 ontologies are primarily exchanged as directed labeled graphs stored as RDF documents. RDF is a standard model for data interchange on the Web. It extends the linking structure of the World Wide Web to name the relationships between things as well as the two ends of the link. This linking structure forms a directed, labeled graph, where the edges represent the named link between two resources, denoted by the graph nodes. Using this simple model, it allows structured and semi-structured data to be stored and shared across different applications. The performance issues constraining the management of large volumes of ontological data in RAM memory have given rise to several technologies for storing ontologies as RDF graphs.<sup>6</sup>

SPARQL is a language conceived to express queries across RDF data sources. SPARQL contains capabilities for querying required and optional graph patterns along with their conjunctions and disjunctions. It also supports extensible value testing and constraining queries by source RDF graph.<sup>7</sup>

All the aforementioned languages have been proposed by the World Wide Web Consortium (W3C) of the Semantic Web, but they have gradually evolved into a de facto standard for a broad spectrum of applications, and over time they have become widely known as part of the *Semantic Technologies*.

In the following subsections, we show how ontologies and graph-oriented databases can be used to query process models by means of SPARQL.

## 4.1 Querying Process Models Stored as Ontologies

A process model can be encoded as an OWL 2 ontology using the BPMN ontology proposed in [34]. The ontology of Raspocher et al. formalizes the BPMN meta-

---

<sup>4</sup> <https://www.w3.org/TR/owl2-overview/>.

<sup>5</sup> <https://www.w3.org/Submission/SWRL/>.

<sup>6</sup> <https://www.w3.org/RDF/>.

<sup>7</sup> <http://www.w3.org/TR/sparql11-query/>.

model by providing a formal description of the BPMN elements in terms of classes, attributes, and relations.

We make use of the BPMN process model in Fig. 2 to exemplify part of the ontological representation of a BPMN model as follows:

```

start_event (se)
end_event (eel)
task (A)
parallel_gateway (ps)
parallel_gateway (pj)
data_based_exclusive_gateway (xs)
sequence_flow (sfl)
has_sequence_flow_source_ref (sfl, se)
has_sequence_flow_target_ref (sfl, A)

```

To improve the readability of examples, we refer to activities “Pay deposit to trade show organizer”, “Sign contract for booth space”, “Produce video”, “Recruit part-time workers for demonstration”, and “Determine visual booth design” in Fig. 2 as activities A, B, C, D, and E, respectively.

The ontology encoding the process model depicted in Fig. 2 was implemented by means of Protégé, a free and open-source ontology editor, and Hermit, an OWL reasoner based on a novel hypertableau calculus, which provides an efficient reasoning algorithm [35].<sup>8,9</sup>

By encoding a process model as an OWL 2 ontology, several reasoning services can be used, such as query answering or compliance checking [34]. In previous work [31, 32], a clear application of this model is that we can use SPARQL queries to define heuristics for determining whether a process model meets a given set of properties. Heuristics are usually informally defined by examples with textual or graphical descriptions or with informal languages [12, 15, 21, 39]. To avoid ambiguities and different interpretations of what problem a given heuristics represents, and how it should be applied or implemented to detect behavioral errors in process models, we can formalize heuristics with ontologies, SWRL, and SPARQL. For example, based on the BPMN2 ontology for process models [34], we can define SWRL rules and SPARQL queries to find indicators for the presence of a deadlock, or to find out if a given task will ever be reached during execution.

We use the process in Fig. 2 as an example. As explained in Sect. 2, in this process there is a typical deadlock situation, where the AND-join *pj* cannot synchronize since the sequence flow originating from *sf8* does not complete. Next, we show a SPARQL query (called #H1) that can be used to detect this deadlock in the process model of Fig. 2.

```

SELECT ?psname ?pjname ?xsname ?eelname
WHERE {
  ?ps a bpmn2:parallelGateway.
  ?ps bpmn2:name ?psname.
  ?pj a bpmn2:parallelGateway.

```

<sup>8</sup> <http://protege.stanford.edu>.

<sup>9</sup> <http://www.hermit-reasoner.com/>.

```

    ?pj bpmn2:name ?pjname.
    ?xs a bpmn2:exclusiveGateway.
    ?xs bpmn2:name ?xsname.
    ?ee1 a bpmn2:endEvent.
    ?ee2 a bpmn2:endEvent.
    ?ee2 bpmn2:name ?ee2name.
    ?ps pack01:connected_to ?xs.
    ?xs pack01:connected_to ?ee1.
    ?xs pack01:connected_to ?pj.
    ?pj pack01:connected_to ?ee2.
    ?ee1 owl:differentFrom ?ee2.
}

```

The query presents a structural definition of the deadlock. It selects the names of the parallel gateways ( $ps$  and  $pj$ ), exclusive gateway ( $xs$ ), and end event ( $ee1$ ). We assume that a query evaluates to true if it returns a non-empty set, indicating as result the elements of the process that cause a deadlock. Otherwise, it evaluates to false. It is necessary to take into account that this query is specific to detect a deadlock caused by a given combination of elements. There could be other elements in a process model that are not part of this query, which may also lead to a deadlock. Other queries should be defined to detect those situations. To define this query, it is necessary to formalize (by means of SWRL Horn-like rules) the notions of *connection* and *path* between two BPMN elements. A *path* is defined recursively by applying the notion of *connection* in a separate ontology as follows:

- A flow element  $x$  is said to be connected to a flow element  $y$  if there is a directed edge (depicting sequence flow) from  $x$  to  $y$  (i.e., there is a direct connection between the elements).
- A flow element  $x$  is connected to a flow element  $z$  if there is a flow element  $y$  such that  $x$  is connected to  $y$  and  $y$  is connected to  $z$ .
- A *path* represents a sequence of sequence flows and flow nodes connecting two given flow elements.

In the context of the aforementioned example, a flow element represents an activity, an event, or a gateway.

The use of heuristics may lead to either false positive or false negative cases [32]. We explain this with the process models in Fig. 6. To define these models, we introduced slight modifications to the process model in Fig. 2. For example, in process  $FP01$ , if sequence flow  $sf5$  is activated, then activities  $C$ ,  $D$  and sequence flow  $sf8$  will also be activated and, hence,  $pj$  will not lead to a deadlock. We say that this is a false positive scenario for query  $\#H1$ , since it returns the names of the flow elements that satisfy the structural definition of the deadlock, but there is actually no deadlock. A similar situation occurs with processes  $FP02$ ,  $FP03$ , and  $FP04$ . In  $FP02$ , activity  $D$  and  $sf8$  will always be activated once activity  $E$  is executed and, hence,  $pj$  will not lead to a deadlock. In  $FP03$ , activity  $D$  and  $sf8$  will always be activated once the process starts, and hence,  $pj$  will not produce a deadlock. In  $FP04$ , activity  $D$  and  $sf8$  will always be activated once activity  $A$  is executed and, hence,  $pj$  will not lead to a deadlock.

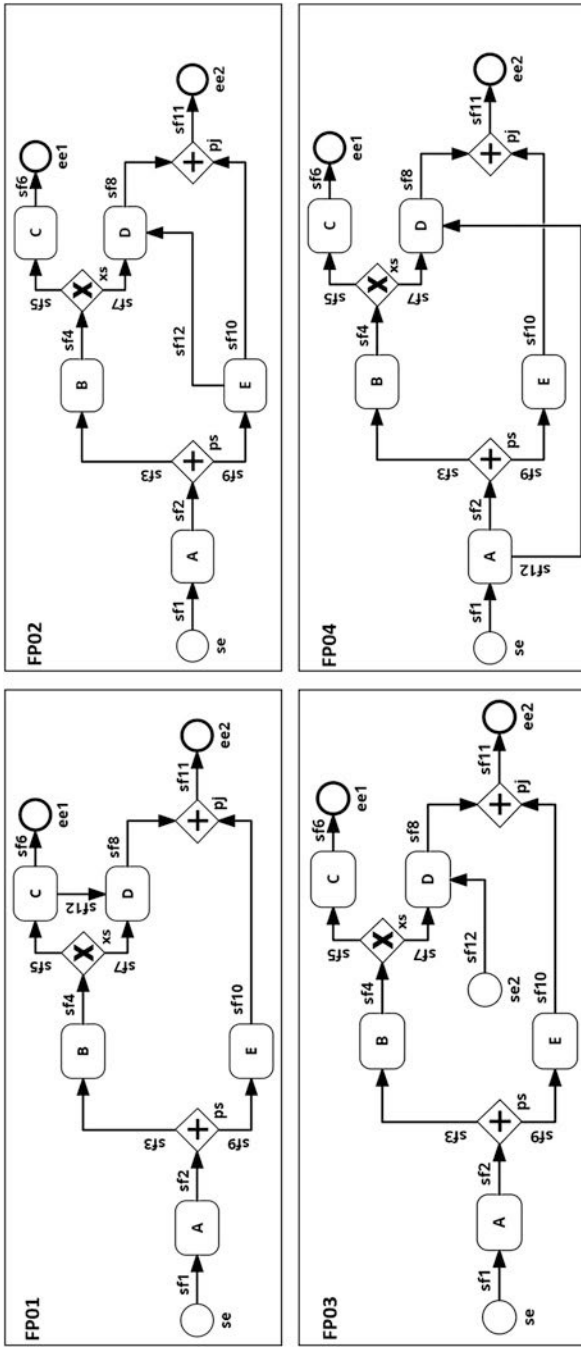


Fig. 6 False positive scenarios for #H1

The previous examples show scenarios where heuristics may fail by wrongly indicating an error (“false positive”). This raises the need to distinguish between queries for positive scenarios and queries for negative scenarios. For the case of error detection in process models, the former should allow detecting errors in process models and we call this the *problem heuristic*, whereas the latter should allow detecting scenarios where queries may fail and we call this *false positive heuristics* [32]. Based on these concepts, a likely error is detected in a process model when a problem heuristic is satisfied by the model and no false positive heuristic is satisfied by the same model.

In [31, 32], we applied these concepts to verify the behavior of BPMN models by formalizing a set of heuristics on an ontological specification of BPMN. In this way, the structure of BPMN models is represented as instantiations of an ontological specification of BPMN [34], extended with a set of rules that specify heuristics to detect behavioral problems in process models.<sup>10</sup>

## 4.2 Querying Process Models Stored in Graph-Oriented DB

While traditional technologies allow to perform some reasoning procedures over an ontology stored in the RAM memory of a computer, the management of large ontologies (e.g., with some gigabytes of size) in main memory becomes impractical. In the last decade, several development initiatives in query processing, access protocols, and triple-store technologies [1, 4, 6, 9, 16, 19, 20, 27, 33] have emerged to overcome this issue.

As a result of such advancements, today it is possible to store ontologies in graph-oriented databases (also known as triple-stores), which provide mechanisms for persistent storage and access to RDF graphs. This subsection shows how a graph-oriented database can be used to query process models encoded as OWL 2 ontologies by means of SPARQL.

The graph-oriented database selected in this work is the *community version* (free for non-commercial use) of *Stardog*. The last version of this tool (v 5.2.1) runs on Java 8 and supports the RDF graph data model, SPARQL query language, property graph model, and Gremlin graph traversal language, OWL 2 and user-defined rules for inference and data analytics, virtual graphs, geospatial query answering, and programmatic interaction via several languages and network interfaces.<sup>11,12,13</sup>

To show how to use a graph DB to query a BPMN process model to detect a deadlock, we use the BPMN model in Fig. 2. A Stardog database is created to store

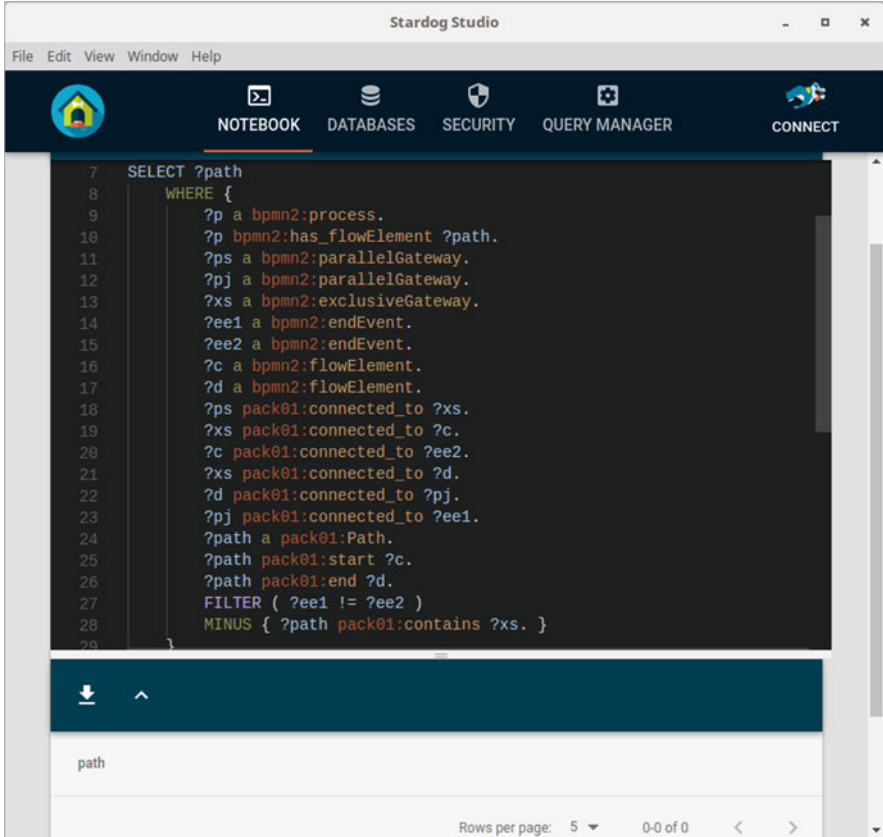
---

<sup>10</sup> Examples of the OWL ontology for BPMN and SPARQL queries to formalize problem and false positive heuristics of process models can be accessed via <http://dx.doi.org/10.17632/xkg32p2bs6.1>.

<sup>11</sup> <https://www.stardog.com/>.

<sup>12</sup> <https://www.java.com/en/download/faq/java8.xml>.

<sup>13</sup> <https://tinkerpop.apache.org/gremlin.html>.



**Fig. 7** Query and results to detect the false positive case *FP01* in Stardog Studio

(1) an ontology describing the BPMN meta-model proposed in [34], (2) an ontology comprising the rules for the specification of problem and false positive heuristics, (3) the ontologies specifying the negative scenarios presented before, and (4) the ontologies specifying the false positive cases associated to the negative scenarios.

The problem and the corresponding false positive heuristics are formalized as SPARQL queries. For a sake of space, just one of them is depicted here. Body and results of the SPARQL query defined to detect the *FP01* false positive occurrence of problem heuristic *#H1* are shown in Stardog Studio v0.1.0 interface (Fig. 7). The query returns the paths between flow elements *c* and *d*, except those that contain the exclusive gateway *xs*. See that *c*, *d*, and *xs* refer to variables in the query, not to the names of activities in the process model. If the result is a non-empty set, then the



false positive case *FPOI* is detected.<sup>14</sup> In Fig. 7, the query is applied to the process model in Fig. 2 and returns an empty set (see bottom of Fig. 7), which means that a false positive was not detected.

## 5 Process Querying Framework

The Process Querying Framework (PQF) provides components with generic functionality that can be selectively replaced to define a new process querying method [30]. Components can be active or passive. Active components refer to the actions performed by the process querying methods such as recording, modeling, or formalizing, among others, whereas passive components refer to the inputs and outputs of actions (objects and aggregations of objects) such as a process repository, process queries, or process querying instructions, among others [30].

The PQF is grouped in components for (i) designing process repositories and queries, (ii) preparing process queries, (iii) executing process queries, and (iv) interpreting results. It is possible to address several of the active and passive components of the PQF by means of the existing technologies presented in this chapter.

For the design of process repositories, it is possible to use logical facts, ontologies, and graph-oriented databases. Logical facts and ontologies can be seen as passive components that represent process models. Hence, besides event log, process model, correlation model, and simulation model, a process repository could be composed of logical facts and ontologies that formalize the structure, semantics, and domain knowledge of process models. The *Formalizing* component of the PQF takes a process querying instruction as input and produces a process query capturing the instruction by means of logical facts e.g., SWRL or SPARQL. The query intent for the examples provided in Sect. 4 is *Read Process*, since such queries can select process models satisfying structural (graph structures) based conditions. Besides this query intent, it is possible to define logical facts and SPARQL queries for creation, deletion, and update of process models.

A graph-oriented database provides the components for preparing and executing process queries with dedicated data structures that can speed up the execution of process queries. It also provides specific components for query optimization. In the examples provided in Sect. 4, the results of executing queries indicate the elements of the process that cause a deadlock or a false positive case. The component *Interpret* should take the results of a query as input and provide a human-readable response to the user, making the understanding of process query results easier.

---

<sup>14</sup> Examples of the OWL ontology for BPMN and SPARQL queries to formalize problem and false positive heuristics of process models can be accessed via <http://dx.doi.org/10.17632/xkg32p2bs6.1>.

## 6 Conclusion

The techniques described in this chapter allow process querying on process models based on existing logic-based technologies. First, we showed how Prolog can be used to encode a BPMN process model as a set of facts. These facts were used to define different types of queries. These queries allow to locate various kinds of patterns, e.g., violations of the BPMN standard, control-flow problems such as deadlocks, possible ambiguities, and problems in labels of model elements.

In addition to Prolog, we showed how ontologies and SPARQL can be used to query process models. In particular, we showed examples of how to define queries to find process models with potential deadlocks, and how to improve the precision of such queries by avoiding false positive cases. Since ontologies can have performance issues when dealing with large volumes of ontological data, we showed that ontological representations of business process models can be stored in the Stardog graph-based database, which provides mechanisms for persistent storage of and access to RDF graphs as a way to overcome performance issues. This type of storage supports SPARQL queries.

We also discussed how to address various components of the Process Querying Framework by means of the existing technologies presented in this chapter. After process models have been encoded as logical facts, queries to the process repository put into effect the query intent *Read Process* from the Process Querying Framework. As the techniques do not only allow querying logic facts but also modifying them, the query intents *Create Process*, *Update Process*, and *Delete Process* can be supported as well.

A key advantage of using existing formalisms for process querying is that it is possible to re-use existing tools. Sophisticated algorithms are included out-of-the-box in existing Prolog systems and SPARQL implementations. For both logic-based techniques (Prolog and ontologies), we showed by means of examples the usefulness of these technologies for process querying.

## References

1. Angles, R.: A comparison of current graph database models. In: 2012 IEEE 28th International Conference on Data Engineering Workshops, pp. 171–177 (2012). <https://doi.org/10.1109/ICDEW.2012.31>
2. Baader, F., Calvanese, D., McGuinness, D., Patel-Schneider, P., Nardi, D.: The Description Logic Handbook: Theory, Implementation and Applications. Cambridge University Press (2003)
3. Bergener, P., Delfmann, P., Weiß, B., Winkelmann, A.: Detecting potential weaknesses in business processes: An exploration of semantic pattern matching in process models. *Bus. Process Manag. J.* **21**(1), 25–54 (2015)
4. Cosmos: Azure Cosmos DB - Graph API (2018). <https://docs.microsoft.com/en-us/azure/cosmos-db/graph-introduction>

5. Delfmann, P., Herwig, S., Lis, L.: Konfliktäre Bezeichnungen in Ereignisgesteuerten Prozessketten — Linguistische Analyse und Vorschlag eines Lösungsansatzes. In: EPK 2009 Geschäftsprozessmanagement mit Ereignisgesteuerten Prozessketten. CEUR- Workshop Proceedings, vol. 554, pp. 178–194 (2009)
6. DSE: DSE Graph. DataStax Enterprise Graph (2018). <https://www.datastax.com/products/datastax-enterprise-graph>
7. Fellbaum, C. (ed.): WordNet: An Electronic Lexical Database (Language, Speech, and Communication). The MIT Press (1998)
8. Feltracco, A., Jezek, E., Magnini, B.: Opposition relations among verb frames. In: Proceedings of The 3rd Workshop on EVENTS: Definition, Detection, Coreference, and Representation, pp. 16–24. Association for Computational Linguistics (2015)
9. GraphDB: GraphDB. An enterprise ready semantic graph database, compliant with W3C standards (2018). <http://graphdb.ontotext.com/>
10. Gruhn, V., Laue, R.: Checking properties of business process models with logic programming. In: Modelling, Simulation, Verification and Validation of Enterprise Information Systems (MSVVEIS) 2007, pp. 84–93. INSTICC Press (2007)
11. Gruhn, V., Laue, R.: Reducing the cognitive complexity of business process models. In: IEEE International Conference on Cognitive Informatics (2009)
12. Gruhn, V., Laue, R.: A heuristic method for detecting problems in business process models. Bus. Process Manag. J. **16**(4) (2010)
13. Gruhn, V., Laue, R.: Detecting common errors in event-driven process chains by label analysis. Enterp. Modell. Inf. Syst. Archit. **6**(1), 3–15 (2011)
14. Guarino, N., Oberle, D., Staab, S.: What Is an Ontology? pp. 1–17. Springer, Berlin, Heidelberg (2009)
15. Han, Z., Gong, P., Zhang, L., Ling, J., Huang, W.: Definition and detection of control-flow anti-patterns in process models. In: 37th Annual IEEE Computer Software and Applications Conference Workshops (COMPSACW), pp. 433–438 (2013)
16. Hunter, J., Wooldridge, M.: Inside MarkLogic server (2011)
17. International Organization for Standardization: Standard ISO/IEC 13211-1:1995: Information technology - Programming languages - Prolog. Tech. rep. (1995)
18. International Organization for Standardization: ISO 19510 International Standard - Information technology - Object Management Group Business Process Model and Notation. Tech. rep. (2013)
19. Janus: JanusGraph. Distributed graph database. (2018). <http://janusgraph.org/>
20. Jena: Apache Jena - a free and open source Java framework for building semantic Web and linked data applications. (2018). <https://jena.apache.org/index.html>
21. Koehler, J., Vanhatalo, J.: Process anti-patterns: How to avoid the common traps of business process modeling. IBM WebSphere Developer Tech. J. **10**(2+4) (2007)
22. Laue, R., Mendling, J.: Structuredness and its significance for correctness of process models. Inf. Syst. E-Bus. Manag. **8**(3), 287–307 (2010)
23. Laue, R., Storch, A.: A flexible approach for validating  $i^*$  models. In: Proceedings of the 5th International  $i^*$  Workshop (2011)
24. Laue, R., Kühne, S., Gadatsch, A.: Evaluating the effect of feedback on syntactic errors for novice modellers. In: EPK 2009, Geschäftsprozessmanagement mit Ereignisgesteuerten Prozessketten, CEUR Workshop Proceedings (2009)
25. Laue, R., Koop, W., Gruhn, V.: Indicators for open issues in business process models. In: REFSQ. Lecture Notes in Computer Science, vol. 9619, pp. 102–116. Springer (2016)
26. Mendling, J., van der Aalst, W.M.P.: Advanced reduction rules for the verification of EPC business process models. In: SIGSAND-EUROPE. LNI, vol. 129, p. 129. GI (2008)
27. Neo4j: Neo4j graph platform (2018). <https://neo4j.com/>
28. Nugues, P.M.: An Introduction to Language Processing with Perl and Prolog: An Outline of Theories, Implementation, and Application with Special Consideration of English, French, and German. Springer Publishing (2010)

29. Pitke, F., Leopold, H., Mendling, J.: When language meets language: Anti patterns resulting from mixing natural and modeling language. In: Business Process Management Workshops - BPM 2014, Lecture Notes in Business Information Processing, vol. 202, pp. 118–129. Springer (2014)
30. Polyvyanyy, A., Ouyang, C., Barros, A., van der Aalst, W.M.P.: Process querying: Enabling business intelligence through query-based process analytics. *Decis. Support Syst.* **100**, 41–56 (2017)
31. Roa, J., Reynares, E., Caliusco, M.L., Villarreal, P.: Towards ontology-based anti-patterns for the verification of business process behavior. In: New Advances in Information Systems and Technologies, pp. 665–673. Springer (2016)
32. Roa, J., Reynares, E., Caliusco, M.L., Villarreal, P.D.: Ontology-based heuristics for process behavior: Formalizing false positive scenarios. In: Business Process Management Workshops - BPM 2016. Lecture Notes in Business Information Processing, vol. 281, pp. 106–117 (2016)
33. Robinson, I., Webber, J., Eifrem, E.: *Graph Databases*. O'Reilly Media (2015)
34. Rospocher, M., Ghidini, C., Serafini, L.: An ontology for the business process modelling notation. In: Formal Ontology in Information Systems - Proceedings of the Eighth International Conference, FOIS2014, September, 22–25, 2014, Rio de Janeiro, Brazil, vol. 267, pp. 133–146. IOS Press (2014). <https://doi.org/10.3233/978-1-61499-438-1-133>
35. Shearer, R., Motik, B., Horrocks, I.: Hermit: A highly-efficient owl reasoner. In: Proceedings of 5th OWL Experienced and Directions Workshop (OWLED 2008), vol. 432, p. 91 (2008)
36. Studer, R., Benjamins, V.R., Fensel, D., et al.: Knowledge engineering: principles and methods. *Data Knowl. Eng.* **25**(1), 161–198 (1998)
37. Trčka, N., van der Aalst, W.M.P., Sidorova, N.: Data-flow anti-patterns: Discovering data-flow errors in workflows. In: 21st International Conference on Advanced Information Systems Engineering (CAiSE), pp. 425–439. Springer (2009)
38. van der Aalst, W.M.P.: Verification of workflow nets. In: Application and Theory of Petri Nets 1997, 18th International Conference, ICATPN '97, Toulouse, France, June 23–27, 1997, Proceedings, pp. 407–426 (1997)
39. van Dongen, B., Mendling, J., van der Aalst, W.: Structural patterns for soundness of business process models. In: Enterprise Distributed Object Computing Conference, 2006. EDOC '06. 10th IEEE International, pp. 116–128 (2006)
40. World Wide Web Consortium: W3C Recommendation: SPARQL 1.1. Tech. rep. (2013)
41. Yu, E., Giorgini, P., Maiden, N., Mylopoulos, J.: *Social Modeling for Requirements Engineering*. MIT Press (2011)

# Process Model Similarity Techniques for Process Querying



Andreas Schoknecht, Tom Thaler, Ralf Laue, Peter Fettke,  
and Andreas Oberweis

**Abstract** Organizations store hundreds or even thousands of models nowadays in business process model repositories. This makes sophisticated operations, like conformance checking or duplicate detection, hard to conduct without automated support. Therefore, querying methods are used to support such tasks. This chapter reports on an evaluation of six techniques for similarity-based search of process models. Five of these approaches are based on Process Model Matching using various aspects of process models for similarity calculation. The sixth approach, however, is based on a technique from Information Retrieval and considers process models as text documents. All the techniques are compared regarding different measures from Information Retrieval. The results show the best performance for the non-matching-based technique, especially when a matching between models is difficult to determine.

## 1 Introduction

Companies and other organizations own lots of business process models and store them in so-called business process model repositories to describe and structure their business operations. These repositories can contain hundreds, or even thousands, of models (see, e.g., the collections mentioned in [14] and [25]), which makes sophisticated operations like conformance checking, duplicate detection, or the reuse of

---

A. Schoknecht (✉) · A. Oberweis  
Institute AIFB, Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany  
e-mail: [andreas.schoknecht@kit.edu](mailto:andreas.schoknecht@kit.edu); [andreas.oberweis@kit.edu](mailto:andreas.oberweis@kit.edu)

T. Thaler · P. Fettke  
German Research Center for Artificial Intelligence (DFKI) and Saarland University, Saarbrücken, Germany  
e-mail: [tom.thaler@dfki.de](mailto:tom.thaler@dfki.de); [peter.fettke@dfki.de](mailto:peter.fettke@dfki.de)

R. Laue  
University of Applied Sciences Zwickau, Zwickau, Germany  
e-mail: [ralf.laue@fh-zwickau.de](mailto:ralf.laue@fh-zwickau.de)

(parts of) models hard to conduct without automated support. Therefore, querying methods are used, for instance, to detect duplicate model fragments automatically.

Process model querying methods like the ones described in various chapters of this book can be used to find models containing a specified model fragment. This chapter, however, focuses on a different kind of querying approach, which is called *similarity-based search* [8]. When using similarity-based search, a process model is used as a query with the intention to find similar models in a repository.

Many similarity-based search techniques have been published (see, e.g., the survey in [24] for an overview). These techniques can be classified into two categories. Approaches from the first category are based on an underlying alignment between the activities or other nodes of the compared process models, which is also called *Process Model Matching* [2]. Before calculating a final similarity value, these approaches require an alignment between the model nodes. Techniques from the second category do not require such an alignment but use other means like process model metrics or document vectors created from the textual content of models.

This chapter provides an assessment of the performance of six techniques for similarity calculation of process models in the context of similarity-based search. Thereby, we extend our analysis described in [28] by comparing the matching-based similarity approaches with the *LS3* technique [23], which does not require a matching of process models for determining similarity values. Besides, we discuss further evaluation results of these approaches regarding essential measures from the Information Retrieval area such as Precision, Recall, F-Measure, and R-Precision [16].

The rest of this chapter is organized as follows: Sect. 2 provides fundamental definitions that are necessary to understand the subsequent sections. Afterward, we discuss related work and the relation of process model similarity to process querying in Sect. 3. The compared similarity techniques are then presented in Sect. 4. The setup of the comparative evaluation, the results of the evaluation, and the limitations of our analysis are discussed in Sect. 5. Finally, Sect. 6 provides a conclusion of this chapter and an outlook on future research.

## 2 Foundations

Fundamentals regarding business process models and the calculation of similarity values for process models are introduced in this section. First, Sects. 2.1 and 2.2 introduce definitions for process models and process model instances, respectively. Afterward, Sect. 2.3 describes Process Model Matching, which is an essential part for calculating a similarity value in most existing process model similarity techniques. We examine process model similarity in detail in Sect. 2.4. Finally, Sect. 2.5 provides background on the measures used in our evaluations.

## 2.1 Business Process Model

Similarity measurement in the context at hand primarily focuses on business process models, where it is distinguished between informal, semi-formal, and formal representations [5]. The models of interest typically have semi-formal or formal characteristics and are mostly represented as EPCs [10], BPMN diagrams, [19] or Petri Nets [18]. However, a business process model should not be understood as a model of a particular modeling language, but as a model of a particular model class describing business processes. Hence, an abstract definition of a process model covering the wide range of existing modeling languages is needed as the foundation. This definition requires an adequate generic representation of the graph structure and labeled nodes, as these are essential components of existing similarity measures.

Several generic formalizations of business process models are proposed in the literature, which generally address specific intentions. An analysis of these formalizations is described in [24], which resulted in the following definition.

**Definition 2.1 (Business Process Model)** A business process model  $M = (N, A, L, \lambda)$  is a directed graph consisting of three sets  $N$ ,  $A$ , and  $L$  and a partial function  $\lambda : N \rightarrow L$  such that

- $N = F \cup E \cup C$  ( $F, E, C$  pairwise disjoint) is a finite non-empty set of nodes with
  - $F \subseteq N$ : a finite non-empty set of activities (also called functions, transitions, tasks)
  - $E \subset N$ : a finite set of events
  - $C \subset N$ : a finite set of connectors (also called gateways)
- $A \subseteq N \times N$  is a finite set of directed arcs (also called edges) between two nodes  $n_i, n_j \in N$  defining the sequence flow.
- $L$  is a finite set of textual labels.
- $\lambda$  assigns to each node  $n \in N$  a textual label  $l \in L$ .

Although further node types such as organizational units and resources are relevant for describing business processes, they only play a minor role for existing similarity measurement. Hence, in this work, we abstract from them.

## 2.2 Business Process Instances

While business process models describe a business process on an abstract level, a business process instance represents an execution of a business process. An execution can either be observed in the real world or simulated. Business process instances are typically described as so-called traces (cf. [4]).

**Definition 2.2 (Trace, Trace Length)** A trace  $\sigma$  of a process model  $M = (N, A, L, \lambda)$  is a valid sequence of activities from  $F$ . A trace denotes the order in which the activities are executed. It is written as  $\sigma = \langle f_1, \dots, f_i, \dots, f_n \rangle$ ,

where  $1 \leq i \leq n$ .  $f_i$  may be equal to  $f_j$  with  $i \neq j$  as it is possible that an activity occurs more than once in a trace. The length of a trace  $|\sigma|$  is the number of activities in the trace.

Note that the term valid trace means that a trace cannot contain any sequence of activities but only sequences which can actually be executed, i.e., which are allowed by the semantics of the process model.

### 2.3 Business Process Model Matching

Structural correspondences of model elements are often the basis for calculating the similarity between business process models. In that sense, matching describes the procedure of taking two models as input, referred to as the source and target, and producing a number of matches between the elements of these two models as output based on a particular correspondence notion [21].

The more specific term Process Model Matching refers to the matching of single nodes, sets of nodes, or node blocks of one process model to corresponding elements of another process model based on criteria like similarity, equality, or analogy [26]. Referring to [31], it is generally distinguished between elementary and complex node matches, which are defined as follows:

**Definition 2.3 (Elementary/Complex Node Match)** A match  $m$  is denoted by a tuple  $(N_1, N_2)$  of two sets of nodes. A match  $(N_1, N_2)$  is called elementary match iff  $|N_1| = |N_2| = 1$  and complex match iff  $|N_1| > 1 \vee |N_2| > 1$ .

There are various approaches that approximate correspondences, respectively matches, between (sets of) nodes of models. A common technique is the consideration of (normalized) edit distances [7] of node labels like the Levenshtein distance [15]. Other approaches described in [2, 3] additionally apply techniques from the area of Natural Language Processing (NLP), thereby taking into account, e.g., semantic information of node labels concerning synonyms, homonyms, and antonyms.

### 2.4 Business Process Model Similarity

Similarity measures quantify the similarity between business processes models, while similarity is interpreted in different manners. Several dimensions of similarity have been identified and studied in the literature, e.g., the graph structure and state space of a process model, the syntax and semantics of process model labels, the behavior of a process or the similarity perceived by a human, as well as combinations of these dimensions [24].



Independently from the interpretation of similarity, a similarity value is usually expressed either on an interval or on a ratio scale. This provides the frame for a typical operationalization of business process model similarity in a metric space. Such a metric fulfills the properties of non-negativity, symmetry, identity, and triangle inequality [33]. However, as shown in [11], most of the existing process model similarity measures do not fulfill the abovementioned properties. Depending on the similarity measurement objective, there might be good reasons for violating particular properties. For example, if a similarity measure is used for searching process models, it might be acceptable to violate the symmetry property.

In the specific “part-of search” scenario, the search query would be a process model fragment. The similarity value should be one iff a process model contains the query fragment. On the contrary, when interchanging the query fragment and the process model containing the fragment, the resulting similarity value should be lower. Essentially, fulfilling the symmetry property is not a necessary requirement for that application.

## 2.5 Evaluation Measures

For the evaluation of the similarity-based search techniques presented in Sect. 5, Precision, Recall and F-Measure are used. Precision is defined as the fraction of relevant and obtained results (true positives  $TP$ ) to all obtained results ( $B$ ), Recall is defined as the fraction of relevant and obtained results to all relevant results ( $A$ ), and F-Measure is defined as the harmonic mean of Precision and Recall. Formally, these values are calculated as follows:

$$P = \frac{|TP|}{|B|}, \quad R = \frac{|TP|}{|A|}, \quad F = 2 \cdot \frac{P \cdot R}{P + R}.$$

In addition, we calculated R-Precision and Precision-at- $k$  values to evaluate ranked retrieval results. R-Precision measures Precision for a query with respect to the first  $|A|$  models, whereby  $|A|$  is the amount of relevant results only. R-Precision is therefore defined as the fraction  $\frac{|TP|}{|A|}$  with  $|TP|$  being the relevant and obtained documents. The difference to Recall is that not all retrieved results are taken into account, but only the  $|A|$  highest ranked results. Precision-at- $k$  does not use the  $|A|$  highest ranked models but considers the first  $k$  models instead. Hence, the following fraction is calculated:  $\frac{|TP|}{k}$ , again with  $|TP|$  being the relevant and obtained documents. For further details on all the used evaluation measures, we refer the reader to [16].

### 3 Process Model Querying and Similarity-Based Search

Process model querying approaches and similarity-based search techniques pursue the same goal: to provide users with a search functionality to satisfy their information needs more efficiently compared to manual browsing of model repositories. But while the goal may be the same, the used means are different. Process model querying approaches provide some kind of query language, which can be used to describe queries. These queries represent conditions which must be fulfilled by models from a repository to be contained in the query result.

Some query languages allow to find possible execution traces through textual query formulation, e.g., models that allow the execution of activity B after activity A. Other query languages allow for the graphical modeling of queries comparable to process modeling itself. In this context, a query is represented as a model fragment, which must be contained in a model to be returned as a query result. Typically, these query languages provide means to increase the variability of query formulation with special query elements like a path connector or wildcard nodes. Finally, some query languages incorporate Process Model Matching to widen the search scope of the queries.

Instead, similarity-based search uses an existing process model as a query and returns all models from the repository which have a similarity value with the query above a certain threshold. Therefore, similarity measures on process models are required to apply similarity-based search. Besides, most of the similarity-based approaches use Process Model Matching as the foundation for similarity calculation [24].

When comparing process model querying with similarity-based search, their commonality is the basic idea of providing users with search functionalities for process model repositories. Additionally, both approaches can rely on Process Model Matching for finding suitable query results. The main difference, however, is their search approach. While querying techniques use specific *query languages* to formulate a query, an *existing process model* is used as query input in the similarity-based search. Furthermore, querying techniques typically do not apply similarity measures on process models to widen the search scope but use other means like wildcard nodes.

Furthermore, similarity-based search can be related to the *Process Querying Framework* described in [20]. Similarity-based search for process models also requires some kind of process model repository for determining query results. Similarity-based search techniques require that such repositories contain business process models as one specific kind of behavior models mentioned in [20]. Additionally, for some techniques, other behavior models like event logs, execution traces, or alignments might be required or must be computed from the process models. A query itself is composed of a process model for which similar models should be detected in a repository. Besides a query model, it can be useful to provide a threshold value for specifying how similar resulting models should be compared to the query model. The intent of a query is always the same: retrieving

similar models. Hence, similarity-based search is not geared toward manipulating or deleting models.

Regarding the *Prepare* part of the Process Querying Framework, the performance of similarity-based search might be increased by indexing or caching mechanisms. For example, the efficiency of queries with the document vector-based LS3 approach [23] is increased when the document vectors of process models are stored in an index. In this case, the document vector generation has to be performed only for the query model. The document vectors for all models from the repository can be retrieved from the index and do not have to be generated for each query.

Yet, not all techniques might be equally well supported by the index structures. The calculation of matches between a query model and the models from the repository are not as easily indexable or cacheable as the document vectors from the previous example. This is due to the fact that the calculation of matches is always dependent on the query model and the possible result models. One difficulty is, for instance, that the matches between a query model and one process model from a repository cannot be used to infer matches between the query model and another process model without additional computations. The same applies for a new query model. Even if matches between other query models and the models from the repository are known, it is not possible to use these directly due to different terminologies in labels or model structures.

With this in mind, it is also difficult to envision a filter mechanism for matching-based similarity techniques, which can be used in the *Execution* part of the Process Querying Framework. If, for example, two process models  $pm_1$  and  $pm_2$  from a repository only have a low similarity score for a specific matching-based similarity technique and if the similarity value between a query model  $qm$  and  $pm_1$  is also low,  $pm_2$  cannot automatically be excluded from similarity calculation, i.e.,  $pm_2$  cannot be filtered. The reason is again that matching-based similarity techniques highly depend on the calculated matches. For the LS3 approach, however, a filtering of results could be applied based on the angles between the document vectors. For the two example models mentioned above,  $pm_2$  could be filtered from similarity calculation if the angle between  $qm$  and  $pm_1$  is too big and the angle between  $qm$  and  $pm_2$  would be even bigger.

## 4 Selection of Similarity Techniques

In order to evaluate the practical applicability and the limitations of the current state of research for similarity search, we need to identify and select proper similarity measurement techniques. This selection is based on the findings in [28]. As the analysis in [28] showed, most similarity techniques produce highly correlating similarity values. Hence, we only compare five of the eight approaches. The other three approaches were not considered, since they already showed a very high correlation with at least one of the selected ones. The selected techniques [1, 9, 12, 29, 32] differ in the dimensions used for similarity calculation and in their complexity so that the

**Table 1** Functional characteristics of all compared techniques

Dimension/reference	LS3 [23]	SSCAN [1]	CF [29]	FBSE [32]	LAROSA [12]	LCST [9]
Natural Lang.: Syntax		x	x	x	x	x
Natural Lang.: Semantics	x				x	
Graph structure		x	x	x	x	
Behavior			x			x
Model as text	x					
Model as element labels		x	x	x	x	x

selection should provide for a differentiated evaluation in the similarity-based search context. All of these similarity approaches use matches to calculate the similarity of process models. Therefore, we intentionally included another technique [23] in the evaluation, which does not use matches for similarity calculation.

Table 1 contains an overview of the techniques used in the evaluation. The calculation and setup details are described in the following subsections.

#### 4.1 Latent Semantic Analysis-Based Similarity Search

The *Latent Semantic Analysis-Based Similarity Search* (LS3) approach [23] is based on Latent Semantic Analysis [13], which is a technique from the Information Retrieval area for searching similar documents. The basic idea of LS3 is to construct so-called document vectors from process models. These document vectors form a Term-Document Matrix, in which each column represents a process model, i.e., a document vector, and each row represents a term<sup>1</sup> from all process models in a repository. The entries of the matrix contain weighted frequency values describing the weight of a certain term in a specific model.

Afterward, singular value decomposition is applied to decompose the constructed Term-Document Matrix of the process model repository into the product of three matrices. These matrices are used to construct another matrix with reduced dimensionality. The document vectors in the reduced matrix span a vector space which is used for calculating the similarity of process models. The similarity of two process models is thereby calculated as the cosine of the angle between their document vectors.<sup>2</sup> We did not include a classical Information Retrieval approach in our comparison as LS3 performed better in an experimental evaluation [22].

<sup>1</sup> In this context, a term should be understood as a word or a meaningful unit of words (e.g., statue of liberty).

<sup>2</sup> For calculating the similarity values, we used the code available at <https://github.com/ASchoknecht/LS3>.

## 4.2 *Similarity Score Based on Common Activity Names*

The similarity of two process models according to [1] (SSCAN) is calculated based on the number of identically labeled activities. We used the implementation proposed in the RefMod-Miner<sup>3</sup> to determine similarity values.

## 4.3 *Causal Footprints*

In the approach from [29] (CF), each process model is transformed into a so-called footprint vector, and the similarity of two models is determined as the cosine of the angle of their footprint vectors. A footprint vector consists of the activities of the process model as well as of two behavioral relations for each activity. The first relation contains all activities that are executed before an activity and the second relation contains all activities that are executed after that activity.

Hence, calculating the causal footprints requires a node matching between two process models. Although there is a proposal of a semantic node similarity measure, the used implementation from <http://rmm.dfki.de> considers two activities as a match if both have the same label.

## 4.4 *Feature-Based Similarity Estimation*

The technique described in [32] (FBSE) uses the syntactical natural language dimension as well as the graph-structural dimension to determine similarity values. Regarding the syntactical dimension, a Levenshtein distance-based similarity value between activity labels is calculated. For the graph-structural dimension, five roles (start, stop, split, join, and regular) are used to characterize an activity. The graph-structural similarity is then based on the common roles of two activities (so-called role feature similarity). Two activities are considered as equivalent if both the syntactic label similarity and the role feature similarity surpass an individual threshold. Finally, the similarity between two process models is defined as the ratio of equivalent activities to the overall number of activities in both models.

We used the implementation from <http://rmm.dfki.de> to determine similarity values. Thereby, the thresholds were set as proposed in the original paper, and the resulting similarity matrix was optimized using the greedy algorithm described in [32].

---

<sup>3</sup> RefMod-Miner as a Server: <http://rmm.dfki.de> and Code on GitHub: <https://github.com/tomson2001/refmodmine>.

## 4.5 *La Rosa Similarity*

The similarity calculation of [12] (LAROSA) is based on the graph-edit distance similarity described in [6]. The basic idea of the technique is to determine matches between two process models and to additionally consider the graph structure of models by calculating a graph-edit distance. The matches in [12] are based on the Levenshtein distance of the node labels and on a linguistic similarity measure using a lexical database. The greedy algorithm in [6] for finding the optimal graph-edit distance has been used with the original implementation. The parameter values were set as described in [12].

## 4.6 *Longest Common Sets of Traces*

The approach proposed in [9] (LCST) uses the traces of two process models  $M_1$  and  $M_2$  to quantify their similarity. Therefore, the two components trace compliance degree  $cd_{trace}(\sigma_1, \sigma_2)$  and trace maturity degree  $md_{trace}(\sigma_1, \sigma_2)$  are used, whereby  $\sigma_1$  is a trace of  $M_1$  and  $\sigma_2$  is a trace of  $M_2$ . The trace compliance degree covers the extent to which a process adheres to ordering rules of activities, while the trace maturity degree covers the extent to which the activities of the other model are recalled. Both components are defined based on the length of their longest common subsequence  $lcs$ , such that  $cd_{trace}(\sigma_1, \sigma_2) = \frac{|lcs(\sigma_1, \sigma_2)|}{|\sigma_2|}$  and  $md_{trace}(\sigma_1, \sigma_2) = \frac{|lcs(\sigma_1, \sigma_2)|}{|\sigma_1|}$ . Based on that, the compliance and maturity degree between two process models are defined as the sum of the maximum trace compliance and trace maturity degrees. Finally, two components are used to express in how far the traces of one model are reflected by the traces of another model.

To provide a comparable similarity value, the average of both components is calculated and interpreted as the final similarity value. The matches required by this approach are determined using the Levenshtein distance-based similarity calculation between two activity labels with a minimum threshold of 0.9. We used the implementation from <http://rmm.dfki.de> to determine similarity values.

## 5 Evaluation

The selected process model similarity measurement techniques are evaluated in this section. First, we present the used data collection (Sect. 5.1) and describe the evaluation design (Sect. 5.2). Afterward, the evaluation results are presented in Sect. 5.3, followed by a discussion of the results and the limitations in Sect. 5.4.

## 5.1 Dataset

The dataset for the comparison is based on the model collection used in [28]. The idea is to conduct an experimental analysis of similarity measures to characterize their behavior in specific application scenarios. For that purpose, one can distinguish laboratory and field investigations. In laboratory investigations, the process models are (possibly synthetically) generated in a controlled environment, while in field investigations, they are generated by human modelers. Since the results of a laboratory investigation cannot easily be transferred to the field, the field setting should be considered as well. We finally use three different groups of samples with different characteristics which are partially taken from a large process model corpus [27]. In contrast to [28], we added four additional laboratory model sets from Camunda™ training sessions. All used model sets with their specific characteristics are described below:

1. **Field models:** To develop these models, no restrictions regarding the labeling of model elements were given to the modeler(s). Thus, in these models, equal or similar aspects might be modeled in a different manner and expressed with different words. A dataset, containing such models from the domain of university admission (9 models) and the domain of birth registration (9 models), is provided in [2].
2. **Models from controlled modeling environments:** Models are created in a controlled environment, wherein different modelers independently model the same process based on a natural language text description. As a terminology is provided in the textual description, it is assumed that this terminology is used by the modelers as well. Student exercises<sup>4</sup> (18 models) serve as an adequate dataset. Additionally, models from Camunda™ training sessions<sup>5</sup> were included in this group (40 models). An analysis based on this dataset covers a laboratory investigation.
3. **Mined models:** The process models in this group are derived using process mining techniques. Thus, the node labels are linguistically harmonized and are (1) unambiguous and (2) consistent over the whole collection (matching problem is essentially evaded as model elements representing the same real-world activity are labeled identically). The models from Dutch governance presented in [30] fulfill this requirement (80 models). However, one can argue whether they are synthetically created in a laboratory sense or, as the processes are executed in the real-world, whether they are derived from the field.

The overall model collection contains 156 distinct models, which were compared to one another in every possible combination. This leads to similarity calculations for 24,336 business process model pairs; both directions were checked as some of the similarity measures are not symmetric (pseudo-metrics).

---

<sup>4</sup> Model set “Exams” is available in the model repository at <http://rmm.dfki.de>.

<sup>5</sup> The original models can be retrieved from <https://github.com/camunda/bpmn-for-research>.

## 5.2 Query Results

Before we were able to determine the query results for each similarity measurement technique, we needed to calculate the similarity values for all model pairs with each technique. The interval used for the similarity values was  $[0, 1]$ , with higher values meaning more similar and lesser values meaning less similar.

The similarity values, calculated with the above mentioned techniques, were used as the foundation for calculating the evaluation measures in the second step. Therefore, a gold standard containing the relevant models to a specific query model was needed to determine Precision, Recall, and F-Measure values. As the underlying processes of the model collection are different, we decided to use all models related to a specific process as the relevant models. For example, when one of the University Admission models was used as a query model, we considered all of the nine University Admission models to be the relevant models for this specific query. Note that we did not remove the query model from the model dataset for querying as we also wanted to analyze how the search approaches handle models identical to the query model.

To finally calculate Precision, Recall, and F-Measure values, we used a threshold value  $\theta$  on the similarity values. Only models having a similarity value equal to or above the threshold value with respect to a query model were deemed as a query result.

Regarding the R-Precision and Precision-at-k evaluation measures, we did not need a threshold value. For a query model, we simply ranked all models in descending order according to their similarity values. Afterward, we calculated Precision of the first  $|R|$  results to determine R-Precision. This means, for instance, that we determined the R-Precision for one of the University Admission models based on the nine models with the highest similarity values compared to this model. The first nine models are used because the gold standard for one of the University Admission models contains nine models. We also calculated the Precision-at-5 values by calculating Precision based on the 5 highest ranked models. We decided to use  $k = 5$  for the Precision-at-k measure to examine the first results, which are most likely to be viewed by a user of such a search functionality. Besides, we used quite a low value for k as the amount of relevant models was mostly nine or ten. Only for the models from the student exercise, 18 relevant models were available.

## 5.3 Evaluation Results

Table 2 shows the results for the similarity-based search experiment described in the previous section. The first two parts contain the macro and micro average values for Precision, Recall, and F-Measure. The macro average calculates the average over all queries, while the micro average is calculated by summing up true positives and the amount of retrieved and relevant results before computing Precision, Recall, and



**Table 2** Statistics for query results (P: Precision, R: Recall, and F: F-Measure)

		LS3 [23]	LCST [9]	FBSE [32]	CF [29]	SSCAN [1]	LAROSA [12]	
Macro	P	AVG	0.92	0.81	0.26	0.90	<b>0.96</b>	0.87
		STD	0.18	0.31	0.19	0.20	0.15	0.26
	R	AVG	<b>0.89</b>	0.55	0.59	0.64	0.56	0.79
		STD	0.21	0.43	0.24	0.40	0.43	0.30
	F	AVG	<b>0.87</b>	0.47	0.33	0.66	0.60	0.80
		STD	0.19	0.34	0.19	0.36	0.39	0.27
Micro	P		0.86	0.47	0.20	0.87	<b>0.95</b>	0.88
	R		<b>0.89</b>	0.52	0.59	0.59	0.53	0.78
	F		<b>0.87</b>	0.49	0.30	0.71	0.68	0.83
R-Precision			<b>0.95</b>	0.50	0.37	0.82	0.78	0.88
Precision-at-5			<b>0.99</b>	0.62	0.56	0.91	0.93	0.93
F = 1			<b>79</b>	20	0	52	54	43
Threshold			$\theta = 0.79$	$\theta = 0.47$	$\theta = 0.91$	$\theta = 0.64$	$\theta = 0.42$	$\theta = 0.27$
Calculation time			2s	> 1d	54min	~1d	12min	2h

F-Measure. The third part contains the results for R-Precision, Precision-at-5, as well as the amount of queries with the F-Measure value of 1. Finally, the last row contains the threshold value, which maximized the macro average F-Measure value of each considered search technique. The best result for each evaluation measure is marked bold.

Regarding the unranked Precision, Recall, and F-Measure, four search techniques showed very good results. LS3, CF, SSCAN, and LAROSA reached (at least for some of the measures) high results. SSCAN got the highest Precision value (0.96), which is expected, as this approach counts identically labeled nodes. Thus, there is a high probability that two models with many identically labeled nodes, in fact, describe the same process. With respect to the Recall and F-Measure values, LS3 reached the highest scores of 0.89 and 0.87, respectively. However, for CF and SSCAN, Recall is the critical measure as their values are significantly lower (0.64 and 0.56). While LAROSA never reached the highest values, all evaluation values are comparably high. Additionally, LAROSA also received the second-highest scores for R-Precision and Precision-at-5. Only LS3 reached higher values for these ranked evaluation measures. Besides, CF and SSCAN again got good to very good values. The LCST and FBSE search techniques, however, reached only low values for all the considered evaluation measures.

Finally, LS3 and SSCAN reached outstanding results. Both approaches show a very good performance not only in terms of calculation complexity and calculation time but also regarding the evaluated measures. LS3 shows the best F-Measure values overall as well as the best R-Precision and the best Precision-at-5. On the contrary, SSCAN reaches the best Precision. Depending on the actual scenario, it might be meaningful to decide on the particular goal. A high Precision stands for a high probability that a query result is relevant in terms of the search argument, while

**Table 3** Micro average values regarding easy and hard matching of models (P: Precision, R: Recall, and F: F-Measure)

		LS3 [23]	LCST [9]	FBSE [32]	CF [29]	SSCAN [1]	LAROSA [12]
Easy	P	0.75	<b>0.95</b>	0.25	0.94	0.94	0.82
	R	0.97	0.63	0.64	0.97	0.98	<b>0.99</b>
	F	0.85	0.75	0.36	<b>0.96</b>	<b>0.96</b>	0.90
Hard	P	<b>1.00</b>	0.28	0.17	0.69	<b>1.00</b>	0.98
	R	<b>0.82</b>	0.42	0.54	0.25	0.12	0.60
	F	<b>0.90</b>	0.33	0.26	0.37	0.21	0.74

a high Recall ensures that a great fraction of the expected results is found. Hence, both Precision and Recall are valid isolated criteria for queries, but an aggregation (F-Measure) of them makes sense for unknown scenarios as well.

In contrast to the lightweight LS3 and SSCAN approaches (in terms of the estimated calculation time<sup>6</sup>), CF and LCST are very expensive to calculate. Both require a derivation of traces or parts of traces to calculate a similarity value. Since the state space of a process model can explode under certain circumstances, such a calculation might even become impossible. Against that background, there is a risk of running into a situation where the approaches cannot be applied. This is indicated by the calculation times mentioned in Table 2, although these values do not allow to derive any reliable statement on performance. In fact, it cannot be expected that the implementations are optimized with regard to performance. Most of them (all but LS3) perform a pairwise comparison, which require a separate loading and parsing of the analyzed model files for each pairwise calculation. We tried to eliminate that problem by performing the task of loading and parsing in isolation, which was possible for all approaches except of LAROSA. Moreover, all approaches considering models as elements need to interpret the source data and instantiate each single node as a dedicated object. Finally, in the best case, the calculation times only state an indication of performance but do not necessarily allow to derive a reliable statement about a practical applicability.

Table 3 shows the micro average results for Precision, Recall, and F-Measure divided into two categories based on the matching difficulty. In the easy part, only the models from the Dutch municipalities dataset are included (80 models). Calculating a matching between these models is simple as the same real-world activities are labeled identically. The hard part contains the models from the field and controlled modeling categories (76 models).

The numbers from Table 3 highlight one essential difference between the LS3 approach and the three top-ranked matching-based search techniques CF, SSCAN, and LAROSA. For the easy part, LS3 is outperformed by the three matching-

<sup>6</sup> The actual calculation time depends on the implementation. In case of a mapping-based similarity calculation (which is the case for all evaluation measures expect of LS3), the calculation of the mapping needs to be considered as well.

based techniques regarding Precision and F-Measure. Recall values are very high for all the four approaches. This clearly shows that especially CF and SSCAN can calculate very good query results in case of easy matching. Yet, the evaluation shows an inversing result for the hard part. In the case of a difficult matching, the LS3 approach outperforms the matching-based techniques. Especially, the Recall values for CF and SSCAN drop to very low values. The problem for both approaches is determining matches as they use simple matching calculations (both do only match in the case of identical labels).

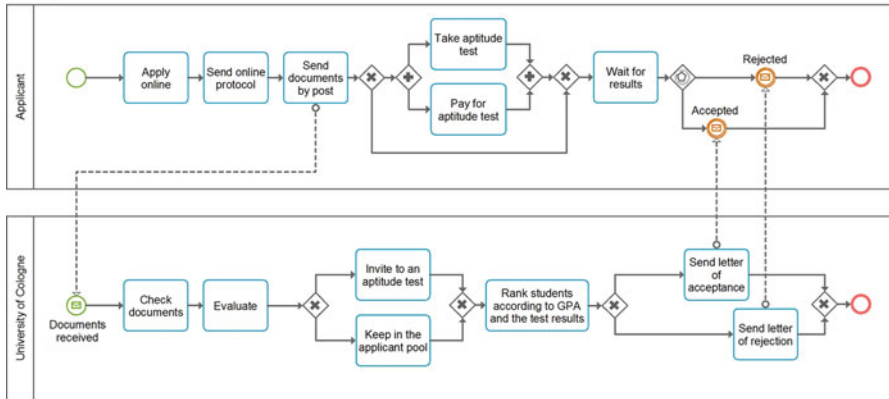
## 5.4 Discussion and Limitations

A limitation of the presented evaluation, and also for all other similar evaluations, is the choice and the size of the model corpus (the used dataset). Since neither the overall set of existing process models is available in a single corpus nor the overall number of existing process models is known, it is not possible to select a specific number of models randomly (which would be necessary to determine the statistical significance). Instead, as mentioned above, we selected models that are (1) appropriate for the evaluation scenario and (2) heterogeneous. Appropriate in this case means that a search in the model set is meaningful—there exist models with a naturally given similarity, so that the expected query results can be determined. Heterogeneity describes the character of the models caused by their origin, i.e., the domain, the modelers background or his modeling experience. This highly influences the complexity of the matching problem: as mined models are automatically derived, a matching problem by itself does not exist. Linguistically similar labels are probable, if the models are designed based on a consistent textual description—they are rather improbable, if this is not the case.

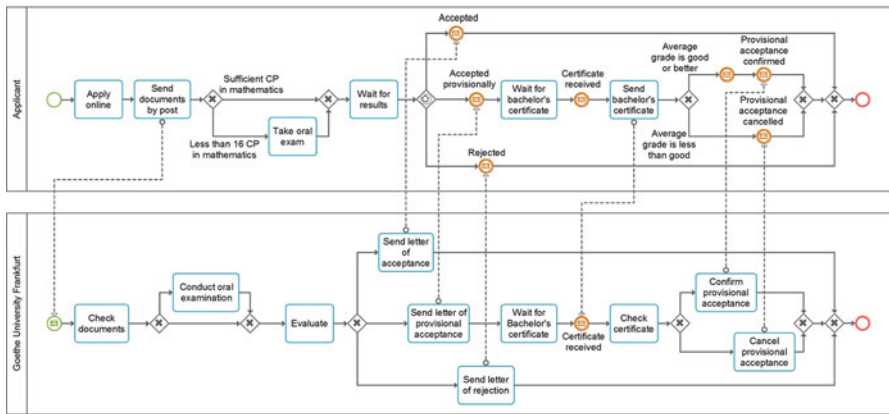
The similarity calculation uncovered limitations in the approaches “La Rosa similarity” (LAROSA) [12] and “Longest common subsequence of traces” (LCST) [9]. 9 of the 156 models in the evaluation dataset could not be processed by the original LAROSA algorithm. Although it was not possible to identify the reason for that, this led to a reduction of the model combinations to be processed by 2727, or, conversely, the similarity of 21,609 out of the overall 24,336 model combinations were successfully calculated. In case of “Longest common subsequence of traces,” 45 of the 156 models could not be processed. Therefore, the similarity could only be calculated for 12,015 model combinations. The challenge for this approach lies in the necessity to calculate all the theoretically possible execution traces for a particular model, since the real-world traces are not available. The used implementation applies the approach of [17] to derive traces; loops were passed only once. Based on the used connectors and the size (in terms of nodes and edges) of a model, this can become very expensive in time and memory. Based on some preceding tests, it was decided to set a trace calculation limit of 40 seconds per model, which led to 41 cancellations. Syntactical errors were a second reason for which the traces of four models could not be calculated. Since no

Type: Models from controlled modeling environments | Dataset: University Admission

Cologne



Frankfurt



Technique	Sim-Value	Technique	Sim-Value
LS3	0,9000	CF	0,6300
LCST	0,5543	SSCAN	0,4828
FBSE	0,8966	LAROSA	0,4028

Fig. 1 Similarity values for two selected models

other approach requires a syntactical correctness of the process model, this is an important limitation. Nevertheless, a similarity analysis with this approach might be meaningful in specific scenarios, e.g., when (1) the real-world traces are known, so that it is not necessary to calculate them based on the model and (2) the intention is to analyze the execution behavior instead of the process concept.

Figure 1 shows two selected models from the dataset with the corresponding similarity values. The example shows several of the above discussed aspects in a concrete setting of the evaluation. First, LCST was not able to deliver a

similarity value since the runtime threshold for calculating all possible traces of 40 seconds was exceeded for at least one of the two models. Running the technique without a time limit, it delivered the similarity value after 35 minutes. Second, the implementation of LAROSA threw an exception since it was not able to handle additional object types, which seem to be unknown by the algorithm. We solved that problem for this model pair by manually removing the organizational elements/lanes. Third, we see significant differences in the resulting similarity values. Having a look at the element labels of the models uncovers a high similarity for a human (both describe a process for a University Admission) although there are different wordings in the process descriptions. Especially, the use of different expressions, such as “take oral exam” vs. “conduct oral examination”, is challenging for purely syntactical similarity measures, like SSCAN.

## 6 Conclusion and Outlook

Based on the practical empirical evaluation, it can be stated that different process model similarity measures lead to substantially different similarity values. The reason for that is founded in (1) different competencies regarding the characteristics of the model dataset (easy vs. hard cases) and (2) the algorithmic approaches for calculating similarity. While FBSE, SSCAN, and LAROSA calculate a similarity value based on a particular node matching only, CF and LCST additionally focus on behavioral characteristics, which are derived from the model structure. In contrast to that, LS3 is the only evaluated measure, which does not require any mapping.

The measures were evaluated with a focus on process model search for process querying, wherefore different relevance criteria can be argued. On the one hand, a high Precision can be desired in order to ensure that all delivered result items are relevant for the search. A high Recall or a high F-Measure can be argued as desirable, as this improves the completeness of the result. Nevertheless, the consideration of additional similarity criteria (like the model structure for CF and LCST) did not lead to an improvement of the measurement results in terms of the expected output (process model result list). LS3 showed an outstanding performance regarding the F-Measure. Also, the results of SSCAN are convincing with constantly high Precision values.

Although the evaluation was executed in the best possible way, there are several threats to validity, which cannot be eliminated in such experiments:

1. **Selection of the models:** For the reason of statistical significance, it would be necessary to randomly select a number of process models from the ground set of existing process models. Since this ground set is unknown, the selection can never be seen as random. Instead, we selected a meaningful mix of synthetic and real-world models.
2. **Validity of the gold standard:** The gold standard is generally determined by humans. Thus, the process of reaching the gold standard is challenging. A

consistent understanding of similarity, correspondence, and the model content is necessary to reach a consensus of the truth. The gold standard represents this consensus, which might be debated again if an additional human is being involved in creating the consensus.

3. **Configuration of the evaluated similarity measurement techniques:** The configurability of similarity measurement techniques allows an adjustment that considers the characteristics of the problem to solve, e.g., depending on the origin of the model data. Since the search goal is not necessarily known, we chose the recommended standard configuration, while other settings may lead to significantly better results.

With these limitations, we can conclude that the measures with the lowest functional complexity (SSCAN and LS3) bring the most appropriate results within the application scenario of process model querying. Since this also affects an outstanding calculation performance in terms of time and consumed resources, they should be further evaluated for the purpose of querying large model repositories to validate their practical applicability.

## References

1. Akkiraju, R., Ivan, A.: Discovering business process similarities: An empirical study with SAP best practice business processes. In: Maglio, P.P., Weske, M., Yang, J., Fantinato, M. (eds.) 8th International Conference on Service Oriented Computing (ICSOC), San Francisco, USA. Lecture Notes in Computer Science, vol. 6470, pp. 515–526. Springer (2010). [https://doi.org/10.1007/978-3-642-17358-5\\_35](https://doi.org/10.1007/978-3-642-17358-5_35)
2. Antunes, G., Bakhshandeh, M., Borbinha, J., Cardoso, J., Dadashnia, S., Francescomarino, C.D., Dragoni, M., Fettke, P., Gal, A., Ghidini, C., Hake, P., Khiat, A., Klinkmüller, C., Kuss, E., Leopold, H., Loos, P., Meilicke, C., Niesen, T., Pesquita, C., Péus, T., Schoknecht, A., Sheerit, E., Sonntag, A., Stuckenschmidt, H., Thaler, T., Weber, I., Weidlich, M.: The process model matching contest 2015. In: Kolb, J., Leopold, H., Mendling, J. (eds.) 6th International Workshop on Enterprise Modelling and Information Systems Architectures (EMISA 2015), Innsbruck, Austria. Lecture Notes in Informatics, vol. P-248, pp. 127–155. Gesellschaft für Informatik (2015)
3. Cayoglu, U., Dijkman, R., Dumas, M., Fettke, P., García-Bañuelos, L., Hake, P., Klinkmüller, C., Leopold, H., Ludwig, A., Loos, P., Mendling, J., Oberweis, A., Schoknecht, A., Sheerit, E., Thaler, T., Ullrich, M., Weber, I., Weidlich, M.: Report: The process model matching contest 2013. In: Lohmann, N., Song, M., Wohed, P. (eds.) Business Process Management Workshops. Lecture Notes in Business Information Processing, vol. 171, pp. 442–463. Springer, Beijing, China (2014). [https://doi.org/10.1007/978-3-319-06257-0\\_35](https://doi.org/10.1007/978-3-319-06257-0_35)
4. de Medeiros, A.K.A., Aalst, W.M.v.d., Weijters, A.J.M.M.: Quantifying process equivalence based on observed behavior. *Data Knowl. Eng.* **64**(1), 55–74 (2008). <https://doi.org/10.1016/j.datak.2007.06.010>
5. Desel, J., Juhás, G.: “What is a Petri net?” informal answers for the informed reader. In: Ehrig, H., Padberg, J., Juhás, G., Rozenberg, G. (eds.) *Unifying Petri Nets: Advances in Petri Nets*, pp. 1–25. Springer, Berlin, Heidelberg (2001). [https://doi.org/10.1007/3-540-45541-8\\_1](https://doi.org/10.1007/3-540-45541-8_1)
6. Dijkman, R., Dumas, M., García-Bañuelos, L.: Graph matching algorithms for business process model similarity search. In: Dayal, U., Eder, J., Koehler, J., Reijers, H.A. (eds.) 7th International Conference on Business Process Management (BPM), Ulm, Germany. Lecture

- Notes in Computer Science, vol. 5701, pp. 48–63. Springer (2009). [https://doi.org/10.1007/978-3-642-03848-8\\_5](https://doi.org/10.1007/978-3-642-03848-8_5)
7. Dijkman, R.M., Dumas, M., Dongen, B.F.v., Käärrik, R., Mendling, J.: Similarity of business process models: Metrics and evaluation. *Information Systems* **36**(2), 498–516 (2011). <https://doi.org/10.1016/j.is.2010.09.006>
  8. Dumas, M., García-Bañuelos, L., Dijkman, R.M.: Similarity search of business process models. *IEEE Data Eng. Bull.* **32**(3), 23–28 (2009)
  9. Gerke, K., Cardoso, J., Claus, A.: Measuring the compliance of processes with reference models. In: Meersman, R., Dillon, T., Herrero, P. (eds.) *On the Move to Meaningful Internet Systems (OTM), Confederated International Conferences, CoopIS, DOA, IS, and ODBASE, Part I*, Vilamoura, Portugal. *Lecture Notes in Computer Science*, vol. 5870, pp. 76–93. Springer (2009). [https://doi.org/10.1007/978-3-642-05148-7\\_8](https://doi.org/10.1007/978-3-642-05148-7_8)
  10. Keller, G., Nüttgens, M., Scheer, A.W.: *Semantische Prozeßmodellierung auf der Grundlage "Ereignisgesteuerter Prozeßketten (EPK)"*. Tech. rep., Institut für Wirtschaftsinformatik, Universität Saarbrücken (1992)
  11. Kunze, M., Weidlich, M., Weske, M.: Behavioral similarity - A proper metric. In: 9th International Conference on Business Process Management, Clermont-Ferrand, France, pp. 166–181 (2011)
  12. La Rosa, M., Dumas, M., Uba, R., Dijkman, R.M.: Business process model merging: An approach to business process consolidation. *ACM Trans. Softw. Eng. Methodol. (TOSEM)* **22**(2), 11:1–11:42 (2013). <https://doi.org/10.1145/2430545.2430547>
  13. Landauer, T.K., Foltz, P.W., Laham, D.: An introduction to latent semantic analysis. *Discourse Processes* **25**(2-3), 259–284 (1998). <https://doi.org/10.1080/01638539809545028>
  14. Lau, C.K., Fournier, A.J., Xia, Y., Recker, J., Bernhard, E.: *Process Model Repository Governance at Suncorp*. Tech. rep., Business Process Management Research Group, Queensland University of Technology (2011). <http://apromore.org/wp-content/uploads/2011/12/Suncorp-project-report-2.pdf>
  15. Levenshtein, V.I.: Binary codes capable of correcting deletions, insertions, and reversals. *Sov. Phys. Doklady* **10**(9), 707–710 (1966)
  16. Manning, C.D., Raghavan, P., Schütze, H.: *Introduction to Information Retrieval*. Cambridge University Press, Cambridge, England (2008). <https://doi.org/10.1017/CBO9780511809071>
  17. Mendling, J.: *Detection and Prediction of Errors in EPC Business Process Models*. Ph.D. thesis, Wirtschaftsuniversität Wien (2007)
  18. Murata, T.: Petri nets properties, analysis and applications. *Proc. IEEE* **77**(4), 541–580 (1989). <https://doi.org/10.1109/5.24143>
  19. Object Management Group (OMG): *Business Process Model and Notation (BPMN)* (2011). <http://www.omg.org/spec/BPMN/2.0>
  20. Polyvyanyy, A., Ouyang, C., Barros, A., van der Aalst, W.M.P.: Process querying: Enabling business intelligence through query-based process analytics. *Decis. Support Syst.* **100**, 41–56 (2017). <https://doi.org/10.1016/j.dss.2017.04.011>
  21. Rahm, E., Bernstein, P.A.: A survey of approaches to automatic schema matching. *VLDB J.* **10**(4), 334–350 (2001). <https://doi.org/10.1007/s007780100057>
  22. Schoknecht, A., Fischer, N., Oberweis, A.: Process model search using latent semantic analysis. In: Dumas, M., Fantinato, M. (eds.) *Business Process Management Workshops: BPM 2016 International Workshops, Rio de Janeiro, Brasilien, Revised Papers. Lecture Notes in Business Information Processing*, vol. 281, pp. 283–295. Springer (2017). [https://doi.org/10.1007/978-3-319-58457-7\\_21](https://doi.org/10.1007/978-3-319-58457-7_21)
  23. Schoknecht, A., Oberweis, A.: LS3: Latent semantic analysis-based similarity search for process models. *Enterp. Modell. Inf. Syst. Archit.* **12**(2), 1–22 (2017). <https://doi.org/10.18417/emisa.12.2>
  24. Schoknecht, A., Thaler, T., Fettke, P., Oberweis, A., Laue, R.: Similarity of business process models — A state-of-the-art analysis. *ACM Comput. Surv.* **50**(4), 52:1–52:33 (2017). <https://doi.org/10.1145/3092694>

25. Song, L., Wang, J., Wen, L., Wang, W., Tan, S., Kong, H.: Querying process models based on the temporal relations between tasks. In: 15th IEEE International Enterprise Distributed Object Computing Conference Workshops (EDOCW), Helsinki, Finland, pp. 213–222. IEEE Computer Society, Helsinki, Finland (2011). <https://doi.org/10.1109/EDOCW.2011.12>
26. Thaler, T., Hake, P., Fettke, P., Loos, P.: Evaluating the evaluation of process matching techniques. In: Kundisch, D., Suhl, L., Beckmann, L. (eds.) Multikonferenz Wirtschaftsinformatik (MKWI), Paderborn, Germany, pp. 1600–1612. University of Paderborn, Paderborn, Germany (2014)
27. Thaler, T., Dadashnia, S., Sonntag, A., Fettke, P., Loos, P.: The IWi Process Model Corpus. Tech. rep., Institute for Information Systems (IWi) at the German Research Center for Artificial Intelligence (DFKI) (2015)
28. Thaler, T., Schoknecht, A., Fettke, P., Oberweis, A., Laue, R.: A comparative analysis of business process model similarity measures. In: Dumas, M., Fantinato, M. (eds.) Business Process Management Workshops: BPM 2016 International Workshops, Rio de Janeiro, Brasil, Revised Papers. Lecture Notes in Business Information Processing, vol. 281, pp. 310–322. Springer (2017). [https://doi.org/10.1007/978-3-319-58457-7\\_23](https://doi.org/10.1007/978-3-319-58457-7_23)
29. van Dongen, B.F., Dijkman, R.M., Mendling, J.: Measuring similarity between business process models. In: Bellahsene, Z., Léonard, M. (eds.) 20th International Conference on Advanced Information Systems Engineering (CAiSE), Montpellier, Frankreich. Lecture Notes in Computer Science, vol. 5074, pp. 450–464. Springer (2008). [https://doi.org/10.1007/978-3-540-69534-9\\_34](https://doi.org/10.1007/978-3-540-69534-9_34)
30. Vogelaar, J., Verbeek, H., Luka, B., van der Aalst, W.M.: Comparing business processes to determine the feasibility of configurable models: A case study. In: Business Process Management Workshops, Clermont-Ferrand, France, pp. 50–61 (2011)
31. Weidlich, M., Dijkman, R., Jan, M.: The ICoP framework: Identification of correspondences between process models. In: 22nd International Conference on Advanced Information Systems Engineering (CAiSE), Hammamet, Tunisia (2010)
32. Yan, Z., Dijkman, R., Grefen, P.: Fast business process similarity search with feature-based similarity estimation. In: Meersman, R., Dillon, T., Herrero, P. (eds.) On the Move to Meaningful Internet Systems (OTM), Confederated International Conferences CoopIS, IS, DOA and ODBASE, Part I, Hersonissos, Greece. Lecture Notes in Computer Science, vol. 6426, pp. 60–77. Springer (2010). [https://doi.org/10.1007/978-3-642-16934-2\\_8](https://doi.org/10.1007/978-3-642-16934-2_8)
33. Zezula, P., Amato, G., Dohnal, V., Batko, M.: Similarity search - The metric space approach. *Adv. Database Syst.*, **32** (2006)



# Complex Event Processing Methods for Process Querying



Han van der Aa, Alexander Artikis, and Matthias Weidlich

**Abstract** Business Process Management targets the design, execution, and optimization of business operations. This includes techniques for process querying, i.e., methods to filter and transform business process representations. Some of these representations may assume the form of event data, with an event denoting an execution of an activity as part of a specific instance of a process. In this chapter, we argue that models and methods developed in the general field of Complex Event Processing (CEP) may be exploited for process querying. Specifically, if event data is generated continuously during process execution, CEP techniques may help to filter and transform process-related information by evaluating queries over event streams. Against this background, this chapter first outlines how CEP fits into common use cases and frameworks for process querying. We then review design choices of CEP models that are of importance when adopting the respective techniques. Finally, we discuss techniques for the application of CEP for process querying, namely those for event–process correlation, model-based query generation, automated discovery of event queries, and diagnostics for event query matches.

---

H. van der Aa (✉)

Data and Web Science Group, University of Mannheim, Mannheim, Germany

e-mail: [han@informatik.uni-mannheim.de](mailto:han@informatik.uni-mannheim.de)

A. Artikis

Department of Maritime Studies, University of Piraeus, Piraeus, Greece

Institute of Informatics & Telecommunications, NCSR Demokritos, Athens, Greece

e-mail: [a.artikis@iit.demokritos.gr](mailto:a.artikis@iit.demokritos.gr)

Ma. Weidlich

Department of Computer Science, Humboldt-Universität zu Berlin, Berlin, Germany

e-mail: [matthias.weidlich@hu-berlin.de](mailto:matthias.weidlich@hu-berlin.de)

© Springer Nature Switzerland AG 2022

A. Polyvyanyy (ed.), *Process Querying Methods*,

[https://doi.org/10.1007/978-3-030-92875-9\\_17](https://doi.org/10.1007/978-3-030-92875-9_17)

## 1 Introduction

The field of Business Process Management (BPM) assumes a process-oriented view on how organizations are structured [24]. In order to analyze the operations of an organization, its business processes are assessed. Such a process is defined by a set of activities, which denote atomic units of work, along with causal and temporal dependencies for their execution. An example would be a Lead-to-Quote process, which comprises activities such as *loading contact data from customer-relationship-management (CRM) system*, *estimating the project effort*, and *preparing a quote template*. Causal dependencies would then define that the loading of contact data precedes the other two activities, which may then be executed concurrently.

Process querying is concerned with models and methods to filter and transform representations of business processes [52]. As such, it supports manifold use cases, reaching from process modeling support, through variation management and performance simulation, to compliance verification. Process representations that are subject to querying may take various forms: A *process model* captures the designed behavior of a process. It specifies the activities and execution dependencies of a process, thereby serving as a blueprint for the execution of a specific instance [39]. A process model may be constructed for various purposes, such as process automation, staff training, or performance simulation. Hence, even for a single process, there may exist various models, each capturing the aspects of the process that are important in light of the purpose of the model [65]. The notion of *event data*, in turn, relates to process representations that capture the recorded behavior of a process, such that an event denotes that a certain state has been reached (e.g., an order request has been received) or that an activity has been executed as part of a specific process instance [60]. Event data is often formalized as an *event log*, a set of traces, each trace being a finite sequence of events that denotes the past behavior for a particular process instance. Process-related data may also be available as an *event stream*, a potentially infinite sequence of events that represent the current behavior of a process.

Complex Event Processing (CEP) defines models and methods to make sense of streams of event data [9, 20]. It defines languages to express queries, which are then evaluated over an event stream, thereby implementing continuous filtering, transformation, and pattern detection. It therefore suggests itself to adopt event-based process querying through CEP, once process-related information is represented by event streams.

While CEP is developed for such *online* event processing, event-based process querying also enables various use cases for *offline* event analysis. This is achieved by replaying event logs, which encode temporal event orders [60], thereby rendering online event-based techniques applicable to static event data.

In this chapter, we outline how CEP methods can be used in the context of process querying. Specifically, this chapter delivers a contextualization and overview of essential techniques in this area, as follows:

- We embed event-based querying by means of CEP methods in the larger process querying context (Sect. 2). That is, we discuss CEP methods with respect to use cases and frameworks for process querying.
- We review common design choices of CEP models (Sect. 3). We highlight which aspects to consider when choosing among available event models, query languages, and system infrastructures when using CEP for process querying.
- We discuss essential techniques for the application of CEP for process querying (Sect. 4). This includes techniques to correlate events with other process representations, to derive queries from process models for control-flow monitoring, to discover such queries automatically from event logs, and to obtain diagnostic information on specific process behavior that is identified by event queries.

The chapter closes with a discussion of open research issues in Sect. 5.

## 2 The Context of Event-Based Process Querying

Event-based process querying can be seen as a special variant of process querying, where the filtered and transformed process representations assume the form of event data. Below, we first elaborate on use cases for event-based querying. Subsequently, we discuss how event-based techniques fit into the broader picture of process querying and provide an illustrative example.

### 2.1 Use Cases

Methods for event-based querying enable the analysis of the recorded behavior of a process. In general, different types of analysis are distinguished, along several dimensions. Analysis questions may relate to a qualitative as well as a quantitative property of a process [24]. The former relates to recorded execution dependencies [18], e.g., whether two activities have been executed in a specific order or a particular number of times. The latter may be defined in terms of execution and wait times, or costs assigned to activity executions [23, 55]. In either case, however, not only the control-flow dimension may be considered. In addition to recorded activity executions, event data often also contains information on processed artifacts or involved resources [45], which can also be subject to event-based querying.

To illustrate the spectrum of applications for event-based querying, we consider two specific use cases: compliance verification and performance monitoring.

**Compliance Verification** Today, the execution of processes is widely supported by information systems. For processes in domains such as logistics [6] or health-care [46], however, the execution of the actual activities is often conducted manually by diverse stakeholders. Yet, there are expectations on how the process is conducted, which, depending on the domain, originate, for instance, from reference models [27] or legal frameworks [43]. When a system guides the execution of a process but does not enforce a specification of its expected behavior, compliance (aka conformance) between expected and recorded behavior needs to be verified explicitly [18, 43, 59]. Based on a formalization of compliance requirements, event-based process querying helps to identify cases of non-compliant process execution [67]. Such mechanisms are particularly useful if applied to event streams representing the most recent behavior of a process: Detecting a compliance violation shortly after it occurred enables the immediate implementation of mitigation and compensation schemes.

In order to fully exploit the potential of event-based querying for compliance verification, several challenges should be addressed. First and foremost, compliance requirements need to be linked to events. For instance, if a particular ordering of activities is required, events must be correlated unambiguously with activity executions as part of a specific process instance to enable conclusions on the compliance of process execution. Moreover, the actual translation of compliance requirements into event queries is cumbersome, since it requires the formalization of the requirements in a (commonly declarative) query model. Hence, the construction of event queries for compliance verification must be supported, e.g., based on models that capture the expected behavior of a business process or event data that is annotated with compliance violations. Furthermore, the interpretation of compliance issues that have been detected by event-based querying is challenging. From a practical point of view, understanding the cause for a compliance violation is important to be effective in the mitigation and compensation of the issue.

**Performance Monitoring** For many processes, ensuring efficient process execution is a core requirement. Whether it is the cycle time of an order process at an e-commerce platform, the wait time of patients as part of clinical pathways, or the cost spent for claim handling processes at an insurance company, a good share of the success of process management depends on quantitative properties of how the process is conducted [24]. Event-based process querying helps to measure these properties: It selects events that are used as input for the computation of performance indicators [21], e.g., the average activity execution time, the delay with which a particular activity is executed after activation, or the accumulated costs induced by a specific type of process instance. At the same time, outliers that represent process execution with anomalous performance can be extracted [54]. Beyond the sheer assessment of the performance, event-based process querying further enables the detection of respective trends, such as continuous deterioration of process performance as well as abrupt drifts [15]. Again, immediate detection of performance issues is a prerequisite for effective countermeasures, which can be achieved through querying of event streams, but not through post hoc analysis of event logs.

As for the previous use case, however, event-based querying for performance monitoring faces the challenge of event correlation. That is, while performance indicators are defined in terms of the business semantics of a process (e.g., based on the time needed to reach a milestone), the task to correlate recorded events with notions of process progress may be non-trivial. Also, the challenges on the construction of event queries and the interpretation of their results, mentioned above for compliance verification, are faced as well in performance monitoring. For instance, a delay in the execution of a business process may lead to ripple effects, so that a large number of performance requirements are not met. Upon detecting all these performance issues, an understanding of the initial deviation that caused further delays is important.

## 2.2 The Context of Event-Based Process Querying

Querying techniques that are grounded in event handling fit into established frameworks for process querying. We illustrate that through the Process Querying Framework (PQF) as introduced by Polyvyanyy et al. [52]. It defines active and passive components, which jointly realize the functionality needed for process querying.

Figure 1 shows a fragment of the PQF, which organizes the respective functionality into four parts, targeting (i) the design of process repositories and queries through modeling, recording, and correlation; (ii) the preparation and (iii) execution

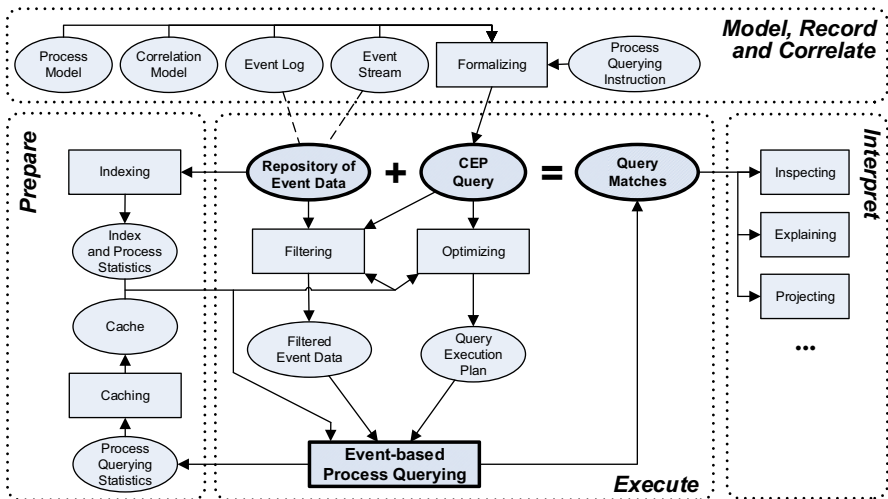


Fig. 1 Instantiation of a fragment of the Process Querying Framework [52] for event-based process querying based on Complex Event Processing

of queries; and (iv) result interpretation. In the context of event-based process querying, components of this framework are instantiated as follows:

(i) *Model, Record, and Correlate* A repository, in general, may comprise various types of artifacts, also referred to as behavior models. For event-based process querying, some types of such models are of particular importance. First and foremost, event logs and event streams represent the event data that is the actual target of any event-based query mechanism. As discussed above, an event log is a set of recorded traces of events that denote the past process execution [60], whereas an event stream is a potentially infinite sequence of events that indicates how the process is currently conducted [17, 31]. Event streams, therefore, cannot be stored in a repository in their entirety. However, most use cases for process querying inherently justify the consideration of a bounded subsequence of an event stream. For instance, queries commonly analyze individual or specific groups of process instances, so that, assuming that process instances eventually terminate, the finiteness of event data relevant to a particular query is ensured.

Moreover, behavior models that assume the form of process models and correlation models play a role for event-based process querying. Neither type of artifact is used as a query target, i.e., they are not part of the repository in the sense of the PQF but may guide the definition of event queries. This is achieved by formalizing a process querying instruction. In part, this instruction may automatically be generated from a process model [10, 67], which we detail later in this chapter. For instance, for either of the above use cases, the expectations regarding compliant and high-performance process execution may have been materialized as a process model. As such, queries that are derived from it can be used to ensure that these expectations are met in the process behavior as recorded in event data. To this end, however, it may be required to first establish the relation between process model elements and event data using a correlation model [11]. Again, we will highlight essential techniques for this step.

In the absence of artifacts that can be used to formalize process querying instructions, the event data itself may also form a starting point for query development. As we will detail later, based on event data that is annotated with relevant situations that shall be matched, a respective query may actually be learned (semi-)automatically.

(ii) *Prepare* Given a repository of event data and an event query, the evaluation of the query may be prepared. The PQF defines indexing and caching schemes as the major means for such preparation. Those also apply to event-based process querying, whereas their implementation depends on the specifics of the respective type of event data and formalism adopted for event queries. Basic techniques include indexing of specific types of events, their frequencies, or constraints on the occurrence or absence of events in an event log or an event stream.

(iii) *Execute* As part of the execution stage, the information resulting from the preparation is exploited to filter the event data and optimize the execution plan of an event query. An example for the former is the projection of all events that cannot be of relevance for the query at hand. Examples for the latter are satisfiability checking [22], rewriting of event queries [68], or sub-pattern sharing [53] based on

information on the event data. In any case, the execution of the query yields a set of query matches. As those are given as event data, they may be thought of yet another process repository as put forward by the PQF.

(iv) *Interpret* Obtained query matches represent the input to the last part of the framework. Matches may be interpreted in light of the respective application scenarios. For the aforementioned use cases, for instance, query matches may denote compliance violations or aggregated performance measurements. By putting query matches into context, e.g., comparing them to those obtained by other queries, for other traces, or at other times, they may also enable the explanation of the queried phenomena. As an example, we later review a technique that provides diagnostic insights into non-compliance by assessing the interplay of the matches obtained with event queries.

Query matches may also be used for further interpretation. In Fig. 1, we included the option to project the results on behavior models for analysis purposes. For instance, projecting query matches on the originating event data may be useful to quantify the relative amount of observed matches. However, in practice, many further types of interpretation of query matches are possible, so that this list is not supposed to be understood as an exhaustive overview of how to make sense of query matches.

### 2.3 An Example Scenario

We now turn to an exemplary scenario related to a *Lead-to-Quote* process, which will serve as a running example throughout the remainder.

Key event types in such a Lead-to-Quote scenario will capture process milestones including *received a lead*, *project details entered into ERP (Enterprise Resource Planning) system*, and *quote sent*. Events of these types may be derived from the relational data stored in the system, e.g., the insertion of a tuple into a relation that captures leads can be interpreted as a signal that an instance of an activity *received a lead* had been executed. Given these events, managers can establish monitoring requirements that track behavioral properties, such as the *qualitative requirement* that project details must always be entered into an ERP system before a quote can be sent out, or a *quantitative requirement* used to ensure that a quote is sent out within two weeks after receiving the lead. The automatic and continuous monitoring of these requirements is then achieved by translating them into event queries. In the presence of models or annotated data, this translation may even be supported using automated techniques. The resulting queries can use both pattern recognition (e.g., events occurring in a particular order for a specific case) and data attribute analysis (e.g., timestamps and other payload data associated with events) to monitor both qualitative and quantitative monitoring requirements.

Finally, we introduce a process model for the Lead-to-Quote scenario, depicted in Fig. 2. The modeled process starts with an import of contact data (activity *a* or *b*) or

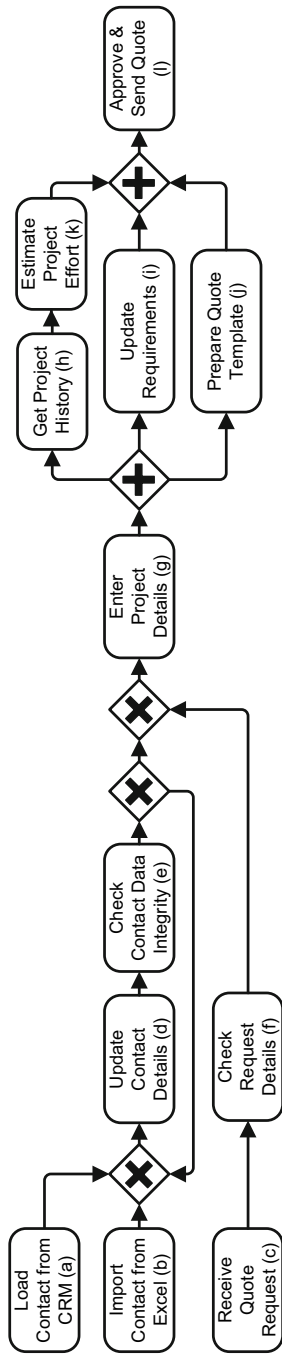


Fig. 2 Example Lead-to-Quote process modelled in BPMN



with the receipt of a request for quote (*c*). In the former case (following *a* or *b*), the contact details are updated (*d*). This step may be repeated if data integrity constraints are not met (*e*). In the latter case (after *c*), the request details are checked (*f*). For both cases, the quote is then prepared by first entering prospective project details (*g*). This is followed by conducting an effort estimation (*h* and *k*), updating the requirements (*i*), and preparing the quote template (*j*). These latter steps are done concurrently. Finally, the quote is approved and sent (*l*). A process model like this may have been created in order to capture the expected or required behavior of the process. It can serve as a basis to (semi-)automatically derive monitoring queries that allow for the continuous comparison of expected and actual process behavior, as described later in this chapter.

### 3 Complex Event Processing

Complex Event Processing (CEP) emerged as a computational paradigm to handle streams of event data [9]. This paradigm has to be seen in the context of the broader areas of data stream processing [28] and stream reasoning [5], which are all concerned with online processing of data that is continuously generated [20]. More specifically, the focus of common CEP models and methods is the detection of event patterns, sets of events that are correlated in terms of their ordering, their payload data, and the context in which they occur.

Even when considering only the scope of CEP, however, we note that a plethora of different event processing models have been proposed in the literature. We therefore refrain from adopting one particular model and rather highlight important aspects of formalisms for event streams and event queries, as well as query evaluation infrastructures. As such, our focus is to highlight the spectrum of models and methods for CEP that may be exploited for process querying.

#### 3.1 Event Streams

An *event* is commonly seen as an “occurrence within a particular system or domain” [26]. It is a recording of some state change that is considered to be of relevance. In the context of process querying, such state changes typically refer to the progress of process execution as, for instance, indicated by the execution of an activity as part of an instance of the process.

Events may be defined following different formalisms and conceptual models. However, most models share the requirement to capture some essential information about an event, as follows. First and foremost, an event *e* has an *identifier*, which we denote by *e.id*. It renders the event uniquely distinguishable. Second, each event is assigned a *timestamp*, denoted by *e.t*. Such event timestamps enable (relative) ordering of events. In addition, they may be exploited in event queries to assess

the absolute time difference between events. Third, each event is of a specific type, denoted by *e.type*. A type refers to a specification that serves as a meta-model for a set of events, i.e., events are instances of their type specification. In many application scenarios, event types define both a specific syntax to capture respective events and their semantics.

An event stream is defined by a potentially infinite set of events  $E$  and an order relation  $< \subseteq E \times E$  (either partial or total). The fact that a stream is potentially infinite means that, in practice, processing is based on the stream at a specific point in time, i.e., the prefix of the stream up to this time.

Despite its simplicity, the above event stream model incorporates assumptions and design choices along several dimensions:

*Timestamp semantics:* The exact meaning of the timestamp of an event may vary.

The timestamp can, for instance, denote the time of event occurrence, the time of event recording (which is potentially delayed with respect to its occurrence), or the time of event arrival at a CEP infrastructure [14].

*Event atomicity:* Events may have point-based or interval-based semantics, meaning that an event is either atomic or has a duration [40]. Conceptually, an even more fine-grained structure may be considered, in which events comprise multiple intervals that represent suspension and resumption.

*Stream ordering:* Event stream models differ in their assumptions on the ordering of events. Specifically, the order relation  $<$  can be assumed to be a total order or a partial order [69]. The former may keep synchronization issues out of the event processing model, whereas the latter enables to incorporate event generation by distributed, independent sources [3]. Also, note that incorporating more than one timestamp semantics leads to multiple notions of order for the events of a stream, e.g., an occurrence order and an arrival order, so that inconsistencies need to be handled [42].

*Payload data:* While the type of an event defines its syntactic structure, this structure may assume various forms. For instance, events may be structured according to a relational model (a type defines a relational schema) [7], adopt an object-oriented model (a type defines a set of concepts and their relations), or use tree-based formalisms to define the data carried by an event (a type defines, for instance, an XML schema) [72]. Note that relational models are often employed already in the context of process querying [52].

*Event semantics:* In a simple model, semantics of events are directly induced by their type definition. However, it has been argued that these semantics shall also be specified explicitly, adopting appropriate formalisms for knowledge representation [30, 57].

For illustration, we take up the scenario introduced in Sect. 2.3. Adopting a relational model for the payload of events, Table 1 lists three exemplary events. This illustrates some of the aforementioned aspects of event models: Instead of having separate types per reached state or executed activity, the example events are of a unified type `Act` (representing that an activity has been executed), while an attribute `name` references milestones (e.g., `QR` for quote request received) and activities (e.g.,

**Table 1** Three example events for the introduced scenario

Id	Timestamp	Type	order_id	Name	Client	Price
11	21.09.18,15:12:36	Act	O23	OR	Franklin	74,500
12	21.09.18,15:12:36	Act	O67	OR	Meyers	28,200
42	24.09.18,09:72:10	Act	O23	ED	Franklin	74,500

ED for entering details). Moreover, events are atomic, while the timestamp semantics is assumed to denote the end of activity execution. The identical timestamps of events 11 and 12 illustrate that events may happen concurrently (e.g., a batch of quote requests is received), inducing a partial order over the stream.

### 3.2 Event Query Languages

Numerous models and languages for the definition of event queries have been proposed in recent years [9, 20, 34]. While there have been recent initiatives to converge on a model, notably the match-recognize operator for row-based pattern matching as included in SQL:2016 [36], as of today, there is no common standard for event query languages. Rather, CEP systems define their own languages, differing in syntax and semantics. Since the respective languages have been developed in various communities, any comparative assessment is further hindered by the differences in the adopted terminologies and underlying event model [20]. In this section, we therefore focus on an overview of generic types of query operators, exemplify the syntax of query languages, and point to different ways to achieve a formal grounding of queries.

**Common Operators' Types** It has been noted that many languages for the specification of event queries, despite all their differences, share at least a set of common operator types [73]. While the specific definitions of query operators may still vary in syntax and semantics, these types describe rather abstract functionality that is typically supported by a query language. Specifically, these types are:

*Disjunction and Conjunction:* Query operators define that an event pattern is characterized by the occurrence of either of a set of events (disjunction) or their joint occurrence (conjunction).

*Sequencing:* A sequence operator defines a list of events that have to occur in the respective temporal order for a query to match.

*Kleene Closure:* An operator defines a pattern as a recurring occurrence of a specific event, with the number of occurrences being finite, but unbounded.

*Negation:* A query operator checks for the absence of a specific event, generating query matches only if the respective event is not part of the processed stream.

*Data Predicates:* A query operator specifies conditions for events to be part of a match, based on their data payload.

*Windowing:* A window operator specifies time-based or ordering-based conditions for events to become part of a query match.

*Event Construction:* A query operator specifies how a new event, emitted as part of an output stream generated by a query, is constructed from matched events.

Again, it has to be stressed that the precise definitions of operators of these types are typically language specific. For instance, the semantics of operators such as sequencing and Kleene closure needs to be further disambiguated through processing policies that clarify, for instance, how often a single event can be part of a match; which further events may occur between matched events; and how to select among multiple candidate events for a query match, see [1, 33].

Moreover, query languages differ significantly in terms of their compositionality, i.e., the support to build complex queries by combining operators of the aforementioned types. Several existing languages restrict compositionality, e.g., in terms of nesting Kleene closure operators.

**Exemplary Languages** To illustrate how the above operators are actually represented in common query languages, we now take up the example scenario introduced in Sect. 2.3. Specifically, we focus on compliance verification and the requirement that project details for a received quote request must be entered *before* a quote can be sent out. As an additional condition, we require that this rule applies only for quotes with a value of more than 10.000 Euro. Let us assume that the underlying event model defines a single type for activity executions (with an attribute capturing the activity) and incorporates atomic events that are totally ordered and denote the occurrence of activity execution, while the event payload is always defined by a relational schema.

Figure 3 exemplifies how such a query would be defined in two languages. Figure 3a defines the query in the SASE language [73]. After specifying the input stream, the query comprises a pattern definition. The latter includes a sequence (SEQ) of three event variables (a1-a3), each of the same type Act, while the second variable is negated (!Act a2). The Where clause then specifies conditions for the pattern to detect. The clause captures processing policies (skip\_till\_next\_match means that irrelevant events may be skipped as long as no relevant event occurs) and

---

```

From act_stream
Pattern
  SEQ(Act a1, !Act a2, Act a3)
  Where skip_till_next_match(a1,a3)
  And [request_id]
  And a1.name='QR'
  And a2.name='ED'
  And a3.name='SQ'
  And a3.price > 10000
  Within 10 days

```

---

(a)

---

```

Create Context qr_context
  Partition By request_id
  From act_stream;
Context qr_context
  Select * From
  Pattern [
    Every Act(name='QR') -> (
      Act(name='SQ' and price>10000)
      And Not Act(name='ED'))
    Where timer:within(10 days)];

```

---

(b)

**Fig. 3** Example queries. (a) Example query in SASE [73]. (b) Example query in EQL [25]

data predicates, such as `[request_id]` correlating all events by the identifier of the quote request and statements that refer to the names of executed activities. Finally, a time-based window is defined for the query (`within`).

Figure 3b illustrates a related (due to subtle differences in semantics not equivalent) query in the Esper Pattern Language (EPL) [25]. Here, the partition of the input event stream is realized through the definition of a `context` over the attribute `request_id`. Under this context, the actual query then selects data from a pattern that defines that an `Act` event, for which the name indicates a quote request, shall be followed by an event related to the quote submission, without an event capturing that the details have been entered. While adopting a different syntax, this query also contains the respective data predicates as well as the window definition.

**Formal Grounding** For query languages such as those illustrated above, different models have been proposed to use as a formal basis. Unfortunately, we note that most of the proposed formalisms suffer from two problems: They are incomplete, i.e., they capture only a subset of the aforementioned query operators; and they are not based on well-established formalisms, so that existing theoretical results and reasoning methods cannot be exploited in the context of query analysis.

As reviewed in [9], the most common proposals to formalize event queries include automata-based, tree-based, and logic-based models. Systems as Cayuga [16], SASE [73], and TESLA [19] adopt automata, in which the states represent the progress in query processing, while state transitions carry guards that encode the conditions for the evolution of a partial match. As these models operate on an infinite universe of events, evaluate predicates over data, and produce output explicitly, they incorporate ideas of symbolic and register automata as well as transducers. However, the models proposed for CEP languages are typically not directly based on such established notions of formal language theory.

Queries may also be formalized as trees, where non-leaf nodes are event operators that construct partial matches from the events matched by their children. Leaf nodes, in turn, define data predicates for the single events to be matched. An example for this approach is the model of ZStream [48].

Finally, logic-based formalisms can serve as a basis for event queries. Examples include chronicle recognition and the event calculus. The former relies on temporal logic. It encodes the occurrence of events by logic predicates that define the time of occurrence and the event payload; see the example of TESLA [19]. Contextual and temporal constraints then define event operators, time windows, and data predicates of an event query. The event calculus [8] relies on fluents as essential building blocks. A fluent is a property that may assume different values over time, while changes in this property are encoded by logic predicates. Queries in the event calculus are then defined as rules over the fluents.

### 3.3 *Event Query Evaluation*

Various systems and infrastructures have been proposed for the evaluation of event queries. In most cases, the formal models mentioned above, which serve as a basis for the definition of event queries, directly give rise to an execution model. An automata-based formalization of a query is used as follows: A CEP system keeps track of partial runs of the automaton. Processing an event then requires to assess whether a new run shall be instantiated according to the initial state of the automaton, and whether the existing runs shall be extended, duplicated, or terminated [16]. Similarly, adopting a tree-based model for evaluation, a CEP system maintains buffers for all leaf nodes. Upon filling them with a batch of new events, buffers for partial matches at non-leaf nodes are filled or emptied [48]. For logic-based models, a CEP system conducts logical inference to see whether query matches materialize. In a streaming setting, this is done whenever facts representing new events have been inserted into the knowledge base [8].

The continuous evaluation of queries over high-velocity event streams is a common performance bottleneck, so that a plethora of optimization strategies have been developed. A recent survey [35], focusing not only on CEP but also the broader area of stream processing, classifies these optimization strategies based on whether they change the topology of an operator graph, whether they change the semantics of queries, and whether they are applicable at design time or run time. Specific examples of optimization techniques include load shedding for CEP that limits processing to a subset of the arriving events [32, 74]; delayed construction of partial matches during run time [73]; semantic query rewriting based on constraints on the event stream [68]; and sharing of partial matches among several queries [53].

## 4 **Methods for Process Querying**

The application of CEP models and methods as discussed above in the context of event-based process querying has to cope with several challenges, partially mentioned for two exemplary use cases in Sect. 2.1. In this section, we take up these challenges and discuss four essential techniques to address them: event–activity correlation (Sect. 4.1), model-based query derivation (Sect. 4.2), discovery of event queries (Sect. 4.3), and diagnostics for event query matches (Sect. 4.4).

### 4.1 *Event–Activity Correlation*

A fundamental requirement for analysis techniques involving event data alongside other representations of a process, i.e., process models, is that observed event types can be linked to process model elements, such as activities or decision points.

In terms of the Process Querying Framework [52], these links are captured in a correlation model that establishes the relation between elements of different process representations. For instance, in compliance verification, the expected behavior of a process may be formalized by a process model, against which the recorded behavior, i.e., events that denote activity executions, is assessed. Typically, however, such required event–activity correlation is not readily available [11, 49]. Furthermore, *manually* establishing correlation is often unfeasible because analysts rarely possess the necessary knowledge on the details of a process implementation [68]. Consequently, it is highly beneficial to establish event–activity correlation in an *automated* fashion. However, to reliably achieve this, challenges including cryptic data values in the definition of events, noisy and non-compliant behavior, as well as complex event–activity relations must be taken into account [62]. A variety of automated techniques have been developed that aim to overcome such challenges and to identify correspondences between recorded events and process model elements, for convenience referred to as *activities* in the remainder. In this sense, the goal of event–activity correlation can be framed as a matching problem. To address this problem, automated correlation techniques can consider various types of information:

**1. Event and Activity Label Information** The labels (or values) assigned to attributes that are associated with recorded events and process model activities represent valuable information for the establishment of event–activity correlation. In optimal scenarios, events and activities can be correlated when they have equal or highly similar names, e.g., an event carries an attribute with the label *project information submitted*, while an activity is labeled *enter project details*. A plethora of *similarity measures* exist that can be used to compute the degree of similarity between two text fragments. These measures can be generally divided into streams of syntactic and semantic similarity measures [4].

*Syntactic* similarity measures, such as the *string-edit distance* and N-gram distance, compute the degree of similarity between two text fragments by comparing their character sequences. By contrast, *semantic* similarity measures, such as measures based on WordNet or on Distributional semantics, consider word similarity based on the meaning of words [2]. For example, the words “*contract*” and “*agreement*” have a high semantic similarity, because they both describe an *exchange of promises*. Both types of similarity measures have their benefits and disadvantages. This is illustrated in Table 2 (derived from [63]), which compares the values of syntactic and semantic similarity measures, respectively, obtained using the *Levenshtein distance* [71] and *Lin similarity* [41]. Note that both measures range between 0.0 and 1.0, where 1.0 denotes perfect similarity. The table, for instance, shows that syntactic similarity measures can recognize similarity in spite of typographical errors (indicated by the high  $sim_{syntactic}$  value for *agreement* vs. *argeement*), whereas semantic measures are able to differentiate among words with similar syntax but different meanings (contract vs. contact). Therefore, both types of similarity measures are often used in conjunction.

**Table 2** Comparison of syntactic and semantic similarity scores

$t_1$	$t_2$	$sim_{syntactic}$	$sim_{semantic}$
<i>agreement</i>	<i>agreement</i>	0.88	n/a
<i>contract</i>	<i>contract</i>	0.88	0.10
<i>contract</i>	<i>agreement</i>	0.11	0.96

**Table 3** Exemplary declarative process constraints

Constraint	Explanation
<code>init(a)</code>	a is executed first in a process instance
<code>coexistence(a, b)</code>	If a occurs in a process instance, then b also occurs, and vice versa
<code>response(a, b)</code>	If a occurs in an instance, then b will eventually occur afterwards
<code>chainResponse(a, b)</code>	If a occurs in an instance, then b will immediately occur afterwards

However, in practice, recorded events are often associated with far less useful labels. For examples, attributes of an event may merely be associated with cryptic database fields such as *CDHDR* or *I\_SM\_E* [12]. In these cases, not even advanced linguistic analysis tools are able to reliably identify a correlation with process model elements, if only label information is considered.

**2. Structural Information** Information on the behavioral relations that exist among events and among process model activities can be highly relevant when establishing event–activity correlation. As an illustration, consider the situation in which two event types, e.g.,  $E_1$  and  $E_2$ , appear to be mutually exclusive, i.e., where for every observed process instance at most one of them occurs. Such an event relation can be a relevant indicator to determine that these event types correspond to, for instance, the *Import contact from Excel* and *Receive quote request* activities from the model in Fig. 2, which also exclude each other. Similarly, the position at which events occur in process instances, e.g., toward the beginning of an instance, can be a strong indicator that these events correlate with activities that appear at comparable positions in the process model.

Event–activity correlation techniques consider structural properties in different ways. Van der Aa et al. [61] propose a measure for *positional similarity* quantifying similarity based on the average position in which events or activities occur in process instances. Baier et al. [13] analyze more complex behavioral relations by deriving *declarative process constraints* for events and activities. These constraints can capture various structural relations, such as those depicted in Table 3. For example, the `init` constraint can be used to specify that an event or activity occurs first in a process instance, whereas constraints such as `coexistence` and `response` identify inter-relations among events or activities. Given such structural relations, the approach from [13] then identifies a 1:1 correlation between events and activities for which the structural relations are most similar, through constraint-based optimization.

A challenge regarding the derivation of process–event correlation based on structural information is that this implicitly assumes that the observed events follow the process in accordance with the specified process model. However, in



practice, factors such as noise and non-compliant behavior can lead to deviations between the recorded events and the specified process model. As a result, correlation techniques can, for instance, not blindly assume that when a process model specifies that activity *A* occurs before activity *B*, the corresponding events will always be observed in this order as well. To deal with this, techniques such as the one from Baier et al. [13] use probabilistic means to identify the most likely correlation, taking into account potential process deviations.

**3. Pattern Matching** A key challenge when establishing event–activity correlation is that there can exist complex relations between events and activities. Often, recorded events correspond to more fine-granular process steps than process model activities, which can lead to many-to-one or many-to-many relations [11]. Recognizing such complex correlation generally requires the consideration of information beyond event and activity labels or structural relations.

A technique by Baier et al. [11] aims to recognize one-to-many and many-to-many correlations by considering natural language texts (i.e., *work instructions*) that may be associated with process model activities. The premise of their approach is that such work instructions contain detailed information regarding the execution of process activities, which may correspond better to the level of detail of observed events.

Still, event–activity correlation can be even more complex, in situations where process model activities correspond to particular event patterns. In these situations, a single model activity may correspond to the occurrence of an event pattern [44]. If these complex correlations are known, they can be manually established, though such a task heavily depends on domain knowledge from analysts.

However, it is also possible to automatically correlate complex event patterns to process model activities. For instance, Senderovich et al. [56] discover correlations between sensor readings (i.e., events) and process model activities by analyzing sensor readings that indicate employee interactions in healthcare environments. Their approach considers factors such as the entities involved in the interaction, its location and duration, and the preceding and succeeding interactions. Given these factors, the approach then aims to identify event patterns that correspond to particular activities in a process model, ultimately yielding an event–activity correlation.

## 4.2 Model-Based Query Generation

As discussed in Sect. 2.1, compliance verification is an important use case for event-based process querying. Assume that a specification of the expected behavior of a process is available in terms of a process model and that its elements are linked to event types by a correlation model. If the respective specification is not enforced during execution, event queries may be formulated to detect any deviation of the recorded from the modeled behavior. In this section, we therefore consider how to

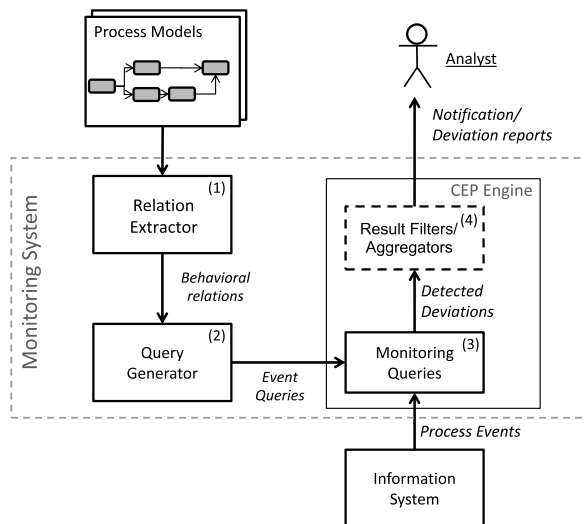
generate such monitoring queries from a process model. For this section, we follow a generic architecture, as introduced next.

### 4.2.1 Overview

Figure 4 provides a generic architecture for systems that facilitate the automatic generation of monitoring queries as well as the preprocessing of their results to provide diagnostic insights. The system takes two kinds of input: process models and process events. The *process models* are used to define the normative behavior of processes. We do not impose any assumptions on the process model notation or language used to define them. Regarding the *process events*, we assume that these events reflect the completion of activities as recorded by some information system. If recorded events do not directly capture this, correlation techniques such as discussed in Sect. 4.1 can be used as a preprocessing step.

As depicted in the figure, the generation and preprocessing of monitoring queries consists of four main steps: (1) the extraction of behavioral relations from process models, (2) the generation of monitoring queries, (3) the evaluation of these queries over process events, and (4) filtering and aggregating the detected deviations. In this section, we focus on Steps 1 and 2, which result in the generation of monitoring queries from normative process models. The evaluation of these queries (Step 3) is performed by CEP engines, as discussed in Sect. 3.3. In the light of the different sets of query operators provided by common query languages, here, we merely assume that event queries may comprise binary operators for conjunctive (*and*), ordered (*ord*), and subsequent occurrence (*sub*) of events of a particular type, whereas a negation operator supports testing for the absence of events (*not*).

**Fig. 4** Overview of a generic architecture for the generation of monitoring queries



Finally, the post-processing of the identified deviations using filtering and aggregation (Step 4) to provide diagnostic insights is discussed in Sect. 4.4.

### 4.2.2 Query Derivation

We derive monitoring queries from a process model by first computing behavioral relations for the model. Behavioral relations [51], such as the sets of alpha relations [58], (causal) behavioral profile relations [66], and the relations of the 4C spectrum [50], define constraints that should hold between process activities according to a process model. Given a certain set of behavioral relations, we can then establish a monitoring query that identifies when the behavioral relation is not satisfied, i.e., when a compliance violation occurs.

As an illustration, we will discuss the derivation of monitoring queries for four behavioral relations: *exclusiveness*, *co-occurrence*, *strict order*, and *directly follows*.

**Exclusiveness** The exclusiveness relation  $+$ , part of the behavioral profile relations, denotes that two activities should not occur in the same process instance. For instance, activities  $a$  and  $c$  should not occur in the same instance of the running example from Fig. 2. To identify compliance violations of these constraints, we define queries that match the joint execution of two exclusive activities, e.g., we define a query that recognizes instances where both  $a$  and  $c$  are executed.

Using  $+$  to denote all pairs of process model activities that are in an exclusiveness relation (including self-relations), we define the *exclusiveness query set* as follows:

$$Q_+ = \bigcup_{(a_1, a_2) \in +} \{and(a_1, a_2)\}.$$

For the model in Fig. 2, monitoring exclusiveness comprises the following queries:  $Q_+ = \{and(a, a), and(a, b), and(b, a), and(a, c), \dots, and(e, f)\}$ . Mirrored queries such as  $and(a, b)$  and  $and(b, a)$  have the same semantics and can, therefore, be filtered through optimization techniques of a CEP implementation.

**Co-occurrence** The co-occurrence relation  $\gg$ , part of the causal behavioral profile relations, denotes that two activities should always occur together in a completed process instance. For example, for the model in Fig. 2, there is no trace that can contain only activity  $c$  but not  $f$ , and vice versa. Contrary to the query patterns derived for exclusiveness, co-occurrence constraints are violated not by the *presence* of a certain activity execution, but by its *absence*. Therefore, a constraint violation materializes only at the completion of a process instance. Only then it becomes visible which activities are missing in the observed execution sequence even though they should have been executed. Using  $\gg$  to denote the set of activities in a co-occurrence relation for the behavioral profile of a model, we construct a corresponding query set as follows:

$$Q_{\gg} = \bigcup_{(a_1, a_2) \in \gg} \{and(and(a_1, END), not(a_2))\}.$$

For a running process instance, the question whether an activity is missing cannot be answered definitely, because the activity may still occur. Nevertheless, the strict order relation can be exploited to identify states of a process instance, which may evolve into a co-occurrence constraint violation. In particular, it is possible to query for activities for which we deduced from the observed execution sequence that they should have been executed already. That is, their execution is implied by a co-occurrence constraint for one of the observed activities and they are in strict order with one of the observed activities; see [67] for a detailed description.

**Strict Order** The strict order relation  $\rightsquigarrow$ , part of the behavioral profile relations, indicates that two activities should occur in a particular order, if they both occur in the same process instance. For example, for the running example, the relation  $e \rightsquigarrow g$  holds, since  $g$  can never occur before  $e$ . By contrast, the relation  $d \rightsquigarrow e$  does not hold, due to the loop surrounding the two activities. We query for violations of the strict order of activities with an order query set that matches pairs of activities for which the order of execution is not in line with the behavioral profile relation  $\rightsquigarrow$ . Using  $\rightsquigarrow$  to denote the set of activities of a model in strict order, we define these queries as

$$Q_{\rightsquigarrow} = \bigcup_{(a_1, a_2) \in \rightsquigarrow} \{ord(a_2, a_1)\}.$$

For the model in Fig. 2, the order query set contains the following queries:  $Q_{\rightsquigarrow} = \{ord(d, a), ord(e, a), ord(g, a), \dots, ord(l, k)\}$ .

**Directly Follows** The directly follows relation  $<$  from the set of alpha relations denotes that two activities only directly succeed each other in a particular order. For instance, activities  $h$  and  $k$  are in the relation  $h < k$ , since  $k$  never directly precedes  $h$ , while it can directly follow  $h$ . We query for violations of the respective relation with a query set that matches pairs of activities with subsequent executions, which are not captured by relation  $<$ . Let  $A$  be the set of all activities in a process model, and let  $<$  denote the set of activities in the directly follows relation. Then, we define the respective set of queries as

$$Q_{<} = \bigcup_{(a_1, a_2) \in (A \times A) \setminus <} \{sub(a_1, a_2)\}.$$

Again, we use the model in Fig. 2 for illustration. The query set contains the following queries:  $Q_{<} = \{sub(a, b), sub(a, c), sub(a, e), \dots, sub(h, l)\}$ .

### 4.3 Discovery of Event Queries

The previous section showed how to construct event queries based on a specification of the expected behavior of a process, under the assumption that events have been correlated with process model elements. However, such an approach may not be

feasible. It may be impossible to link events to a process model, e.g., due to severe differences in the assumed level of abstraction, or a process model specifying the expected behavior may not be available at all. If so, however, historic event data that is annotated with the situation of interest, e.g., a compliance violation or the attainment of a milestone, may serve as the basis for process querying. In this section, we first outline how such annotated event data leads to a supervised learning problem, before discussing a specific algorithm to address this problem.

### 4.3.1 The Problem of Event Query Discovery

The setting of event query discovery is illustrated in Fig. 5. Some event data (assumed to be totally ordered in the figure) is assigned annotations that indicate that a specific situation materialized. Such a situation may, for example, be an instance of the process violating a compliance rule or reaching a milestone during processing. Either way, the respective situation may be identified in retrospect in order to obtain such annotations. While the annotations identify the point in time at which the situation occurred, the actual event pattern indicating it is not, or only partially, known.

The problem of event query discovery is the construction of a query that matches whenever an annotation indicates that the situation of interest occurred. That is, the annotated event data serves as a training dataset with respect to some data without annotations, presumably generated by the same process. Under the assumption that the materialization of the situation in the annotated event data can be projected to the one without annotations, analysis may then rely on the derived query to detect formerly unknown occurrences of the situation. Put differently, query learning postulates that the event pattern signaling the situation is the same in the annotated and plain event data.

When aiming at the construction of a query that matches whenever there is an annotation, the scope of potential matches has to be limited, though. As the query shall generalize over the multiple occurrences of an event pattern (one per annotation), a *set* of training sequences has to be derived from the annotated event

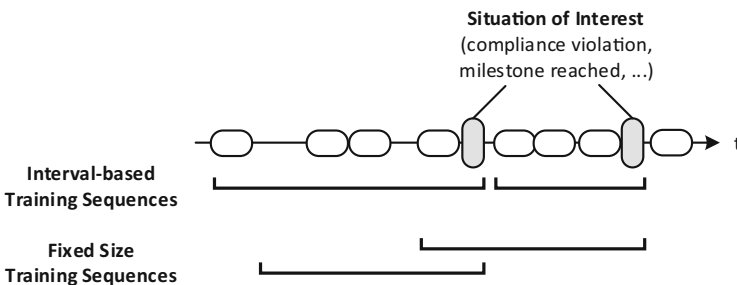


Fig. 5 The setting of event query discovery

data. As illustrated in Fig. 5, different approaches may be followed for this purpose. For example, all events between two subsequent annotations may be considered as a separate *interval-based* training sequence, so that no two sequences overlap. This implicitly encodes an assumption on how occurrences of the situation of interest materialize—on each occurrence the state of processing is entirely reset. Depending on the process at hand, however, this assumption may not hold true. In that case, *fixed size* training sequences may be derived from a fixed amount of the event data preceding each annotation. Here, the amount may be determined based on volume (a fixed number of events) or time (a fixed temporal period).

### 4.3.2 Discovery Algorithms

Any attempt to address the problem of event query discovery has to be tailored to the models assumed for event streams and event queries, respectively. Intuitively, the more expressive these models, the larger the space of candidate queries to be considered as solutions for the discovery problem. A simple case would be the one of an event stream that comprises a total order of symbols and a query model that defines queries solely as a sequence of such symbols. Then, query discovery may simply be traced back to frequent sequence mining [64, 70], detecting the subsequences that are shared among all training sequences.

However, as described in Sects. 3.1 and 3.2, common models for event streams and query languages, in particular in the context of event data generated by processes, are much more complex. Events are not simple symbols but typed and carry a structured payload. Queries are not limited to sequences of symbols but may comprise data predicates, negation operators, and time windows. Against this background, a few tailored algorithms have recently been proposed for event query discovery, specifically iCEP [47] and the IL-Miner [29]. While these algorithms cannot cope with the full expressiveness of CEP languages (e.g., neglecting Kleene closure and negation operators), they discover queries built of sequence operators, data predicates, and time windows. Below, we summarize the main ideas of the IL-Miner [29], as it supports a more expressive query model compared to iCEP [47]. Note that there are also techniques for complex event pattern discovery supporting logic-based approaches, such as OLED [37, 38] for event calculus.

In essence, one first tries to identify data predicates that refer to single events and appear to be relevant for query discovery. To this end, it is assessed, for which atomic predicates, an event that satisfies it can be found in any of the training sequences. Subsequently, the identified predicates, coined in relevant event templates, are used to abstract the training sequences, yielding sequences of templates. Based thereon, standard frequent sequence mining is conducted, which yields sequences of templates. Intuitively, each such sequence provides a skeleton for the construction of a set of event queries. In a next step, to construct a query, each sequence of templates is linked to the events of the original training sequences. From the obtained sequences of events, further data predicates (e.g., those referring to more than one event in a pattern) and a time window are extracted.

Running such a discovery algorithm in practice may lead to a very large number of discovered event queries, even when ignoring obvious redundancies such as inclusion dependencies between queries. To cope with that phenomenon and to enable manual inspection of the discovered queries, the result may be filtered to obtain a representative set of queries. To this end, one may leverage clustering techniques based on syntactic and semantic similarity measures for event queries.

Finally, it should be highlighted that various kinds of domain knowledge, if available, may be incorporated to increase the effectiveness and efficiency of event query discovery. For instance, knowing that particular attributes distinguish instances of a process, then any discovered event query should contain the respective data predicates that correlate events based on these attributes.

#### 4.4 Diagnostics for Event Query Matches

This section discusses how to gain diagnostic insights into specific process behavior. If a query, derived using the methods described in Sects. 4.2 and 4.3, matches, the result needs to be interpreted as a violation of some normative behavior. Specifically, diagnostics for these matches are important, as fine-granular queries may lead to an overload of triggered alerts for certain deviations from the expected process behavior. For example, the occurrence of a single out-of-order event may trigger a large amount of *strict order* violations, even though these violations all stem from the same source.

To avoid such an overload, monitoring alerts can be filtered by identifying the earliest indicator of non-compliant behavior in a set of compliance violations. This identification requires a different approach depending on the type of violation. Here, we illustrate a diagnostics technique for the violation of two constraint types for which the respective queries have been introduced in Sect. 4.2: exclusiveness and ordering constraints. Afterwards, we discuss how these techniques may be lifted beyond the control-flow perspective.

##### 4.4.1 Diagnostics for Exclusiveness Violations

We first consider violations that stem from the exclusiveness monitoring query set  $Q_+$ . Let  $\sigma = \langle a_1, a_2, \dots, a_n \rangle$  be the sequence of recorded events in a process instance and  $+$  be the exclusiveness relation, as introduced in Sect. 4.2, of the respective process model. Then, we derive the set of violations  $V_+^n$  at the time event  $a_n$  is recorded as follows:

$$V_+^n = \{(a_x, a_y) \in + \mid a_y = a_n \wedge a_x \in \sigma\}.$$

*Trigger Violation* A single event may cause multiple exclusiveness violations. Given such a set of violations,  $V_+^n$ , the trigger of these violations is the violation that

relates to the earliest event in the sequence of recorded events  $\sigma = \langle a_1, a_2, \dots, a_n \rangle$ . Therefore, the trigger refers to the earliest event that implies that the event  $a_n$  was not allowed. We define a function *trigger* to extract the trigger for the most recent violations with respect to  $a_n$  as follows:

$$\text{trigger}(V_+^n) = (a_x, a_y) \in V_+^n \text{ such that } \forall (a_k, a_l) \in V_+^n [x \leq k].$$

We illustrate the introduced concept for the process from Fig. 2. Assume that the activities  $a$  and  $d$  have been executed, when we observe an event that signals the completion of activity  $c$ , i.e., we recorded  $\sigma = \langle a, d, c \rangle$ . The exclusiveness monitoring query set  $Q_+$  matches and identifies two violations,  $V_+^3 = \{(a, c), (d, c)\}$ . The violation of exclusiveness for  $a$  and  $c$  is the trigger, since  $a$  was the first activity to complete in the recorded event sequence, i.e.,  $\text{trigger}(V_+^3) = (a, c)$ .

*Consecutive Violation* The identification of a trigger for a set of violations triggered by a single event is the first step to structure the feedback on violations. Once a violation is identified, subsequent events may result in violations that logically follow from the violations already observed. For a trigger violation  $(a_x, a_y)$ , consecutive violations  $(a_p, a_q)$  are characterized by the fact that either (1)  $a_x$  with  $a_p$  and  $a_y$  with  $a_q$  are not conflicting or (2) this non-conflicting property is observed for  $a_x$  with  $a_q$  and  $a_y$  with  $a_p$ . Further, we have to consider the case that potentially it holds  $a_p = a_x$  or  $a_p = a_y$ . Consecutive violations are recorded but explicitly marked once they are observed. Given a trigger  $(a_x, a_y)$  of exclusiveness violations, we define the set of consecutive violations by a function *consec*.

$$\begin{aligned} \text{consec}_+(a_x, a_y) = & \{(a_p, a_q) \in + \mid ((a_x = a_p \vee a_x \not\sim a_p) \wedge (a_q = a_y \vee a_y \not\sim a_q)) \\ & \vee ((a_y = a_p \vee a_y \not\sim a_p) \wedge (a_x = a_q \vee a_x \not\sim a_q))\}. \end{aligned}$$

Reconsider the process from Fig. 2 and the recorded event sequence  $\sigma = \langle a, d, c \rangle$ . Now, assume a subsequent recording of event  $e$ . The exclusiveness monitoring query set  $Q_+$  matches and we extract a set of violations  $V_+^4 = \{(c, e)\}$  (in addition to  $V_+^3$ ) with the trigger  $\text{trigger}(V_+^4) = (c, e)$ . Apparently, this violation follows directly from the violations identified when event  $c$  has been recorded, because  $e$  is expected to occur subsequent to  $d$ . This is captured by our notion of consecutive violations for the previously identified trigger  $(a, c)$ . Since  $c$  and  $e$  are expected to be exclusive and it holds  $a_y = c = a_p$  and  $a \not\sim e$ , we observe that  $(c, e) \in \text{consec}(a, c)$ . Hence, the exclusiveness violation  $(c, e)$  would be reported as a consecutive violation of the previous violations identified by their trigger  $\text{trigger}(V_+^3) = (a, c)$ .

Further, assume that the next recorded event is  $b$ , so that  $\sigma = \langle a, d, c, e, b \rangle$  and  $V_+^5 = \{(a, b), (c, b)\}$ . Then, both violations represent a situation that does not follow logically from violations observed so far, i.e., they are non-consecutive and reported as independent violations to the analyst. Still, the feedback is structured as we identify a trigger as  $\text{trigger}(V_+^5) = (a, b)$ , since event  $a$  has occurred before event  $c$ .



#### 4.4.2 Diagnostics for Order Violations

Now, we consider violations that stem from the order monitoring query set  $Q_{\rightsquigarrow}$ . Let  $\sigma = \langle a_1, a_2, \dots, a_n \rangle$  be a sequence of recorded events in a process instance and  $\rightsquigarrow$  be the strict order relation, as defined in Sect. 4.2, of the respective process model. Let  $\rightsquigarrow^{-1}$  be the inverse relation of the strict order relation,  $(a_x \rightsquigarrow a_y) \Leftrightarrow (a_y \rightsquigarrow^{-1} a_x)$ . Then, we derive the set of violations  $V_{\rightsquigarrow}^n$  at the time event  $a_n$  is recorded as follows:

$$V_{\rightsquigarrow}^n = \{(a_x, a_y) \in \rightsquigarrow^{-1} \mid a_y = a_n \wedge a_x \in \sigma\}.$$

*Trigger Violation* For this set of violations, it may be the case that a single event causes multiple order violations. Given a set of order violations  $V_{\rightsquigarrow}^n$ , the trigger is the violation that relates to the earliest event in the sequence of recorded events  $\sigma = \langle a_1, a_2, \dots, a_n \rangle$ . Again, we define a function to extract this trigger.

$$\text{trigger}(V_{\rightsquigarrow}^n) = (a_x, a_y) \in V_{\rightsquigarrow}^n \text{ such that } \forall (a_k, a_l) \in V_{\rightsquigarrow}^n [x \leq k].$$

For illustration, consider the example of Fig. 2 and a sequence of recorded events  $\sigma = \langle a, d, h, k \rangle$ . Now,  $e$  is completed, which points to a violation of the order constraint between  $e$  and  $h$  as well as between  $e$  and  $k$ . The idea is to report the earliest event in the execution sequence, which was supposed to be executed after  $e$ . Then, the violation  $\text{trigger}(V_{\rightsquigarrow}^5) = (h, e)$  is identified as the trigger, since  $h$  has been the first event in this case.

*Consecutive Violation* As for exclusiveness constraint violations, we also define consecutive violations. These include violations from subsequent events that logically follow from violations observed earlier. For a trigger  $(a_x, a_y)$ , consecutive violations  $(a_p, a_q)$  are characterized by the fact that either  $a_x$  with  $a_p$  and  $a_y$  with  $a_q$  are in strict order or  $a_x$  with  $a_q$  and  $a_y$  with  $a_p$ , respectively. Taking into account that it may hold  $a_p = a_x$  or  $a_p = a_y$ , we lift the function *consec* to triggers of strict order violations. Given such a trigger  $(a_x, a_y)$ , it is defined as follows:

$$\begin{aligned} \text{consec}(a_x, a_y) = \{ & (a_p, a_q) \in \rightsquigarrow^{-1} \mid ((a_x = a_p \vee a_x \rightsquigarrow a_p) \wedge (a_q = a_y \vee a_y \rightsquigarrow a_q)) \\ & \vee ((a_y = a_p \vee a_y \rightsquigarrow a_p) \wedge (a_x = a_q \vee a_x \rightsquigarrow a_q)) \}. \end{aligned}$$

Consider the example of Fig. 2 and the sequence  $\sigma = \langle a, d, h, k, e \rangle$ , which resulted in  $\text{trigger}(V_{\rightsquigarrow}^5) = (h, e)$ . Now,  $g$  is observed, which violates the order with  $h$  and  $k$  as monitored by the order monitoring query  $Q_{\rightsquigarrow}$ . Apparently, this violation follows from the earlier violations. It is identified as a consecutive violation for the previous trigger  $(h, e)$ , since  $h$  is in both violations and  $e$  and  $g$  are not conflicting in terms of order, i.e.,  $(h, g) \in \text{consec}(h, e)$ . Since  $k \rightsquigarrow^{-1} g$ ,  $h \rightsquigarrow k$ , and  $e \rightsquigarrow g$ , this also holds for the second violation. Hence, violations  $(h, g)$  and  $(h, k)$  are reported as consecutive violations.

### 4.4.3 Diagnostics on the Violation Context

Next to the identification of particular events that resulted in compliance violations, it is possible to assess whether there are connections between violations and their occurrence context as reflected in data attributes associated with process instances. That is, the goal is to check for attribute values that differentiate cases with the violation from cases without the violation.

As an illustration, consider that an organization monitors the time it takes from receiving a lead to sending out a quote in their Lead-to-Quote process, particularly, they want to identify where this takes more than two weeks. By considering the data values associated with events, the organization may discover that these violations occur for a specific context, e.g., for orders stemming from quote requests originating from a particular *country* and that are related to a specific *order type*. If such information can be identified, this provides valuable diagnostic insights into the factors that are correlated with the observed delays, which may guide the efforts to resolve the issue.

The automated identification of the context of specific violations can be achieved by applying *classification* techniques on an annotated set of process instances, where two classes are used to distinguish between instances with violations and instances without. Classifiers that produce human interpretable output, such as *decision trees*, are particularly useful for this setting. These techniques can produce clear rules indicating in which contexts monitoring violations have been observed.

## 5 Discussion

This chapter outlined how Complex Event Processing methods can be leveraged for process querying that works on event-based representations of processes. In particular, we discussed how CEP can be embedded in the process querying context, by relating CEP methods to process querying use cases and the Process Querying Framework. In this way, we showed that event-based process querying can be used both for online querying, through the analysis of event streams, and for offline querying, by replaying event logs containing static event data. Moreover, we highlighted design choices of CEP models, as those govern which of these models may be appropriate for process querying in a specific application context.

We argued that using CEP methods for event-based process querying faces several challenges, which may be addressed by four essential techniques: First, event–activity correlation is used to link elements from a process specification (e.g., process model activities) to types of observed events, which represents a fundamental requirement for a broad range of analysis techniques. Second, model-based query generation can be used to automatically derive monitoring queries from process models, which enables the identification of deviations between modeled and recorded behavior. Third, we also discussed how such monitoring queries can be based on historic event data, in case a suitable process model is not available.

Fourth, we discussed techniques to analyze the matches of event queries, which can be used to gain diagnostic insights into specific process behavior. Most importantly, this includes the identification of trigger violations that represent the earliest signal of deviating process behavior.

While the aforementioned have shown that CEP methods have a range of applications in the context of process querying, open research questions remain. In relation to the techniques discussed in Sect. 4, we identify the main open directions, as follows:

- Although different techniques have been developed for event–activity correlation, it has been recognized that this problem in practice has only been resolved through the development of probabilistic techniques, which are unable to provide correlations in a deterministic manner. As a result, analysis techniques should be adapted to this so-called mapping uncertainty [62].
- Model-based query generation allows for the automatic establishment of monitoring queries from process models. However, given a sufficiently complex model, the amount of generated queries can quickly become unmanageable. Recognizing that some process violations will have a greater impact on organizations than others, the question of how to identify the semantically most relevant monitoring queries remains open.
- The problem of specifically weighting the relevance of particular queries is also present when discovering queries automatically from labeled event data. Moreover, common discovery techniques do show scalability issues. Hence, optimizations of discovery algorithms that incorporate domain knowledge on the process are a promising direction for future research.
- Having exemplified the potential of diagnostics for the matches of event queries, future work should focus on recognizing the complimentary nature of control-flow and data-aware techniques. A combination of these types of techniques would be able, for instance, to identify particular control-flow-related deviations that only occur if the throughput time or other data-based values are above a particular threshold. This is currently not achieved, because the control-flow and data perspectives are only considered in isolation.

**Reprint** Figure 2 is reprinted with permission from M. Weidlich, H. Ziekow, J. Mendling, O. Günther, M. Weske, and N. Desai. *Event-Based Monitoring of Process Execution Violations*. 9th International Conference on Business Process Management. Springer, 2011. pp 182–198 (“© Springer-Verlag Berlin Heidelberg 2011”).

## References

1. Adi, A., Etzion, O.: Amit - the situation manager. *VLDB J.* **13**(2), 177–203 (2004). <https://doi.org/10.1007/s00778-003-0108-y>
2. Agirre, E., Alfonseca, E., Hall, K., Kravalova, J., Paşca, M., Soroa, A.: A study on similarity and relatedness using distributional and WordNet-based approaches. In: *Proceedings of Human*

- Language Technologies: The 2009 Annual Conference of the North American Chapter of the Association for Computational Linguistics, pp. 19–27. Association for Computational Linguistics (2009)
3. Akidau, T., Bradshaw, R., Chambers, C., Chernyak, S., Fernández-Moctezuma, R., Lax, R., McVeety, S., Mills, D., Perry, F., Schmidt, E., Whittle, S.: The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *PVLDB* **8**(12), 1792–1803 (2015). <https://doi.org/10.14778/2824032.2824076>. <http://www.vldb.org/pvldb/vol8/p1792-Akidau.pdf>
  4. Algergawy, A., Nayak, R., Saake, G.: Element similarity measures in xml schema matching. *Information Sciences* **180**(24), 4975–4998 (2010)
  5. Anicic, D., Rudolph, S., Fodor, P., Stojanovic, N.: Stream reasoning and complex event processing in ETALIS. *Semantic Web* **3**(4), 397–407 (2012). <https://doi.org/10.3233/SW-2011-0053>
  6. Appel, S., Kleber, P., Frischbier, S., Freudenreich, T., Buchmann, A.P.: Modeling and execution of event stream processing in business processes. *Inf. Syst.* **46**, 140–156 (2014). <https://doi.org/10.1016/j.is.2014.04.002>
  7. Arasu, A., Babcock, B., Babu, S., Cieslewicz, J., Datar, M., Ito, K., Motwani, R., Srivastava, U., Widom, J.: STREAM: The Stanford data stream management system. In: Garofalakis et al. [28], pp. 317–336. [https://doi.org/10.1007/978-3-540-28608-0\\_16](https://doi.org/10.1007/978-3-540-28608-0_16)
  8. Artikis, A., Sergot, M.J., Paliouras, G.: An event calculus for event recognition. *IEEE Trans. Knowl. Data Eng.* **27**(4), 895–908 (2015). <https://doi.org/10.1109/TKDE.2014.2356476>
  9. Artikis, A., Margara, A., Ugarte, M., Vansummeren, S., Weidlich, M.: Complex event recognition languages: Tutorial. In: Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems, DEBS 2017, Barcelona, Spain, June 19–23, 2017, pp. 7–10. ACM (2017). <https://doi.org/10.1145/3093742.3095106>
  10. Backmann, M., Baumgrass, A., Herzberg, N., Meyer, A., Weske, M.: Model-driven event query generation for business process monitoring. In: Lomuscio, A., Nepal, S., Patrizi, F., Benatallah, B., Brandic, I. (eds.) *Service-Oriented Computing - ICSOC 2013 Workshops - CCSA, CSB, PASCEB, SWESE, WESOA, and PhD Symposium*, Berlin, Germany, December 2–5, 2013. Revised Selected Papers. *Lecture Notes in Computer Science*, vol. 8377, pp. 406–418. Springer (2013). [https://doi.org/10.1007/978-3-319-06859-6\\_36](https://doi.org/10.1007/978-3-319-06859-6_36)
  11. Baier, T., Mendling, J., Weske, M.: Bridging abstraction layers in process mining. *Information Systems* **46**, 123–139 (2014)
  12. Baier, T., Rogge-Solti, A., Weske, M., Mendling, J.: Matching of events and activities - an approach based on constraint satisfaction. In: *IFIP Working Conference on The Practice of Enterprise Modeling*, pp. 58–72. Springer (2014)
  13. Baier, T., Di Ciccio, C., Mendling, J., Weske, M.: Matching events and activities by integrating behavioral aspects and label analysis. *Softw. Syst. Model.*, 1–26 (2017)
  14. Barga, R.S., Goldstein, J., Ali, M.H., Hong, M.: Consistent streaming through time: A vision for event stream processing. In: *CIDR 2007, Third Biennial Conference on Innovative Data Systems Research*, Asilomar, CA, USA, January 7–10, 2007, Online Proceedings, pp. 363–374. [www.cidrdb.org](http://cidrdb.org) (2007). <http://cidrdb.org/cidr2007/papers/cidr07p42.pdf>
  15. Bose, R.J.C., Van Der Aalst, W.M., Zliobaite, I., Pechenizkiy, M.: Dealing with concept drifts in process mining. *IEEE Trans. Neural Netw. Learn. Syst.* **25**(1), 154–171 (2014)
  16. Brenna, L., Demers, A.J., Gehrke, J., Hong, M., Ossher, J., Panda, B., Riedewald, M., Thatte, M., White, W.M.: Cayuga: a high-performance event processing engine. In: Chan, C.Y., Ooi, B.C., Zhou, A. (eds.) *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Beijing, China, June 12–14, 2007, pp. 1100–1102. ACM (2007). <https://doi.org/10.1145/1247480.1247620>
  17. Burattin, A., Sperduti, A., van der Aalst, W.M.P.: Control-flow discovery from event streams. In: *Proceedings of the IEEE Congress on Evolutionary Computation, CEC 2014*, Beijing, China, July 6–11, 2014, pp. 2420–2427. IEEE (2014). <https://doi.org/10.1109/CEC.2014.6900341>

18. Carmona, J., van Dongen, B.F., Solti, A., Weidlich, M.: Conformance Checking - Relating Processes and Models. Springer (2018). <https://doi.org/10.1007/978-3-319-99414-7>
19. Cugola, G., Margara, A.: TESLA: a formally defined event specification language. In: Bacon, J., Pietzuch, P.R., Sventek, J., Çetintemel, U. (eds.) Proceedings of the Fourth ACM International Conference on Distributed Event-Based Systems, DEBS 2010, Cambridge, United Kingdom, July 12–15, 2010, pp. 50–61. ACM (2010). <https://doi.org/10.1145/1827418.1827427>
20. Cugola, G., Margara, A.: Processing flows of information: From data stream to complex event processing. *ACM Comput. Surv.* **44**(3), 15:1–15:62 (2012). <https://doi.org/10.1145/2187671.2187677>
21. del-Río-Ortega, A., Resinas, M., Cabanillas, C., Cortés, A.R.: On the definition and design-time analysis of process performance indicators. *Inf. Syst.* **38**(4), 470–490 (2013). <https://doi.org/10.1016/j.is.2012.11.004>
22. Ding, L., Chen, S., Rundensteiner, E.A., Tatemura, J., Hsiung, W., Candan, K.S.: Runtime semantic query optimization for event stream processing. In: Alonso, G., Blakeley, J.A., Chen, A.L.P. (eds.) Proceedings of the 24th International Conference on Data Engineering, ICDE 2008, April 7–12, 2008, Cancún, Mexico, pp. 676–685. IEEE Computer Society (2008). <https://doi.org/10.1109/ICDE.2008.4497476>
23. Dumas, M., García-Bañuelos, L., Polyvyanyy, A., Yang, Y., Zhang, L.: Aggregate quality of service computation for composite services. In: Maglio, P.P., Weske, M., Yang, J., Fantinato, M. (eds.) Service-Oriented Computing - 8th International Conference, ICSOC 2010, San Francisco, CA, USA, December 7–10, 2010. Proceedings. Lecture Notes in Computer Science, vol. 6470, pp. 213–227 (2010). [https://doi.org/10.1007/978-3-642-17358-5\\_15](https://doi.org/10.1007/978-3-642-17358-5_15)
24. Dumas, M., Rosa, M.L., Mendling, J., Reijers, H.A.: Fundamentals of Business Process Management, 2nd edn. Springer (2018). <https://doi.org/10.1007/978-3-662-56509-4>
25. EsperTech: Esper documentation (2018). <http://www.espertech.com/esper/esper-documentation/>
26. Etzion, O., Niblett, P.: Event Processing in Action. Manning Publications Company (2010). <http://www.manning.com/etzion/>
27. Fettke, P., Loos, P., Zwicker, J.: Business process reference models: Survey and classification. In: Bussler, C., Haller, A. (eds.) Business Process Management Workshops, BPM 2005 International Workshops, BPI, BPD, ENEI, BPRM, WSCOBPM, BPS, Nancy, France, September 5, 2005, Revised Selected Papers, vol. 3812, pp. 469–483 (2005). [https://doi.org/10.1007/11678564\\_44](https://doi.org/10.1007/11678564_44)
28. Garofalakis, M.N., Gehrke, J., Rastogi, R. (eds.): Data Stream Management - Processing High-Speed Data Streams. Data-Centric Systems and Applications. Springer (2016). <https://doi.org/10.1007/978-3-540-28608-0>
29. George, L., Cadonna, B., Weidlich, M.: Il-miner: Instance-level discovery of complex event patterns. *PVLDB* **10**(1), 25–36 (2016). <https://doi.org/10.14778/3015270.3015273>. <http://www.vldb.org/pvldb/vol10/p25-weidlich.pdf>
30. Grez, A., Riveros, C., Ugarte, M.: A formal framework for complex event processing. In: Barcelo, P., Calautti, M. (eds.) 22nd International Conference on Database Theory (ICDT 2019), Leibniz International Proceedings in Informatics (LIPIcs), vol. 127, pp. 5:1–5:18. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany (2019). <https://doi.org/10.4230/LIPIcs.ICDT.2019.5>. <http://drops.dagstuhl.de/opus/volltexte/2019/10307>
31. Hassani, M., Siccha, S., Richter, F., Seidl, T.: Efficient process discovery from event streams using sequential pattern mining. In: IEEE Symposium Series on Computational Intelligence, SSCI 2015, Cape Town, South Africa, December 7–10, 2015, pp. 1366–1373. IEEE (2015). <https://doi.org/10.1109/SSCI.2015.195>
32. He, Y., Barman, S., Naughton, J.F.: On load shedding in complex event processing. In: Schweikardt, N., Christophides, V., Leroy, V. (eds.) Proc. 17th International Conference on Database Theory (ICDT), Athens, Greece, March 24–28, 2014, pp. 213–224. OpenProceedings.org (2014). <https://doi.org/10.5441/002/icdt.2014.23>

33. Hinze, A., Voisard, A.: EVA: an event algebra supporting complex event specification. *Inf. Syst.* **48**, 1–25 (2015). <https://doi.org/10.1016/j.is.2014.07.003>
34. Hirzel, M., Baudart, G.: *Stream Processing Languages and Abstractions*, pp. 1–8. Springer International Publishing, Cham (2018). [https://doi.org/10.1007/978-3-319-63962-8\\_260-1](https://doi.org/10.1007/978-3-319-63962-8_260-1)
35. Hirzel, M., Soulé, R., Schneider, S., Gedik, B., Grimm, R.: A catalog of stream processing optimizations. *ACM Comput. Surv.* **46**(4), 46:1–46:34 (2013). <https://doi.org/10.1145/2528412>
36. ISO/IEC: Information technology - Database languages - SQL Technical Reports - Part 5: Row Pattern Recognition in SQL. TR 19075-5
37. Katzouris, N., Artikis, A., Paliouras, G.: Online learning of event definitions. *TPLP* **16**(5-6), 817–833 (2016). <https://doi.org/10.1017/S1471068416000260>
38. Katzouris, N., Artikis, A., Paliouras, G.: Parallel online event calculus learning for complex event recognition. *Futur. Gener. Comput. Syst.* **94**, 468–478 (2019). <https://doi.org/10.1016/j.future.2018.11.033>
39. Kunze, M., Weske, M.: *Behavioural Models - From Modelling Finite Automata to Analysing Business Processes*. Springer (2016). <https://doi.org/10.1007/978-3-319-44960-9>
40. Li, M., Mani, M., Rundensteiner, E.A., Lin, T.: Complex event pattern detection over streams with interval-based temporal semantics. In: Eysers, D.M., Etzion, O., Gal, A., Zdonik, S.B., Vincent, P. (eds.) *Proceedings of the Fifth ACM International Conference on Distributed Event-Based Systems, DEBS 2011, New York, NY, USA, July 11–15, 2011*, pp. 291–302. ACM (2011). <https://doi.org/10.1145/2002259.2002297>
41. Lin, D.: An information-theoretic definition of similarity. In: *ICML*, vol. 98, pp. 296–304 (1998)
42. Liu, M., Li, M., Golovnya, D., Rundensteiner, E.A., Claypool, K.T.: Sequence pattern query processing over out-of-order event streams. In: Ioannidis, Y.E., Lee, D.L., Ng, R.T. (eds.) *Proceedings of the 25th International Conference on Data Engineering, ICDE 2009, March 29 2009–April 2 2009, Shanghai, China*, pp. 784–795. IEEE Computer Society (2009). <https://doi.org/10.1109/ICDE.2009.95>
43. Ly, L.T., Maggi, F.M., Montali, M., Rinderle-Ma, S., van der Aalst, W.M.P.: Compliance monitoring in business processes: Functionalities, application, and tool-support. *Inf. Syst.* **54**, 209–234 (2015). <https://doi.org/10.1016/j.is.2015.02.007>
44. Mannhardt, F., De Leoni, M., Reijers, H.A., Van Der Aalst, W.M., Toussaint, P.J.: From low-level events to activities—a pattern-based approach. In: *International Conference on Business Process Management*, pp. 125–141. Springer (2016)
45. Mannhardt, F., de Leoni, M., Reijers, H.A., van der Aalst, W.M.P.: Balanced multi-perspective checking of process conformance. *Computing* **98**(4), 407–437 (2016). <https://doi.org/10.1007/s00607-015-0441-1>
46. Mans, R., van der Aalst, W.M.P., Vanwersch, R.J.B.: *Process Mining in Healthcare - Evaluating and Exploiting Operational Healthcare Processes*. Springer Briefs in Business Process Management. Springer (2015). <https://doi.org/10.1007/978-3-319-16071-9>
47. Margara, A., Cugola, G., Tamburrelli, G.: Learning from the past: automated rule generation for complex event processing. In: Bellur, U., Kothari, R. (eds.) *The 8th ACM International Conference on Distributed Event-Based Systems, DEBS '14, Mumbai, India, May 26–29, 2014*, pp. 47–58. ACM (2014). <https://doi.org/10.1145/2611286.2611289>
48. Mei, Y., Madden, S.: ZStream: a cost-based query processor for adaptively detecting composite events. In: Çetintemel, U., Zdonik, S.B., Kossmann, D., Tatbul, N. (eds.) *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2009, Providence, Rhode Island, USA, June 29–July 2, 2009*, pp. 193–206. ACM (2009). <https://doi.org/10.1145/1559845.1559867>
49. Oliner, A., Ganapathi, A., Xu, W.: Advances and challenges in log analysis. *Commun. ACM* **55**(2), 55–61 (2012)
50. Polyvyanyy, A., Weidlich, M., Conforti, R., Rosa, M.L., ter Hofstede, A.H.M.: The 4c spectrum of fundamental behavioral relations for concurrent systems. In: Ciardo, G., Kindler, E. (eds.) *Application and Theory of Petri Nets and Concurrency - 35th International Conference, PETRI*

- NETS 2014, Tunis, Tunisia, June 23–27, 2014. Proceedings. Lecture Notes in Computer Science, vol. 8489, pp. 210–232. Springer (2014). [https://doi.org/10.1007/978-3-319-07734-5\\_12](https://doi.org/10.1007/978-3-319-07734-5_12)
51. Polyvyanyy, A., Armas-Cervantes, A., Dumas, M., García-Bañuelos, L.: On the expressive power of behavioral profiles. *Formal Asp. Comput.* **28**(4), 597–613 (2016). <https://doi.org/10.1007/s00165-016-0372-4>
  52. Polyvyanyy, A., Ouyang, C., Barros, A., van der Aalst, W.M.P.: Process querying: Enabling business intelligence through query-based process analytics. *Decis. Support Syst.* **100**, 41–56 (2017). <https://doi.org/10.1016/j.dss.2017.04.011>
  53. Ray, M., Lei, C., Rundensteiner, E.A.: Scalable pattern sharing on event streams. In: Özcan, F., Koutrika, G., Madden, S. (eds.) Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26–July 01, 2016, pp. 495–510. ACM (2016). <https://doi.org/10.1145/2882903.2882947>
  54. Rogge-Solti, A., Kasneci, G.: Temporal anomaly detection in business processes. In: Sadiq, S.W., Soffer, P., Völzer, H. (eds.) Business Process Management - 12th International Conference, BPM 2014, Haifa, Israel, September 7–11, 2014. Proceedings. Lecture Notes in Computer Science, vol. 8659, pp. 234–249. Springer (2014). [https://doi.org/10.1007/978-3-319-10172-9\\_15](https://doi.org/10.1007/978-3-319-10172-9_15)
  55. Sadiq, S., Governatori, G., Namiri, K.: Modeling control objectives for business process compliance. In: International Conference on Business Process Management, pp. 149–164. Springer (2007)
  56. Senderovich, A., Rogge-Solti, A., Gal, A., Mendling, J., Mandelbaum, A.: The road from sensor data to process instances via interaction mining. In: International Conference on Advanced Information Systems Engineering, pp. 257–273. Springer (2016)
  57. Teymourian, K., Rohde, M., Paschke, A.: Fusion of background knowledge and streams of events. In: Bry, F., Paschke, A., Eugster, P.T., Fetzer, C., Behrend, A. (eds.) Proceedings of the Sixth ACM International Conference on Distributed Event-Based Systems, DEBS 2012, Berlin, Germany, July 16–20, 2012, pp. 302–313. ACM (2012). <https://doi.org/10.1145/2335484.2335517>
  58. Van der Aalst, W.M.P., Weijters, T., Maruster, L.: Workflow mining: Discovering process models from event logs. *IEEE Trans. Knowl. Data Eng.* (9), 1128–1142 (2004)
  59. Van der Aalst, W.M.P., van Hee, K.M., van der Werf, J.M.E.M., Kumar, A., Verdonk, M.: Conceptual model for online auditing. *Decis. Support Syst.* **50**(3), 636–647 (2011). <https://doi.org/10.1016/j.dss.2010.08.014>
  60. Van der Aalst, W.M.P.: Process Mining - Data Science in Action, Second Edition. Springer (2016). <https://doi.org/10.1007/978-3-662-49851-4>
  61. Van der Aa, H., Gal, A., Leopold, H., Reijers, H.A., Sagi, T., Shraga, R.: Instance-based process matching using event-log information. In: International Conference on Advanced Information Systems Engineering, pp. 283–297. Springer (2017)
  62. Van der Aa, H., Leopold, H., Reijers, H.A.: Checking process compliance on the basis of uncertain event-to-activity mappings. In: International Conference on Advanced Information Systems Engineering, pp. 79–93. Springer (2017)
  63. Van der Aa, H.: Comparing and Aligning Process Representations: Foundations and Technical Solutions, vol. 323. Springer (2018)
  64. Wang, J., Han, J.: BIDE: efficient mining of frequent closed sequences. In: Özsoyoglu, Z.M., Zdonik, S.B. (eds.) Proceedings of the 20th International Conference on Data Engineering, ICDE 2004, 30 March–2 April 2004, Boston, MA, USA, pp. 79–90. IEEE Computer Society (2004). <https://doi.org/10.1109/ICDE.2004.1319986>
  65. Weidlich, M., Mendling, J., Weske, M.: Efficient consistency measurement based on behavioral profiles of process models. *IEEE Trans. Softw. Eng.* **37**(3), 410–429 (2011). <https://doi.org/10.1109/TSE.2010.96>
  66. Weidlich, M., Polyvyanyy, A., Mendling, J., Weske, M.: Causal behavioural profiles—efficient computation, applications, and evaluation. *Fundamenta Informaticae* **113**(3–4), 399–435 (2011)

67. Weidlich, M., Ziekow, H., Mendling, J., Günther, O., Weske, M., Desai, N.: Event-based monitoring of process execution violations. In: Rinderle-Ma, S., Toumani, F., Wolf, K. (eds.) *Business Process Management - 9th International Conference, BPM 2011, Clermont-Ferrand, France, August 30–September 2, 2011*. Proceedings. Lecture Notes in Computer Science, vol. 6896, pp. 182–198. Springer (2011). [https://doi.org/10.1007/978-3-642-23059-2\\_16](https://doi.org/10.1007/978-3-642-23059-2_16)
68. Weidlich, M., Ziekow, H., Gal, A., Mendling, J., Weske, M.: Optimizing event pattern matching using business process models. *IEEE Trans. Knowl. Data Eng.* **26**(11), 2759–2773 (2014). <https://doi.org/10.1109/TKDE.2014.2302306>
69. White, W.M., Riedewald, M., Gehrke, J., Demers, A.J.: What is “next” in event processing? In: Libkin, L. (ed.) *Proceedings of the Twenty-Sixth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 11–13, 2007, Beijing, China*, pp. 263–272. ACM (2007). <https://doi.org/10.1145/1265530.1265567>
70. Yan, X., Han, J., Afshar, R.: CloSpan: Mining closed sequential patterns in large datasets. In: Barbará, D., Kamath, C. (eds.) *Proceedings of the Third SIAM International Conference on Data Mining, San Francisco, CA, USA, May 1–3, 2003*, pp. 166–177. SIAM (2003). <https://doi.org/10.1137/1.9781611972733.15>
71. Yujian, L., Bo, L.: A normalized Levenshtein distance metric. *IEEE Trans. Pattern Anal. Mach. Intell.* **29**(6), 1091–1095 (2007)
72. Zeng, K., Yang, M., Mozafari, B., Zaniolo, C.: Complex pattern matching in complex structures: The XSeq approach. In: Jensen, C.S., Jermaine, C.M., Zhou, X. (eds.) *29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8–12, 2013*, pp. 1328–1331. IEEE Computer Society (2013). <https://doi.org/10.1109/ICDE.2013.6544936>
73. Zhang, H., Diao, Y., Immerman, N.: On complexity and optimization of expensive queries in complex event processing. In: Dyreson, C.E., Li, F., Özsu, M.T. (eds.) *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22–27, 2014*, pp. 217–228. ACM (2014). <https://doi.org/10.1145/2588555.2593671>
74. Zhao, B., Hung, N.Q.V., Weidlich, M.: Load shedding for complex event processing: Input-based and state-based techniques. In: *36th IEEE International Conference on Data Engineering, ICDE 2020, Dallas, TX, USA, April 20–24, 2020*, pp. 1093–1104. IEEE (2020). <https://doi.org/10.1109/ICDE48307.2020.00099>



# Process Querying: Methods, Techniques, and Applications



Artem Polyvyanyy

**Abstract** Process querying studies concepts and methods from fields like Big data, process modeling and analysis, business process intelligence, and process analytics and applies them to retrieve and manipulate real-world and designed processes. This chapter reviews state-of-the-art methods for process querying, summarizes techniques used to implement process querying methods, discusses typical applications of process querying, and identifies research gaps and suggests directions for future research in process querying.

## 1 Introduction

Process querying aims to combine concepts studied by disciplines that look into the use of large and complex datasets, like Big data, and research areas that investigate aspects related to process modeling and analysis, like business process management, process mining, business process intelligence, and process analytics to develop methods and tools for automatically manipulating, e.g., redesigning and optimizing, real-world and designed processes, and systematically extracting process-related information for subsequent use [28]. A *process querying method* is a technique that given a collection of processes and a process query, i.e., a statement that articulates a process-related information need or specifies an instruction for process manipulations, systematically implements the query over the processes.

The idea of process querying started to shape at the beginning of the twenty-first century. However, the lion's share of concepts, principles, and methods for process querying appeared only recently. The reason for this is at least twofold. First, recent large-scale digitization of real-world processes governed by organizations has led to the generation of large volumes of digital footprints of real-world processes and supporting data. Second, increasing evidence of added value due to the use of data

---

A. Polyvyanyy (✉)

School of Computing and Information Systems, Faculty of Engineering and Information Technology, The University of Melbourne, Melbourne, VIC, Australia

e-mail: [artem.polyvyanyy@unimelb.edu.au](mailto:artem.polyvyanyy@unimelb.edu.au)

© Springer Nature Switzerland AG 2022

A. Polyvyanyy (ed.), *Process Querying Methods*,

[https://doi.org/10.1007/978-3-030-92875-9\\_18](https://doi.org/10.1007/978-3-030-92875-9_18)

511

generated by processes [31], as well as of information about the processes in forms of models and ontologies, for improving future real-world processes has spawned research on new analysis techniques aimed to understand and interpret process phenomena, giving the birth to the discipline of Process Science.

This chapter gives an overview of the state-of-the-art methods for process querying. The methods are classified into those for querying observed and recorded processes, i.e., event logs, as studied in process mining [40], process models, either automatically discovered from event logs using process mining algorithms or manually designed, as traditionally studied in business process management [11, 42], and such that address the querying of both logs and models. Besides, the chapter discusses techniques that underpin the existing process querying methods and practical applications of process querying methods as recommended and employed by academia and industry. The content of this chapter is based on the literature reviews reported in [22, 28] and extends them by further insights reported by the contributors of this book.

The rest of this chapter unfolds as follows. The next section provides the foundations necessary for following the subsequent discussions. Section 3 discusses the existing methods for process querying. Next, Sect. 4 gives an overview of techniques that are commonly used to implement process querying methods. Section 5 summarizes the main applications of process querying, as recommended or evaluated by the authors of the process querying methods. Finally, Sect. 6 summarizes research gaps and suggests directions for future research in process querying.

## 2 Foundations

This section lists basic concepts in process querying that are necessary to support the subsequent discussions. Let  $\mathcal{U}_{an}$  and  $\mathcal{U}_{av}$  be the universe of *attribute names* and *attribute values*, respectively.

**Event.** An *event*  $e$  is a relation between some attribute names and attribute values with the property that each attribute name that participates in the relation is related to exactly one attribute value, i.e., it is a partial function  $e : \mathcal{U}_{an} \rightarrow \mathcal{U}_{av}$ .

For example,  $e = \{(case, 120327), (time, 2020-02-03T00:27:29Z), (act, \text{“Open claim”})\}$  is an event with three attributes. A possible interpretation of these attributes is that event  $e$  belongs to the process with case identifier  $e(case) = 120327$ , was recorded at timestamp  $e(time) = 2019-10-22T11:37:21Z$  (ISO 8601), and was generated by the activity with name  $e(act) = \text{“Open claim”}$ .

**Process.** A *process*  $\pi$  is a partially ordered set  $(E, \leq)$ , where  $E$  is a set of events and  $\leq$  is a partial order over  $E$ .

**Trace.** A *trace*  $\tau$  is a process  $(E, <)$ , where  $<$  is a total order over  $E$ .

A trace is often specified as a sequence of events with the same case identifier ordered by timestamps in ascending order. Hence, a trace groups events that were generated by the same process instance. For example,  $(e_1, e_2, e_3)$  is a trace composed of three events. In a process, in general, some events may be unordered to reflect that they are independent and, thus, can be, or were already, executed simultaneously.

**Behavior.** A *behavior*  $b$  is a multiset of processes.

The fact that a behavior is a multiset of processes can be used, for example, when describing that a process was observed and recorded several times. Behavior  $b_1 = [\tau_1^9, \tau_2^3]$ , where  $\tau_1 = \{(act, \text{“Open claim”}), (act, \text{“Review claim”}), (act, \text{“Close claim”})\}$  and  $\tau_2 = \{(act, \text{“Open claim”}), (act, \text{“Close claim”})\}$ , specifies that a claim is opened, then reviewed, and finally closed nine times and is opened and then immediately closed three times.

**Behavior model.** A *behavior model* is a simplified representation of real-world or envisioned behaviors, and relationships between the behaviors, that serves a particular purpose for a target audience.

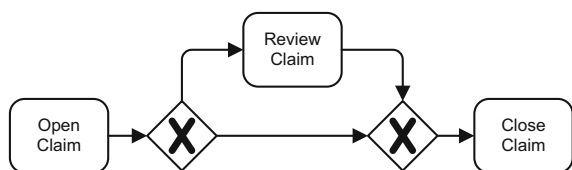
The model in Fig. 1 is a model of an envisioned behavior captured as a BPMN diagram. According to the semantics of BPMN, the model specifies behavior composed of traces  $\tau_1$  and  $\tau_2$ , whereas behavior  $b_1$  is a possible description of the observed behavior induced by the executions of the diagram. In [28], we suggest several classes of behavior models. According to this classification, the diagram in Fig. 1 identifies a *process model*, while behavior  $b_1$  identifies an *event log*. In general, a process model describes a potentially infinite collection of envisioned processes, whereas an event log captures a finite collection of already executed processes. The reader can refer to [28] for further details on the different classes of behavior models.

**Process repository.** A *process repository* is an organized collection of behavior models.

For example, a process repository can be composed of process models and/or logs organized in a folder hierarchy.

**Process query.** A *process query* is a statement that describes an information need in a process repository or specifies an instruction to manipulate a process repository.

Fig. 1 A process model



A sample process query  $q$  can specify an instruction to retrieve all process models from a repository that describe behaviors with processes in which activity “Review claim” occurs.

Let  $\mathcal{U}_{pr}$  and  $\mathcal{U}_{pq}$  be the universe of *process repositories* and the universe of *process queries*, respectively.

**Process querying method.** A *process querying method*  $m$  is a relation between ordered pairs, where in each pair the first entry is a process repository and the second entry is a process query, and process repositories with the property that each pair is related to exactly one process repository, i.e., it is a function  $m : \mathcal{U}_{pr} \times \mathcal{U}_{pq} \rightarrow \mathcal{U}_{pr}$ .

Therefore, a process querying method maps an input process repository and a process query onto a resulting process repository obtained by implementing the query statement on the input repository. For instance, given that the model in Fig. 1 is contained in the input repository, query  $q$  mentioned above will result in a repository that contains the model.

### 3 Process Querying Methods

This section summarizes all the process querying methods covered in this book. The methods are grouped based on the types of *behavior models* that can be taken as input in *process repositories* and are referred to by the short names of the corresponding languages for specifying *process queries*.

#### 3.1 Log Querying

Process querying methods discussed in this section operate over event logs.

**BP-SPARQL** BP-SPARQL is a textual language for summarizing and analyzing process execution data, for example, event logs [3–6]. The language extends SPARQL with constructs for querying Big Process Data described in an RDF graph of process-related entities. Such process-related entities are, for instance, events, actors, and relationships. Examples of relationships include the ordering relations between events and relations between entities and their attributes. The language is implemented using Hadoop, Map Reduce, and Pig-Latin technologies. Being an extension of SPARQL, BP-SPARQL allows querying using standard SPARQL capabilities.

**DAPOQ-Lang** The Data-Aware Process Oriented Query Language (DAPOQ-Lang) is a textual language for retrieving sub-logs of event logs and querying data constraints [8, 9]. In this way, DAPOQ-Lang aims to support answering business

questions that arise in process mining [40]. DAPOQ-Lang is an SQL wrapper that aims to simplify SQL queries by defining and operating over a concrete meta-model for representing event log data, called the OpenSLEX meta-model. Hence, the focus is on the usability of the language as compared to corresponding SQL queries. The language supports querying over events in traces and related data objects, their versions, and data schemas, as well as over the temporal properties of all these elements.

**PIQL** Process Instance Query Language (PIQL) is a textual language for computing Boolean and numeric process performance indicators over traces and instances of process tasks [25]. The language aims to support business users in decision-making and monitoring and management of business processes. PIQL is user friendly to nonexperts, as in addition to machine-readable, it also offers a human-readable syntax formulated in terms of natural language statements. The language can be used to define decision logic that depends on the information kept in the historical traces and used during future process execution. PIQL can be integrated with Decision Model and Notation (DMN) to support process decisions, obtain measurements to display on process dashboards, and support the dataflow.

### 3.2 *Model Querying*

This section summarizes methods for querying process models. These methods can be split into those that operate over process model specifications and those that target behaviors encoded in process models. The former methods can be further split into two subclasses. The first subclass comprises methods originally proposed for querying general conceptual models and reported useful for querying process models. DMQL and VM\* belong to this subclass. The second subclass consists of dedicated techniques for querying process models. BPMN VQL and Descriptive PQL are languages that belong to the second subclass. Finally, CRL, QuBPAL, and PQL operate over behaviors encoded in process models.

**DMQL** The Diagrammed Model Query Language (DMQL) is a visual language for querying collections of conceptual models created in arbitrary graph-based modeling languages [10]. The language also supports approximate analysis of the executions of process models in the presence of loop structures. DMQL querying is implemented as searching for model subgraphs that correspond to a predefined pattern query. DMQL is proposed to query process models, but can also be used to query data models, organizational charts, and other model types.

**VM\*** The Visual Model Manipulation Language (VM\*) is a family of languages for expressing queries (VMQL), constraints (VMCL), and transformations (VMTL) over conceptual models [1, 37, 38]. The authors of VM\* languages advocate their application over process models, for example, UML Activity Diagrams and BPMN models. VM\* extends the meta-model of the host language with several intuitive

annotations. Such an approach broadens the usability of VM\* queries, as they resemble models captured in the host language, which decreases the user effort for reading and writing queries. For example, VMQL for querying BPMN models subsumes the syntax and notation of BPMN. The matching of VM\* queries is implemented through pattern matching over model graphs.

**BPMN VQL** BPMN Visual Query Language (BPMN VQL) is a visual process query language. BPMN VQL can be used to retrieve structural information about the queried models and knowledge related to the domain of the models. BPMN VQL queries follow the structure of SQL, borrow the syntax from BPMN, while the semantics of BPMN VQL queries is grounded in SPARQL [14]. A BPMN VQL query consists of two parts. The matching part of the query specifies a structural condition to match in process models, whereas the selection part specifies parts of models to retrieve as a query result. When executed, BPMN VQL queries get translated to SPARQL using a formalization of the BPMN meta-model as an RDF ontology. The authors of the language have conducted empirical studies that demonstrate that BPMN VQL queries are easier to understand than natural language queries and that it is easier to formulate BPMN VQL queries than to match natural language queries against process models.

**Descriptive PQL** Descriptive Process Query Language (Descriptive PQL) is a textual language for retrieving process models and specifying abstractions and changes over process models [17]. The manipulations on models are defined using Single-Entry-Single-Exit subgraphs [23] and implemented via translations to the Cypher graph query language, where Cypher is a declarative query language for querying and changing graphs stored in graph database [21].

**QuBPAL** Querying Business Process Abstract modeling Language (QuBPAL) is a textual ontology-based language for retrieving process fragments and their subsequent reuse, e.g., when constructing fresh process models. QuBPAL queries resemble SPARQL queries and are executed over collections of BPMN process models, their meta-model and behavioral semantics, domain knowledge, and semantic annotations. When executed, QuBPAL queries are translated into Logic Programming queries and evaluated using the Prolog inference engine [33–35]. The authors suggest that QuBPAL can also be used for querying at run-time over running process instances and over the logs of completed processes.

**CRL** Compliance Request Language (CRL) is a process query language grounded in temporal logic designed to support standard process compliance rules [13]. Specifically, CRL supports process compliance rules that address control flow, resource, and temporal aspects of business processes. CRL queries are executed over BPEL specifications by translating the queries to LTL or MTL and then model checking the resulting temporal logic properties over BPEL specifications using the SPIN model checker [16]. CRL is designed with the relevance and usability of the supported queries in mind. For instance, the control flow compliance rules are grounded in the patterns identified in a comprehensive survey [12].

**PQL** Process Query Language (PQL) is a textual query language based on executions and behavioral relations, e.g., ordering, mutual exclusion, and concurrency, between tasks in executions of process models [26, 29, 30]. PQL reuses parts of the abstract syntax of APQL and has an SQL-like concrete syntax. The semantics of PQL is defined over all possible executions of process models and is independent of notations used to describe process models.

The core of PQL is grounded in the behavioral relations of the 4C spectrum [24]—a systematic collection of the co-occurrence, conflict, causality, and concurrency relations. In addition, PQL can reason at the level of process scenario templates (a.k.a. sample process executions with placeholders). For example, one can retrieve models from a process model collection that can execute a specified process scenario or augment models to describe a fresh process scenario template. The latter query type is implemented in PQL as a solution to the process model repair problem [27].

### 3.3 *Log and Model Querying*

Methods from this section can be used to query process models and event logs.

**BPQL** Business Process Query Language (BPQL) is a textual language for querying over process models and event logs [18, 19]. It aims at making process specification more flexible by defining such elements as resource assignment and transition conditions via BPQL queries evaluated during process execution. The language is defined as an extension of the Stack-Based Query Language (SBQL) [39]. The semantics of the language is defined over the proposed abstract model for capturing process specifications and execution traces. The authors of BPQL proposed BPQL templates for monitoring process execution and integrated the language with the BPMN standard. BPQL can be used to aggregate information over the attributes of tasks, for instance, to compute the cost of a trace based on the costs of individual tasks in the trace.

**Celonis PQL** Celonis Process Query Language (Celonis PQL) is a domain-specific language that operates over a process data model that combines data about a process of interest from various systems into one snowflake schema [41]; snowflake schemas are often used to develop data warehouses. Celonis PQL is designed for business users and aims to support process discovery, enhancement, and monitoring, three well-studied problems in process mining [40]. Business users can use Celonis PQL to formalize their process questions and execute them automatically to gain valuable process-related insights. The language supports over 150 operators, including process-related functions, machine learning, and mathematical operators. The syntax of the language resembles SQL. Thousands of users from different industries use Celonis PQL daily.

## 4 Process Querying Techniques

Existing process querying methods are often founded on well-established techniques. Some most commonly used techniques used to implement process querying are summarized in this section.

**Structural Analysis** Behavior models like process models or event logs can be formalized as graphs. Several methods perform process querying by analyzing structural properties of graphs used to describe behavior models, e.g., DMQL and VM\*. Examples of graph analysis tasks used for process querying include the *problem of determining if a path exists in a graph* and *subgraph isomorphism problem*.

**Behavioral Analysis** Several existing process querying methods can support the analysis of properties of behaviors encoded in models, e.g., QuBPAL. These methods can be used to issue process queries that specify conditions over all the processes (of which there can be potentially infinitely many) encoded in the behaviors of models stored in a process repository.

Given a model of a finite-state system, e.g., a behavior model or an event log, and a formal property, *model checking* techniques verify whether the property holds for the system [2]. The formal properties are usually specified using formal logics.

**Temporal Logic** A temporal logic is a formal language for specifying and reasoning about propositions qualified in terms of time [20]. Several process querying methods are grounded in temporal logics, e.g., CRL. Given a process repository and a process query, these methods proceed by translating the query into a temporal logic expression, e.g., Linear Temporal Logic (LTL), Computation Tree Logic (CTL), or Metric Temporal Logic (MTL) expression. Then, each behavior model from the repository is translated into a finite-state system and verified against the expression. Those behavior models that translate to systems for which the property captured in the expression holds constitute the query result.

**First-Order Logic** First-Order Logic (FOL) is a formal logic that extends propositional logic, which operates over declarative propositions, with the use of predicates and quantifiers [36]. The QuBPAL language for process querying is an example language that operates by translating its queries to FOL and model checking them against the models in the repository. Hence, the process repository is interpreted as a formal FOL theory, while queries verify the formal properties of the theory.

Process querying can be grounded in techniques for *data querying*.

**Data Querying** Data querying is a technique for retrieving and augmenting data. Structured Query Language (SQL) is a language for managing data stored in a relational database [7]. DAPOQ-Lang is an example language for process querying grounded on SQL. It is an SQL wrapper that aims to simplify query formulation over a relational model for storing collections of event logs.



**SPARQL Protocol and RDF Query Language (SPARQL)** SPARQL is a semantic query language for retrieving and manipulating data captured in the Resource Description Framework (RDF) format [15]. Hence, SPARQL queries operate over data stored as a collection of “subject-predicate-object” triples. Several methods for process querying are based on SPARQL, e.g., BPMN VQL and BP-SPARQL. These methods encode process repositories as RDF data and implement process querying by first transforming process queries into SPARQL queries and then executing SPARQL queries over the RDF data.

**Graph Querying** A graph querying technique can be used for data querying in annotated graphs [32]. For instance, Descriptive PQL uses the Cypher graph query language [21] to implement process querying; Cypher is a declarative graph query language used in the Neo4J graph database. By relying on Cypher, Descriptive PQL queries can formulate intents to retrieve and augment underlying graph structures of the queried process models.

## 5 Process Querying Applications

Due to their generic purpose, namely retrieval, manipulation, and management of behavior models stored in process repositories, process querying methods have many applications in process mining and business process management. Some example applications of process querying are listed below. These applications were mentioned and exemplified by the authors of the process querying methods covered in this book. The list is not meant to be exhaustive but should provide an impression about the broad range of process querying applications.

**Business Process Compliance Management** Business process compliance management studies approaches to check and ensure that business processes satisfy relevant compliance requirements, for example, legal regulations and policies. Process queries are often used to specify conditions that lead to violations of compliance requirements, while the corresponding query results contain models that violate the requirements.

**Business Process Weakness Detection** A weakness of a business process is its part that hinders process execution or has a negative impact on process performance. Such weaknesses are often modeled using syntactically correct model fragments, but according to their semantic, they are undesirable or even harmful. An example of a weakness in a process is when a document is first printed and later scanned. A condition that represents a business process weakness can be formulated as a process query and then executed over a process repository to identify all occurrences of the weakness.

**Infrequent Behavior Detection** Real-world event data are full of noise and rare anomalous behavior. Using such raw data as input to process mining techniques is detrimental to the results and, hence, insights about real-world processes these

techniques deliver. Infrequent behavior detection studies algorithms for identifying noise and irregular event patterns in event data to be filtered before applying process mining. Some process querying methods allow specifying conditions and, thus, querying for infrequent process behavior.

**Process Discovery** Process discovery is a problem studied in process mining and consists of automating the task of process modeling. That is, given event data, e.g., an event log of an IT system, a process discovery algorithm automatically constructs a process model that describes the data. Process querying methods can be used to support process discovery, for example, by retrieving event data of interest and filtering out the data that discovery algorithms deem not important for fulfilling their tasks.

**Process Enhancement** Process enhancement can be seen as a problem that generalizes the problem of process discovery. Process enhancement aims to improve, for example, extend, correct, or annotate, an existing process model based on event data about its actual executions. Hence, process discovery can be seen as process enhancement when the original process model is empty. Similar to process discovery, various process queries can be used to support steps of process enhancement algorithms concerning event data selection.

**Process Instance Migration** Process instance migration is the task of adapting an incomplete execution of a process model to continue execution according to the rules of a different process model. This different process model can be a redesigned version of the original model that caters to new requirements. The migration instructions can be formalized as process queries and executed using process querying methods.

**Process Model Comparison** The problem of process model comparison deals with assessing how similar two given process models are. Note that the problem can be instantiated with different notions of similarity, including structural and behavioral criteria. Such criteria for assessing similarity can be specified as process queries. Then, models that get included in the results of most of such similarity queries can be accepted as most similar.

**Process Model Translation** Process model translation deals with translating process model labels from one natural language to another, for example, from English to Ukrainian, or vice versa. Solving this problem is useful when process models are used in multinational companies, as process models can be reused at different branches of the company around the globe. The operationalization of the relabeling of the process model concepts can be implemented through process querying.

**Process Monitoring** Process monitoring is the task of identifying problematic and successfully performing processes. The aggregated information about the performance of currently executing processes can often be implemented via process queries and then visualized via process performance dashboards.

**Process Reuse** Process reuse refers to the problem of constructing process models from the existing ones or their fragments. Hence, existing process models, fragments, and process patterns from other contexts are reused instead of creating them from scratch. Process querying can be used to discover reusable process pieces with desired characteristics for subsequent composition in new process models.

**Process Scheduling** As resources are in general scarce and usually follow certain availability patterns, for example, due to the work shifts or maintenance cycles, their availability needs to be scheduled. Process scheduling is concerned with determining which resources and when to utilize in order to maximize the performance of currently executing processes or process parts. Process querying can support process scheduling, for example, to inquire about the status of the resources, execute the scheduling logic, or assign resources to pending process tasks.

**Process Selection** Process selection refers to the practice of choosing how an organization carries out its operations, for instance, customer claim handling. Indeed, the exact process followed may differ for customers of various demographic groups. Such process selection rules, or business rules, can be encoded as process queries that use process case information and information on past process executions to implement the decision logic.

**Process Standardization** Standard process models are models that should be used as references; that is, they are exemplar models for describing best practices for certain classes of behaviors. Process standardization refers to the problem of replacing similar process models or similar model fragments with a single unified model or fragment. Process querying is useful at the early stages of process standardization, as queries can help identify similar fragments that should be standardized. Such similar models or model fragments can be included in the result of a corresponding carefully designed process query.

**Process Variance Management** Process variance management is the task of identifying, maintaining, and improving the handling of variants of the same process in an organization. This task can be supported by process querying at various stages. For example, process queries can specify conditions for identifying process variants, pinpointing their differences, and supporting their standardization.

**Syntactical Correctness Checking** Syntactical correctness checking is the task of identifying process models that violate the syntax rules of the modeling language used to capture them. Alternatively, syntactical correctness checking can be used to verify whether process models adhere to the modeling guidelines established by the organization. For example, the organization can establish that only a subset of the modeling constructs is allowed in process models. Rules for checking the syntactical correctness of process models can be captured as process queries.

## 6 Past, Present, and Future of Process Querying

Future work on process querying will aim to achieve *process querying compromise* [28], i.e., it will propose new and improve the existing process querying methods to support more practically relevant process queries that can be computed efficiently. As of today, the development of process querying methods still constitutes a non-coordinated effort. Many languages and systems for process querying were designed and developed in silos. Future work should contribute to the consolidation of various methods leading to the definition of a standard meta-model for behavior models and behaviors that these models describe, and a standard language for capturing process queries. Such consolidation may take place, for instance, around the Process Querying Framework [28], which defines typical components of a process querying solution as well as their interfaces and details roles of the constituent components. This book is the first step in this direction.

The existing process querying languages and methods cover a wide range of use cases and applications. A small tendency is observed toward the design of more methods that operate over specifications of behavior models. Besides, while there is a plethora of languages and techniques for capturing and executing instructions for filtering process repositories, i.e., selecting processes with specific characteristics, methods for manipulating process repositories, i.e., changing processes, are scarce and are still in their infancy. Future research will close these gaps by devising techniques that operate over the behaviors process models describe rather than their structures and developing methods that manipulate models to include or exclude certain behaviors they describe.

Many process querying techniques suffer from performance issues, if not for most basic queries, then for the intricate ones. To overcome such performance limitations, dedicated indexes can be designed and constructed for certain classes of process queries to allow trading the additional space for storing the indexes for the speedup in the processing of queries. This design of indexes should be guided by empirical investigations on which queries are considered most useful by the users. Indeed, those intricate and useful queries should inform the development of process querying indexes. Unfortunately, only several such empirical works exist, and this gap must be addressed in the future work.

## References

1. Acretoaie, V., Störrle, H., Strüber, D.: VMTL: A language for end-user model transformation. *Softw. Syst. Model.* **17**(4), 1139–1167 (2018). <https://doi.org/10.1007/s10270-016-0546-9>
2. Baier, C., Katoen, J.: *Principles of Model Checking*. MIT Press (2008)
3. Beheshti, S., Benatallah, B., Nezhad, H.R.M., Sakr, S.: A query language for analyzing business processes execution. In: *Business Process Management. Lecture Notes in Computer Science*, vol. 6896, pp. 281–297. Springer, Berlin, Heidelberg (2011). [https://doi.org/10.1007/978-3-642-23059-2\\_22](https://doi.org/10.1007/978-3-642-23059-2_22)

4. Beheshti, S., Benatallah, B., Nezhad, H.R.M.: Enabling the analysis of cross-cutting aspects in ad-hoc processes. In: *Advanced Information Systems Engineering. Lecture Notes in Computer Science*, vol. 7908, pp. 51–67. Springer (2013). [https://doi.org/10.1007/978-3-642-38709-8\\_4](https://doi.org/10.1007/978-3-642-38709-8_4)
5. Beheshti, S., Benatallah, B., Motahari-Nezhad, H.R.: Scalable graph-based OLAP analytics over process execution data. *Distrib. Parallel Databases* **34**(3), 379–423 (2016). <https://doi.org/10.1007/s10619-014-7171-9>
6. Beheshti, A., Benatallah, B., Motahari-Nezhad, H.R.: ProcessAtlas: A scalable and extensible platform for business process analytics. *Softw. Pract. Exp.* **48**(4), 842–866 (2018). <https://doi.org/10.1002/spe.2558>
7. Date, C., Darwen, H.: *A Guide to the SQL Standard: A User's Guide to the Standard Database Language SQL*. Addison-Wesley (1997)
8. de Murillas, E.G.L., Reijers, H.A., van der Aalst, W.M.P.: Everything you always wanted to know about your process, but did not know how to ask. In: *Business Process Management Workshops: Process Querying. Lecture Notes in Business Information Processing*, vol. 281, pp. 296–309 (2016). [https://doi.org/10.1007/978-3-319-58457-7\\_22](https://doi.org/10.1007/978-3-319-58457-7_22)
9. de Murillas, E.G.L., Reijers, H.A., van der Aalst, W.M.P.: Connecting databases with process mining: a meta model and toolset. *Softw. Syst. Model.* **18**(2), 1209–1247 (2018). <https://doi.org/10.1007/s10270-018-0664-7>
10. Delfmann, P., Breuker, D., Matzner, M., Becker, J.: Supporting information systems analysis through conceptual model query – the diagramed model query language (DMQL). *Commun. Assoc. Inf. Syst.* **37**, 24 (2015)
11. Dumas, M., Rosa, M.L., Mendling, J., Reijers, H.A.: *Fundamentals of Business Process Management*, 2nd edn. Springer (2018). <https://doi.org/10.1007/978-3-662-56509-4>
12. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Patterns in property specifications for finite-state verification. In: *Proceedings of the 21st International Conference on Software Engineering - ICSE '99*, pp. 411–420. ACM Press (1999). <https://doi.org/10.1145/302405.302672>
13. Elgammal, A., Turetken, O., Heuvel, W.J., Papazoglou, M.: Formalizing and applying compliance patterns for business process compliance. *Softw. Syst. Model.* **15**(1), 119–146 (2016). <https://doi.org/10.1007/s10270-014-0395-3>
14. Francescomarino, C.D., Tonella, P.: Crosscutting concern documentation by visual query of business processes. In: *Business Process Management Workshops. Lecture Notes in Business Information Processing*, vol. 17, pp. 18–31. Springer, Berlin, Heidelberg (2008). [https://doi.org/10.1007/978-3-642-00328-8\\_3](https://doi.org/10.1007/978-3-642-00328-8_3)
15. Hebel, J., Fisher, M., Blace, R., Perez-Lopez, A., Dean, M.: *Semantic Web Programming*, 1st edn. Wiley (2009)
16. Holzmann, G.J.: The model checker SPIN. *IEEE Trans. Softw. Eng.* **23**(5), 279–295 (1997). <https://doi.org/10.1109/32.588521>
17. Kammerer, K., Kolb, J., Reichert, M.: PQL – A descriptive language for querying, abstracting and changing process models. In: *Enterprise, Business-Process and Information Systems Modeling. Lecture Notes in Business Information Processing*, vol. 214, pp. 135–150. Springer (2015). [https://doi.org/10.1007/978-3-319-19237-6\\_9](https://doi.org/10.1007/978-3-319-19237-6_9)
18. Momotko, M.: *Tools for monitoring workflow processes to support dynamic workflow changes*. Ph.D. thesis, Polish Academy of Sciences (2005)
19. Momotko, M., Subieta, K.: Process query language: A way to make workflow processes more flexible. In: *Advances in Databases and Information Systems. Lecture Notes in Computer Science*, vol. 3255, pp. 306–321. Springer Berlin Heidelberg (2004). [https://doi.org/10.1007/978-3-540-30204-9\\_21](https://doi.org/10.1007/978-3-540-30204-9_21)
20. Ohrstrom, P., Hasle, P.F.V.: *Temporal Logic: From Ancient Ideas to Artificial Intelligence. Studies in Linguistics and Philosophy*. Springer Netherlands (2007)
21. Panzarino, O.: *Learning Cypher*. Packt Publishing (2014)
22. Polyvyanyy, A.: Business process querying. In: *Encyclopedia of Big Data Technologies*. Springer (2019). [https://doi.org/10.1007/978-3-319-63962-8\\_108-1](https://doi.org/10.1007/978-3-319-63962-8_108-1). [https://doi.org/10.1007/978-3-319-63962-8\\_108-1](https://doi.org/10.1007/978-3-319-63962-8_108-1)

23. Polyvyanyy, A., Vanhatalo, J., Völzer, H.: Simplified computation and generalization of the refined process structure tree. In: *Web Services and Formal Methods. Lecture Notes in Computer Science*, vol. 6551, pp. 25–41. Springer (2010). [https://doi.org/10.1007/978-3-642-19589-1\\_2](https://doi.org/10.1007/978-3-642-19589-1_2)
24. Polyvyanyy, A., Weidlich, M., Conforti, R., Rosa, M.L., ter Hofstede, A.H.M.: The 4C spectrum of fundamental behavioral relations for concurrent systems. In: *Application and Theory of Petri Nets and Concurrency. Lecture Notes in Computer Science*, vol. 8489, pp. 210–232. Springer International Publishing (2014). [https://doi.org/10.1007/978-3-319-07734-5\\_12](https://doi.org/10.1007/978-3-319-07734-5_12)
25. Pérez-Álvarez, J.M., López, M.T.G., Parody, L., Gasca, R.M.: Process instance query language to include process performance indicators in DMN. In: *IEEE Enterprise Distributed Object Computing Workshops*, pp. 1–8. IEEE Computer Society (2016). <https://doi.org/10.1109/EDOCW.2016.7584381>
26. Polyvyanyy, A., Corno, L., Conforti, R., Raboczi, S., Rosa, M.L., Fortino, G.: Process querying in Apromore. In: *Business Process Management Demo Session. CEUR Workshop Proceedings*, vol. 1418, pp. 105–109. CEUR-WS.org (2015)
27. Polyvyanyy, A., van der Aalst, W.M.P., ter Hofstede, A.H.M., Wynn, M.T.: Impact-driven process model repair. *ACM Trans. Softw. Eng. Methodol.* **25**(4), 1–60 (2017). <https://doi.org/10.1145/2980764>
28. Polyvyanyy, A., Ouyang, C., Barros, A., van der Aalst, W.M.P.: Process querying: Enabling business intelligence through query-based process analytics. *Decis. Support Syst.* **100**, 41–56 (2017). <https://doi.org/10.1016/j.dss.2017.04.011>
29. Polyvyanyy, A., ter Hofstede, A.H.M., Rosa, M.L., Ouyang, C., Pika, A.: Process query language: Design, implementation, and evaluation. *CoRR* **abs/1909.09543** (2019)
30. Polyvyanyy, A., Pika, A., ter Hofstede, A.H.M.: Scenario-based process querying for compliance, reuse, and standardization. *Inf. Syst.* **93**, 101,563 (2020)
31. Reinkemeyer, L.: *Process Mining in Action: Principles, Use Cases and Outlook*. Springer International Publishing (2020). <https://books.google.com.au/books?id=OrHWDwAAQBAJ>
32. Robinson, I., Webber, J., Eifrem, E.: *Graph Databases*. O’Reilly (2013)
33. Smith, F., Proietti, M.: Rule-based behavioral reasoning on semantic business processes. In: *International Conference on Agents and Artificial Intelligence*, pp. 130–143. SciTePress (2013)
34. Smith, F., Proietti, M.: Ontology-based representation and reasoning on process models: A logic programming approach. *CoRR* **abs/1410.1776** (2014)
35. Smith, F., Missikoff, M., Proietti, M.: Ontology-based querying of composite services. In: *Business System Management and Engineering. Lecture Notes in Computer Science*, vol. 7350, pp. 159–180. Springer, Berlin, Heidelberg (2010). [https://doi.org/10.1007/978-3-642-32439-0\\_10](https://doi.org/10.1007/978-3-642-32439-0_10)
36. Smullyan, R.M.: *First-Order Logic*. Courier Corporation (1995)
37. Störrle, H.: VMQL: A generic visual model query language. In: *IEEE Visual Languages and Human-Centric Computing*, pp. 199–206. IEEE Computer Society (2009). <https://doi.org/10.1109/VLHCC.2009.5295261>
38. Störrle, H.: VMQL: A visual language for ad-hoc model querying. *J. Vis. Lang. Comput.* **22**(1), 3–29 (2011). <https://doi.org/10.1016/j.jvlc.2010.11.004>
39. Subieta, K.: Stack-based query language. In: *Encyclopedia of Database Systems*, pp. 2771–2772. Springer US (2009). [https://doi.org/10.1007/978-0-387-39940-9\\_1115](https://doi.org/10.1007/978-0-387-39940-9_1115)
40. van der Aalst, W.M.P.: *Process Mining – Data Science in Action*, 2nd edn. Springer, Berlin, Heidelberg (2016). <https://doi.org/10.1007/978-3-662-49851-4>
41. Vogelgesang, T., Ambrosy, J., Becher, D., Seilbeck, R., Geyer-Klingenberg, J., Klenk, M.: *Celonis PQL: A Query Language for Process Mining*. Springer (2020). (In Press)
42. Weske, M.: *Business Process Management: Concepts, Languages, Architectures*, 3rd edn. Springer (2019). <https://doi.org/10.1007/978-3-662-59432-2>

# Index

## Symbols

*i\**, *see* *istar*

**MQ**, 167

BPMN VQL DOR operator, 193

BPMN VQL NEST operator, 193

BPMN VQL NOT operator, 193

BPMN VQL OR operator, 188

BPMN VQL PATH operator, 193

BPMN VQL stereotypes, 187

context, 153

4C behavioral predicate, 314, 331

    AlwaysOccurs, 315, 317

    CanOccur, 317

    Conflict, 317

    Cooccur, 315

    TotalCausal, 325

    TotalConcurrent, 317

4C spectrum, 322

## A

ABox, *see* Assertion Box

Action, 313, 314

Active component, 329

    Caching, 331

    Filtering, 331

    Indexing, 331

    Process querying, 332

Activity, 257, 379

    compound, 259

    diagram, 153, 156

    table, 381

    task, 259

Adonis, 152

Alignment, 328, 339

Annotation

    functional, 264

    semantic, 263

    terminological, 264

Assertion Box (ABox), 266

Atom, 257

Atomic pattern, 298

## B

Backus Naur Form (BNF), 234

Behavioral analysis, 518

Behavioral predicate, 270, 314, 322, 331

Behavior model, 5, 513

Benchmark, 70, 402

    TPC-H, 402

BI, *see* Business Intelligence

Big data, 3, 22, 45

BNF, *see* Backus Naur Form

Bottleneck, 71

BPAL, *see* Business Process Abstract  
    Language (BPAL)

BPD, *see* Business Process Diagram

BPKB, *see* Business Process Knowledge Base

BPM, *see* Business Process Management

BPMN, *see* Business Process Model and  
    Notation

BPMN Visual Query Language, 186

BPQL, *see* Business Process Query Language

BPS, *see* Business Process Schema

BP-SPARQL, 22, 25, 29, 31, 33–35

BRO, *see* Business Reference Ontology

Business domain ontology, 183

Business Intelligence (BI), 383, 402

- Business process, 22, 23, 27, 43, 46, 87, 88, 255, 377
    - execution, 87
    - graph, 413
    - instance, 85, 87, 88
    - management, 1, 46, 50, 139, 255, 480
    - model, 85, 88, 255, 461
    - modeling framework, 255
    - modeling language, 255
    - outsourcing, 425
    - repository, 303, 314
  - Business Process Abstract Language (BPAL), 256
    - framework, 256
    - platform, 275
    - reasoner, 274
  - Business Process Diagram (BPD), 183
    - populator, 198
  - Business Process Knowledge Base (BPKB), 183, 256
  - Business Process Management (BPM), 1, 46, 50, 139, 255, 480
    - lifecycle, 286
    - system, 87, 345
  - Business Process Model and Notation (BPMN), 256, 396, 438
    - diagram, 157
    - ontology, 183, 449
  - Business process outsourcing, 425
  - Business Process Query Language (BPQL), 354
    - architecture, 371
    - business process metamodel, 348
    - monitoring function, 368
  - Business Process Query Language semantics
    - algebraic operator, 362
    - context dependent function, 367
    - ENVS, 359
    - non-algebraic operator, 362
    - QRES, 360
  - Business Process Schema (BPS), 257
    - predicate, 270
  - Business Reference Ontology (BRO), 263
- C**
- Caching, 11, 70, 331
  - Case, 379, 381, 397
  - Case table, 381
  - Causal footprint, 420, 467
  - Causality, 314
  - Celonis PQL, 378, 380, 382–384, 386, 397, 402, 404
  - CEP, *see* Complex Event Processing
  - Complex Event Processing (CEP), 487
  - Compliance
    - checking, 71, 119
    - pattern, 297
    - repository, 303
    - verification, 482
  - Compliance Request Language (CRL), 287, 296, 298
    - framework, 287
    - meta-model, 298
  - Compliance rule
    - manager, 303
    - modeler, 305
  - Composite pattern, 301
  - Computational Tree Logic (CTL), 263
  - Conceptual model, 119
  - Concurrency, 314
  - Conflict, 314
  - Conformance checking, 71
  - Consistency, 266
  - Context, 153
  - Co-occurrence, 314
  - Correlation model, 6, 7
  - Covering problem, 332
  - Creation pair, 228
  - CRL, *see* Compliance Request Language
  - CTL, *see* Computational Tree Logic
  - Cypher Query Language, 232
- D**
- DAPOQ-Lang, 50
  - Dashboard, 87, 88, 92, 104
  - Data-driven process, 43
  - Dataflow, 94
  - Data lake, 45
  - Deadlock, 440
  - Data model, 51
  - Data querying, 518
  - Data state model, 244
  - Decision table, 90, 105
  - DELETE, 319, 331
  - Diagramed Model Query Language (DMQL), 128
    - 2.0, 136
    - notation, 132
    - performance, 138
    - runtime complexity, 138
    - syntax, 129
    - utility, 139
  - Dictionary encoding, 403
  - Distance measure, 422
  - DMN table, 92, 106



DMQL, *see* Diagramed Model Query Language  
 Domain Specific Language (DSL), 11, 69, 89, 152, 285, 314, 377  
 DSL, *see* Domain Specific Language  
 Dynamic system, 4

**E**

Edit distance, 414  
   graph-edit distance, 417  
   string-edit distance, 414  
 Effect, 265  
   negative, 266  
   positive, 266  
 Enabling condition, 265  
 Enactment, 260  
 Environment transparency, 152  
 Event, 4, 257, 379, 487, 512  
   end, 259  
   intermediate, 259  
   query, 489  
   start, 259  
   stream, 488  
   time-out, 259  
 Event–activity correlation, 492  
 Event-based process querying, 483  
 Event data, 71  
 Event log, 6, 379, 402  
 Event query, 489  
   discovery, 498  
   evaluation, 492  
   formalization, 491  
   generation, 495  
   operator, 489  
 Event stream, 488  
 Execution transparency, 152

**F**

Fact, 257  
 False positive, 451  
 Feature net (F-Net), 423  
 Filtering, 331  
 Find pattern, 161  
 First-Order logic, 518  
 Flow  
   element, 259  
   item, 259  
   sequence, 259  
 Fluent, 260  
   calculus, 260  
   expression, 261  
 F-measure, 463

F-Net, *see* Feature net  
 Forbid pattern, 161  
 FROM, 269  
 Function tree, 153

**G**

Gateway, 259, 438  
   exclusive, 259  
   inclusive, 259  
   parallel, 259  
 Generic Model Query Language  
   notation, 126  
   semantics, 126  
   syntax, 122  
 Generic Model Query Language (GMQL), 122  
 GMQL, *see* Generic Model Query Language  
 Graph database, 453  
 Graphical User Interface (GUI), 381, 383, 386, 397, 406  
 Graph querying, 519  
 Groovy, 69  
 GUI, *see* Graphical User Interface

**H**

Henshin, 176  
 Host language, 151

**I**

IBC, *see* Intelligent Business Cloud  
 Index, 421  
 Indexing, 331  
 Information content, 415  
 Infrequent behavior detection, 519  
 INSERT, 319, 331  
 Intelligent Business Cloud (IBC), 382, 404  
 istar, 443

**J**

jBPT library, 332  
 JIT, *see* Just-In-Time  
 Just-In-Time (JIT), 403  
   compilation, 403

**K**

Key Performance Indicator (KPI), 387, 392, 404  
 Knowledge intensive process, 43  
 Knowledge lake, 45  
 KPI, *see* Key Performance Indicator

**L**

Label ambiguity, 445  
 Lack of synchronization, 440  
 Latent semantic analysis, 466  
 Layout, 443  
 Lexer, 248  
 Linear Temporal Logic (LTL), 289, 294  
 Literal, 257  
 Log data, 378  
 Logic program, 257  
 Logic programming, 256  
 LTL, *see* Linear Temporal Logic

**M**

Matching relation
 

- exact matching, 427
- inexact matching, 427
- plug-in matching, 427
- process alignment, 430
- relation with similarity measures, 429
- requirement, 426

 Meta-edit, 152  
 Meta-model, 259  
 Metrical Temporal Logic (MTL), 289  
 Model
 

- collection, 469
- constraint, 151
- graph, 164
- query, 151
- repository, 153
- transformation, 151

 Model checker
 

- LoLA, 332
- SPIN, 305

 Model checking, 262, 339  
 Model vector, 420  
 Modeling language, 121  
 moq, 167  
 MTL, *see* Metrical Temporal Logic

**N**

NAC, *see* Negative Application Condition  
 Negative Application Condition (NAC), 161, 162

**O**

OLAP, *see* Online analytical processing  
 Online analytical processing (OLAP), 25, 29, 31  
 Ontology, 256, 448  
 Ontology Web Language, 256

ontology, 256  
 triple, 256  
 OpenSLEX, 51  
 Optimal alignment, 328, 339  
 OWL, *see* Web Ontology Language

**P**

PAC, *see* Positive Application Condition  
 PADAS, *see* Process Aware Data Suite  
 Parameter expression, 123  
 Parameter set, 231  
 Parser, 248  
 Participant, 259  
 Passive component, 329  
 Performance analysis, 71  
 Performance monitoring, 482  
 Petri net, 315, 332, 395  
 Petri Net Markup Language (PNML), 332  
 Ping-pong-case, 397, 399
 

- direct, 399
- indirect, 400

 PIQL, *see* Process Instance Query Language  
 PIQL engine, 102  
 PNML, *see* Petri Net Markup Language  
 Positive Application Condition (PAC), 161, 162  
 PQF, *see* Process Querying Framework  
 PQF part, *see* Process Querying Framework part  
 PQL, *see* Process Query Language  
 PQL bot, 332
 

- instance, 332

 PQL matching pattern, 234  
 PQL tool, 332, 333  
 Precedence, 263, 321  
 Precision, 463  
 Precision-at-k, 463  
 Precondition, 265  
 Process, 4, 5, 51, 313, 314, 512
 

- analysis, 3
- analytics, 3
- compliance, 116
- concurrent process, 317, 321
- deletion, 335
- discovery, 382, 384, 520
- enhancement, 382, 384, 520
- identifier, 269
- improvement, 448
- insertion, 335
- instance, 86, 87, 461
- instance migration, 520
- intelligence, 4
- landscape, 153

- manipulation, 313, 314, 319, 328
  - mining, 2, 6, 31, 49, 71, 329, 378, 469, 512
  - model, 6, 71, 224, 314, 379, 461
  - model comparison, 520
  - modeling, 3
  - model translation, 520
  - monitoring, 382, 384, 520
  - query, 8, 11, 513
  - querying, 4, 313, 314, 321, 335
  - repair, 314
  - repository, 8, 313, 314, 513
  - reuse, 521
  - scenario, 314
  - scheduling, 521
  - selection, 521
  - standardization, 521
  - trace, 461
  - update, 335
  - variance management, 521
  - view, 240, 428
  - weakness, 116, 139
  - Process Aware Data Suite (PADAS), 69
  - Process compliance management, 519
  - Process data analysis, 22, 25, 33
  - Process data analytics, 46
  - Process deletion problem, 335
  - Process graph, 22, 25, 34–36
    - modeling, 45
  - Process insertion problem, 335
  - Process Instance Query Language (PIQL), 87
    - engine, 89
  - Process mining, 2, 6, 31, 49, 329, 378, 469, 512
    - professional profiles, 71
  - Process model, 6, 71, 225, 314, 379, 461
    - abstraction, 227, 238
    - change, 226
    - comparison, 520
    - matching, 462
    - querying, 115
    - refactoring, 228, 447
    - selection, 234
    - similarity, 412, 462
    - translation, 520
  - Process modeling, 3
  - Process monitoring, 382, 384, 520
  - Process query, 8, 11, 513
  - Process querying, 4, 313, 314, 321, 332, 334
    - compromise, 15, 334, 522
    - decidability, 15
    - efficiency, 15
    - event-based, 483
    - instruction, 11, 331
    - method, 2, 8, 329, 511, 514
    - problem, 335
    - statistics, 11
    - suitability, 15
  - Process Querying Framework (PQF), 9, 175, 257, 329, 404
  - Process Querying Framework part (PQF part), 329
    - Execute, 12, 331
    - Interpret, 12, 332
    - Model, Simulate, Record, and Correlate, 9, 329
    - Prepare, 11, 331
  - Process Query Language (PQL), 220, 232, 313, 314, 321, 378
  - Process update problem, 335
  - Process weakness, 116, 139
    - automation, 139
    - detection, 116, 126, 519
    - environment, 139
    - information handling, 139
    - modeling error, 139
    - organization, 139
    - process flow, 139
    - technology switch, 139
  - Prolog, 256, 441
    - with tabling, 272
  - XSB, 256
  - Promela, 291
- ## Q
- QuBPAL, 256
  - Query, 116, 255
    - algorithm, 118
    - condition, 11
    - editor, 118
    - engine, 382
    - executor, 199
    - intent, 11
    - language, 118
    - occurrence, 122
    - result (*see* Query, occurrence)
    - translator, 199
  - Query intent, 11, 331
    - Create, 331
    - Read, 331
    - Update, 331
- ## R
- RDF, *see* Resource Description Framework
  - Reachability problem, 332
  - Recall, 463
  - Redo log, 53
  - Require pattern, 161

- Resource Description Framework (RDF), 448
- Resource pattern, 300
- Response, 263
- Role-Activity-Diagram, 152
- R-precision, 463
- Rule, 257, 259
  
- S**
- Segregation of Duties (SoD), 392, 397, 401, 402
- SELECT, 269, 315, 331
- Semantics
  - annotation predicate, 270
  - behavioral, 257
  - business process, 256
  - ontology-based, 257
  - procedural, 256
- Semantic Web, 256
- SESE, *see* Single-Entry-Single-Exit
- Similarity
  - activity-based, 417
  - attribute, 416
  - behavior-based, 419
  - edit distance, 415
  - measure, 414
  - metric, 463
  - property, 463
  - scale, 463
  - semantic, 415
  - structure-based, 417
  - syntactic, 414
  - word, 415
- Similarity-based search, 464
- Simulation model, 6
- Simulink, 152
- Single-Entry-Single-Exit (SESE), 226
- Snapshot, 403
- Snowflake schema, 380
- SoD, *see* Segregation of Duties
- Software repository, 314
- Soundness, 440
  - violation, 444, 450
- SPARQL, *see* SPARQL Protocol and RDF Query Language
- SPARQL Protocol and RDF Query Language (SPARQL), 256, 448, 519
- SQL, *see* Structured Query Language
- State, 4, 260
  - reachable, 262
  - successor, 262
- Stemming, 416
- Stop word removal, 416
- Structural analysis, 518
- Structured Query Language (SQL), 51, 313, 382, 383, 386
- Subgraph homeomorphism, 116
- Sub-pattern, 135
- Summarization, 22, 23, 25, 33, 34, 45
- Syntactical correctness, 443
- Syntactical correctness checking, 521
- Syntax transparency, 152
  
- T**
- Task instance, 89, 94
- TBox, *see* Terminological Box
- Temporal interval algebra, 58
  - Allen's interval algebra, 58
- Temporal logic, 263, 518
  - Computational Tree Logic, 263
  - formula, 263
  - Linear Temporal Logic, 289
  - Metrical Temporal Logic, 289
- Temporal property
  - duration, 57
  - period, 57
- Term, 257
- Terminological Box (TBox), 264
- Throughput time, 379, 389, 397
- Timed pattern, 302
- Timestamp, 379
- Trace, 5, 262, 512
  - correct, 262
  - with wildcards, 327
- Triple predicate, 270
  
- U**
- UML Activity diagram, 153, 156
- UML Use Case diagram, 153
- Unfolding, 332
- Untangling, 331
- UPDATE, 321, 331
- Update pattern, 161
- Use Case diagram, 153
  
- V**
- Variant, 379, 393
- Visual Model Manipulation Language, 150
- VM\*, 150
  - annotation, 162
  - pattern, 161

**W**

Weakness category, *see* Process weakness

Web Ontology Language (OWL), [448](#)

2.0, [448](#)

WHERE, [269](#)

Workflow, [255](#)

Working capital optimization, [397](#)

**X**

XES, [51](#)