A COMPLIANCE MANAGEMENT FRAMEWORK FOR BUSINESS PROCESS MODELS

AHMED MAHMOUD HANY ALY AWAD

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

DISSERTATION
ZUR ERLANGUNG DES AKADEMISCHEN GRADES
"DOCTOR RERUM NATURALIUM"
(DR. RER. NAT.)

— DOCTORAL THESIS —

MAY, 2010

Published online at the
Institutional Repository of the University of Potsdam:
URL http://opus.kobv.de/ubp/volltexte/2010/4922/
URN urn:nbn:de:kobv:517-opus-49222
http://nbn-resolving.org/urn:nbn:de:kobv:517-opus-49222

To Radwa my wife and Nour my daughter the innocent soul

Zusammenfassung

Firmen entwickeln Prozessmodelle um ihre Geschäftstätigkeit explizit zu beschreiben. Geschäftsprozesse müssen verschiedene Arten von Compliance-Anforderungen einhalten. Solche Compliance-Anforderungen entstammen einer Vielzahl von Quellen, z.B. Verordnung wie dem Sarbanes Oxley Act von 2002, interne Richtlinien und Best Practices. Die Nichteinhaltung von Compliance-Anforderungen kann zu gesetztlichen Strafen oder dem Verlust von Wettbewerbsvorteilen und somit dem Verlust von Marktanteilen führen.

Im Gegensatz zu dem klassischen, domänen-unabhängigen Begriff der Korrektheit von Geschäftsprozessen, sind Compliance-Anforderungen domain-spezifisch und ändern sich im Laufe der Zeit. Neue Anforderungen resultieren aus neuen Gesetzen und der Einführung neuer Unternehmensrichtlinien. Aufgrund der Vielzahl der Quellen für Compliance-Anforderungen, können sie unterschiedliche Ziele verfolgen und somit widersprüchliche Aussagen treffen. Schliesslich betreffen Compliance-Anforderungen verschiedene Aspekte von Geschäftsprozessen, wie Kontrollfluss- und Datenabhängigkeiten. Auf Grund dessen können Compliance-Prüfungen nicht direkt Hard-coded werden. Vielmehr ist ein Prozess der wiederholten Modellierung von Compliance-Regeln und ihrer anschliessenden automatischen Prüfung gegen die Geschäftsprozesse nötig.

Diese Dissertation stellt einen formalen Ansatz zur Überprüfung der Einhaltung von Compliance-Regeln während der Spezifikation von Geschäftsprozessen vor. Mit visuellen Mustern ist es möglich, Compliance-Regeln hinsichtlich Kontrollfluss- und Datenabhängigkeiten sowie bedingte Regeln zu spezifizieren. Jedes Muster wird in eine Formel der temporalen Logik abgebildet. Die Dissertation behandelt das Problem der Konsistenzprüfung zwischen verschiedenen Compliance-Anforderungen, wie sie sich aus unterschiedlichen Quellen ergeben können. Ebenfalls zeigt diese Dissertation, wie Compliance-Regeln gegen die Geschäftsprozesse automatisch mittels Model Checking geprüft werden. Es wird aufgezeigt, dass zusätzliche Domänen-Kenntnisse notwendig sind, um richtige Entscheidungen zu treffen.

Der vorgestelle Ansatz ermöglicht nützliches Feedback für Modellierer im Fall eines Compliance-Verstosses. Das Feedback wird in Form von Teilen des Prozessmodells gegeben, deren Ausführung die Verletzung verursacht. In einigen Fällen ist der vorgestelle Ansatz in der Lage, den Compliance-Verstoss automatisch zu beheben.

Abstract

Companies develop process models to explicitly describe their business operations. In the same time, business operations, business processes, must adhere to various types of compliance requirements. Regulations, e.g., Sarbanes Oxley Act of 2002, internal policies, best practices are just a few sources of compliance requirements. In some cases, non-adherence to compliance requirements makes the organization subject to legal punishment. In other cases, non-adherence to compliance leads to loss of competitive advantage and thus loss of market share.

Unlike the classical domain-independent behavioral correctness of business processes, compliance requirements are domain-specific. Moreover, compliance requirements change over time. New requirements might appear due to change in laws and adoption of new policies. Compliance requirements are offered or enforced by different entities that have different objectives behind these requirements. Finally, compliance requirements might affect different aspects of business processes, e.g., control flow and data flow. As a result, it is infeasible to hard-code compliance checks in tools. Rather, a repeatable process of modeling compliance rules and checking them against business processes automatically is needed.

This thesis provides a formal approach to support process design-time compliance checking. Using visual patterns, it is possible to model compliance requirements concerning control flow, data flow and conditional flow rules. Each pattern is mapped into a temporal logic formula. The thesis addresses the problem of consistency checking among various compliance requirements, as they might stem from divergent sources. Also, the thesis contributes to automatically check compliance requirements against process models using model checking. We show that extra domain knowledge, other than expressed in compliance rules, is needed to reach correct decisions.

In case of violations, we are able to provide a useful feedback to the user. The feedback is in the form of parts of the process model whose execution causes the violation. In some cases, our approach is capable of providing automated remedy of the violation.

Contents

Abstract **iii**

Contents **v**

1 Introduction **3**
1.1 Problem Statement 5
1.2 Scope of the Thesis 6
1.3 Organization of the Thesis 8
1.4 Publications in the Course of this Thesis 9

2 Requirements for Compliance Management **11**
2.1 Categories of Compliance Requirements 11
2.2 Requirements 13
2.3 Contribution of the Thesis 17

3 Related Work **23**
3.1 Compliance Management Frameworks 23
3.2 Design Time Compliance Checking 28
3.3 Runtime Compliance Monitoring 36
3.4 Compliance Auditing 37
3.5 Other Approaches 38

4 Foundations **41**
4.1 Business Process Modeling 41
4.2 BPM-Q 52
4.3 BPMN-Q 57
4.4 The Business Knowledge 59
4.5 Model Checking 60

5 Modeling and Checking Compliance Rules **67**
5.1 Categories of Compliance Rules 68

5.2 Modeling Compliance Rules 69
5.3 Consistency Checking Among Rules 89
5.4 Checking Rules Against Processes 98
5.5 Example 102
5.6 Summary & Outlook 110

6 Explaining Compliance Rules Violations 113
6.1 Deriving Anti Patterns 114
6.2 Control Flow Rules Violations 114
6.3 Data Flow Rules Violations 120
6.4 Conditional Control Flow Rules Violations 122
6.5 Evaluation of Temporal Logic Queries 130
6.6 Matching Anti Pattern Queries to Process Models With Multiple Activity Occurrences 131
6.7 Back to Example 134
6.8 Summary & Outlook 140

7 Resolution of Compliance Violation 143
7.1 Another Tool Set 144
7.2 Catalog of Violations 147
7.3 Resolving Violations 152
7.4 Directions to Resolve Violations to Other Patterns 165
7.5 Discussion 166

8 Implementation 169
8.1 Oryx Architecture and Extensions 170
8.2 Integrating BPMN-Q and Enabling Compliance Checking 171
8.3 Example 173

9 Discussion 179

Bibliography 185

Acknowledgments

This thesis would not have been written and the research behind would not have been done without the support of many people. Mathias Weske who accepted me in his research group and guided my research. My professors at Information Systems department, Cairo University, who granted me the scholarship to work for a PhD degree. Dr. Tag eldin and all fellows in the Egyptian cultural office who have been very supportive and helpful. My colleagues at the Business Process Technology group, especially, Hagen Overdick who was very helpful to me in my first day in Germany, Katrin Heinrich whose support to me started even before I arrive at Germany, Gero Decker for his helpful and insightful discussions, Matthias Weidlich for long discussions about formal aspects of process behavior, Artem Polyvyanyy and Sergey Smirnov for their feedback and support regarding process decomposition, Mohammed AbuJarour from the Information Systems group who gave me useful as an external to the field.

I would like to admit that without the support of my wife, Radwa, for giving me love and care, my parents and sister, it would not be possible to accomplish that work.

Chapter 1

Introduction

With the start of the new millennium, a number of financial scandals in many places of the world, ending with the financial crisis from the year 2008, have drawn attention to the severe impact of lack of control over business. Fraudulent business transactions, lack of trusted reporting mechanism of companies and uncontrolled money transfer are just a few examples for what lack of control can lead to. As a reaction, several regulations, e.g., SOX [99], and financial guidelines, e.g., BASELII [101], were established to force organizations to have internal controls over their business and be able to show that to authorities. The objective behind these regulations and guide lines is of course to avoid such scandals and to safeguard the economic system.

For instance, the Sarbanes-Oxley Act of 2002 [99] is a United States federal law. It was enacted in 2002 as a reaction to a number of major corporate and accounting scandals including those affecting Enron, Tyco International, Adelphia, Peregrine Systems and WorldCom. These scandals, which cost investors billions of dollars when the share prices of affected companies collapsed, shook public confidence in the nation's securities markets. The Act consists of several titles and sections which regulate the structure of financial disclosures. Section 302 of the Act obliges companies to establish and maintain internal controls, to evaluate the effectiveness of internal controls and to report on their conclusions about the effectiveness of their internal controls.

Compliance checking and enforcement is the act of establishing internal controls with which adherence to regulations is guaranteed. Compliance management is the ongoing process of identifying relevant regulations to the organization; assessing the risk of not obeying the identified compliance requirements; establishing effective internal controls to prevent/avoid/detect violations to compliance; maintain the effectiveness of these controls. A compliance officer is a new role created within the organization structure to be in charge of managing and following up the compliance status assessment of the organization.

Compliance requirements might not just stem from regulations. Rather, an organization might want to establish controls for its own internal policies and to benefit from best

practices in the business domain. Moreover, compliance is a domain-specific problem where requirements vary from one domain to another. For instance, Sarbanes-Oxley act [99] is concerned with regulating companies, Anti Money Laundering [25] guidelines are relevant to financial institutions. This calls for a repeatable compliance checking and enforcement processes within the organization.

Internal controls are meant to limit the way organizations are allowed to act. The restrictions can affect different aspects. For instance, in financial institutions like banks, a control objective could be to report large amount deposits coming from foreign banks. This control objective could be established to mitigate the risk of financing criminal acts, e.g. terroristic attacks. Other compliance requirements could restrict the usage of clients personal data within an organization. Table 1.1 lists a few examples of compliance requirements.

Domain	Risk	Internal Control
Banking	False identity	Identity of new customers have to be checked before opening a new bank account
Banking	Untrusted customers	If the customer's certificate is false, a bank account must never be opened
Procurement	Fraudulent purchases	A purchase request is approved by a person other than the requester
Clinic	Unwanted side effects	For patients older than 60 years an additional tolerance test is required before surgery

Table 1.1: Sample compliance requirements

Achieving compliance requires massive effort and costs a lot of money [57]. Currently, compliance is achieved by hiring expensive auditors who typically use a heuristic approach to select and investigate audit trails to show evidence about compliance. An audit trail could be any evidence on a business activity, e.g., bills, bank statement, or logs of information systems. In addition to the impact on the organization's budget, compliance checking with this approach incurs a large overhead in terms of time consumed to check for compliance. Moreover, the check is always of a detective nature. That is, an auditor can detect violations. In this case, organizations might be subject to penalties due to non-compliance or due to being late to declare compliance.

The external auditing is necessary as a proof of organization compliance for authorities and regulatory bodies. However, external auditing does not help the organization have a self-assessment and continuously repeatable evaluation of its compliance status. Thus, it is necessary for the organization to introduce approaches that help assess the compliance status internally.

Process models are developed as a means to document the operational activities of an organization. With the maturity of workflow systems, process models are the first steps to automate the day-to-day business operations. With the guide of these models, organizations can realize how business objectives are achieved, align their IT infrastructure, estimate resources, people, time and material, required to accomplish processes. Thus, a considerable amount of research has been devoted to check several correctness criteria of process models.

With the explicit view business process models provide over business activities, business process models are good candidates to check the effectiveness of established internal controls [65]. This means, business processes are confronted with new correctness criteria, imposed by compliance requirements. Unlike classical correctness criteria, e.g., the different notions of soundness [3, 30, 111, 87] and correct data flow[120, 134], compliance requirements are considered as *semantic* constraints over the way processes behave. Compliance requirements vary from one domain to the other. Also, these requirements can change due to new laws, policies or other types of constraints. As a response to the nature of change, new approaches and tools have to be developed where compliance rules, are no longer hard coded. Moreover, compliance management is seen as a lifetime process. That is, at different phases of a business process life cycle, compliance follow up takes different forms. This increases the complexity of the compliance process and in the mean time makes the need for automation stronger.

In addition to business experts and practitioners, we assume the role of a compliance officer. A compliance officer is responsible for reviewing laws and legislative documents, may be with the help of lawyers, to extract compliance requirements. The compliance officer establishes internal controls that translate compliance requirements into the organization's terms; discuss these controls with other people like stakeholders and business experts to agree upon internal controls' relevance and importance.

1.1 Problem Statement

This thesis introduces a compliance management framework and an approach to automate the checking of compliance requirements against process models. We provide users, compliance officers, with means to express internal controls (compliance rules) and check them against business process models.

On the one hand, the automation of compliance checking calls for formal approaches where compliance requirements and process models are represented in a mathematical form. On the other hand, compliance officers need to represent these internal controls in a way comprehensible by stakeholders and business experts who not necessarily have a sufficient mathematical background to understand mathematical formulas. Thus, one question to be addressed by this thesis is how to find a balance between formalization and readability of compliance requirements?

Usually, organizations maintain their business process models in repositories. A

compliance officer not necessarily knows before-hand which process models are subject to check against which compliance rules. Another question to be addressed is how to help the compliance officer correlate internal controls to business processes.

Building on the fact that a compliance officer lacks a priori knowledge about investigated processes, upon finding violations to compliance requirements, we are concerned with the question of how to provide the user with helpful feedback about the violation? Violation explanation is required for many reasons. First, localizing the problematic part of a process helps focus the discussion between compliance officers and business experts. Second, in large business processes, it is tedious and time consuming to manually track problematic parts of the process. Third, this helps develop semi automated approaches to resolve the violation. Resolving violations is also a question of interest to the thesis. That is, how is it possible to provide automated resolutions of violations? What knowledge needs to be explicitly collected in order to achieve automated resolution of violations?

Compliance requirements might affect more than one business process. In the mean time, a business process might be subject to checking against several compliance requirements. This calls for the separate maintenance of compliance requirements as independent artifacts from process models. This thesis raises the need to maintain compliance repositories as well as the need to maintain business process model repositories.

As compliance requirements might stem from different sources and, yet, are subject to check against a common set of business processes, it is likely to have inconsistencies among these compliance requirements. Inconsistency can be in the form of redundancy or conflicts. This calls for not only correlating compliance requirements and business processes, but also correlating compliance requirements with other compliance requirements. This thesis addresses the problem of deciding about inconsistency among related compliance requirements as prerequisite to check compliance against business processes.

1.2 Scope of the Thesis

This section sets the scope of the contributions in this thesis. We discuss this scope from two points of view. The first viewpoint is concerned with the phases to establish compliance requirements. The second viewpoint is concerned with the lifecycle of the business processes.

The different phases for compliance requirements establishment span the whole spectrum between the discovery of relevant regulations, standards, etc., to the point of assessment of compliance to compliance requirements.

Figure 1.1 depicts the phases of compliance requirements establishment [119]. The first phase is to discover relevant regulations,e.g. SOX [99], standards, e.g., ISO family, contracts, or any other source of information that might require ensuring controls over the business. Usually these requirements are stated informally in natural language. Thus, an effort has to be done to extract a specific set of compliance requirements to formalize it. Priorities of compliance requirements vary. Thus, risk assessment has to be conducted

Figure 1.1: Phases of compliance requirements establishment

in order to evaluate the threat of not obeying these requirements. Thereafter, internal controls are designed in order to realize compliance requirements. As internal controls stem from different requirements, it is crucial to assess the consistency among these requirements. At this point, the organization has to assess its compliance status with these internal controls. The whole sequence of phases is repeated each time the organization is obliged to adhere to new compliance requirements. Nevertheless, subsequences of these phases could be executed as well. For instance, due to entering a new market, risk has to be reassessed and this causes redesign of compliance rules and so forth.

Steps 1-3 are human-centric activities by nature[119]. Human experts, e.g., compliance officers, have to identify what is relevant to the organization from the set of laws and regulations. They have to assess the risk of not obeying these requirements and have to develop internal controls that detect and/or prevent violations to such requirements. These steps are out of scope of the thesis. Nevertheless, we discuss how our approach helps users track and link compliance rules to their sources.

The objective of this thesis is to give automated support for the steps 4-6. Modeling internal controls (compliance rules), checking their consistency and verifying them against process models constitute the scope of the thesis. The assessment of compliance rule, step 6, can be applied to different artifacts within the organization. Business processes,

IT-systems or audit trails can all be subject to compliance assessment. This is the step where compliance requirements can be linked to business processes.

The business process lifecycle is the other scope on the work in this thesis. Figure 1.2 shows the different stages of business process lifecycle. Within the design and analysis phase, new process model are identified and created or existing ones are adapted to cope with the new situation. Afterwards, the process model is configured and deployed. Then, a process is enacted on a specific execution platform. The execution might be monitored. Afterwards, executed processes are evaluated against several metrics, e.g., performance.

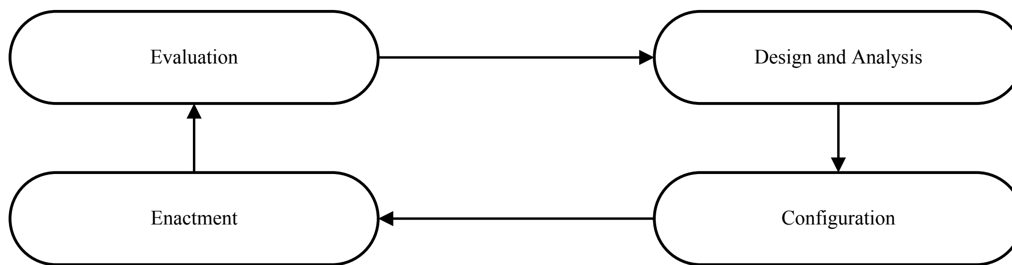


Figure 1.2: Business process lifecycle

In this regard, the thesis scope is limited to the design and analysis phase of the business process lifecycle. The support for checking compliance at design time would give a bottom line guarantee of compliance, if the organization is committed to do business in the way stated in process models. Moreover, we are addressing compliance requirements related to control and data flow aspects of business processes, e.g., activity execution ordering [159]. We believe that there has been work addressing other aspects of compliance as will be pointed out in Chapter 3.

1.3 Organization of the Thesis

The rest of the thesis is organized as follows. Chapter 2 discusses compliance aspects that can be addressed on the business process model level. Moreover, it sets the requirements framework that guided the contributions of the thesis. Finally, it gives an overview of the approach contributed in this thesis.

Related work is discussed in Chapter 3 where other approaches to compliance management of business processes are evaluated against the requirements we established.

Chapter 4 describes the foundations of the work done in the thesis. It describes concepts and techniques existing in literature and techniques developed by the author of the thesis.

Chapters 5-7 describe the contribution of the thesis. The contents of these chapters will be briefly described in Section 2.3 after the description of our requirements framework, so that the contribution is clear to the reader.

A prototype was implemented as a proof of the contributions of this thesis. Chapter 8 elaborates on details of this implementation.

Finally, a critical discussion of our approach is given in Chapter 9 along with open issues for future work.

1.4 Publications in the Course of this Thesis

The following is the list of publications achieved in the course of accomplishing this thesis:

- Ahmed Awad, Matthias Weidlich, and Mathias Weske. Consistency Checking of Compliance Rules. In Business Information Systems, volume 47 of Lecture Notes in Business Information Processing, pages 106-118. Springer, 2010.

- Ahmed Awad, Sergey Smirnov, and Mathias Weske. Resolution of Compliance Violation in Business Process Models: A Planning-based Approach. In Robert Meersman, Tharam S. Dillon, and Pilar Herrero, editors, OTM Conferences (1), volume 5870 of Lecture Notes in Computer Science, pages 6-23. Springer, 2009.

- Ahmed Awad, Matthias Weidlich, and Mathias Weske. Specification, Verification and Explanation of Violation for Data Aware Compliance Rules. In Luciano Baresi, Chi-Hung Chi, and Jun Suzuki, editors, International Conference on Service Oriented Computing/ServiceWave, volume 5900 of Lecture Notes in Computer Science, pages 500-515, 2009.

- Ahmed Awad, Gero Decker, and Niels Lohmann. Diagnosing and Repairing Data Anomalies in Process Models. In Stefanie Rinderle-Ma, Shazia Sadiq, and Frank Leymann, editors, Business Process Management Workshops, volume 43 of Lecture Notes in Business Information Processing, pages 5-16. Springer-Verlag, September 2009.

- Ahmed Awad and Mathias Weske. Visualization of Compliance Violation in Business Process Models. In Stefanie Rinderle-Ma, Shazia Sadiq, and Frank Leymann, editors, Business Process Management Workshops, volume 43 of Lecture Notes in Business Information Processing, pages 182-193. Springer, 2009.

- Ahmed Awad, Sergey Smirnov, and Mathias Weske. Towards Resolving Compliance Violations in Business Process Models. In Shazia Sadiq, Marta Indulska, and Michael zur Muehlen, editors, The Impact of Governance Risk and Compliance on Information Systems (GRCIS), volume 459, pages 18-32. CEUR-WS.org, 2009.

- Ahmed Awad, Artem Polyvyanyy, and Mathias Weske. Semantic Querying of Business Process Models. In Enterprise Distributed Object Computing Conference (EDOC), pages 85-94. IEEE Computer Society, 2008.

- Ahmed Awad, Gero Decker, and Mathias Weske. Efficient Compliance Checking Using BPMN-Q and Temporal Logic. In Marlon Dumas, Manfred Reichert, and Ming-Chien Shan, editors, Business Process Management (BPM), volume 5240 of Lecture Notes in Computer Science, pages 326-341. Springer, 2008.
- Ahmed Awad. BPMN-Q: A Language to Query Business Processes. In Manfred Reichert, Stefan Strecker, and Klaus Turowski, editors, Enterprise Modeling and Information Systems Architecture (EMISA), volume P-119 of LNI, pages 115-128. GI, 2007.

Chapter 2

Requirements for Compliance Management

The need to ensure compliance is an ongoing effort where several roles within the organization are involved. Its ongoing nature calls for automated approaches that allow effective assessment of the compliance status of the organization. The involvement of people with different backgrounds requires that internal controls are represented in different forms in order to allow people to negotiate about them.

Since compliance management is an enterprise wide problem, it is hard to cover all its aspects in a single study. Thus, as was stated earlier, we limit ourselves to compliance management on the level of business processes. Moreover, we are concerned with design-time checking of compliance aspects, especially control and data flow aspects.

In Section 2.1 we discuss the different aspects of compliance management on business processes. The section gives an overview on the different aspects, not necessarily checked only at design time. Requirements for a design time compliance management for business processes are discussed in Section 2.2.

Based on the requirements discussed in Section 2.2, an overview of the contribution of the thesis is given in Section 2.3. We provide the compliance management framework. Also, we justify our choices of tools and techniques to realize the different components of the framework.

2.1 Categories of Compliance Requirements

Compliance requirements affect the aspects that can be modeled in a business process, control flow, data flow, resources, and timing [121, 159].

2.1.1 Control Flow

Compliance rules might require that certain activities must be executed in the course of a business process. For instance, an anti money laundering requires "reporting a large deposit transaction by a bank" [25]. In some other rules, dependency between activities must exist. For instance, in an "open account" process, "customer identity checking step must precede the step of opening an account". We believe that this type of compliance rules must be checked and enforced at process design time. With a process model correctly designed to satisfy these compliance requirements and assuming a faithful execution of these processes, it is not possible to violate compliance at run time, unless for exceptional situations.

2.1.2 Data Flow

One aspect of compliance is to ensure that business objects are manipulated by process models as expected. Also, data and control flow aspects can be mixed to define so-called conditional flow rules. An example of conditional rule is "If the customer identity check fails; the customer must be added to a black list". We call this type of compliance rules "conditional" because the dependency between activities depends on conditions. Conditional flow rules are, of course, more expressive than plain control flow rules. That is, conditional rules are able to catch finer compliance rules. However, the level of details about data in process models and compliance rules might vary. Thus, it is necessary to guarantee a uniform level of data abstraction at checking time between business processes and compliance rules. This type of rules is also addressed by our framework as they can be efficiently checked at design time and compliance at process run time can be guaranteed if the execution respects the process model.

2.1.3 Human Resources

Security policies regarding "who will execute what" are seen as compliance requirements. Mainly, the separation of duty principle is of major importance to prevent fraud. That is, the initiator of a request cannot be the same person to approve that request. We believe that sufficient work has been done to verify such human resource constraints at process design time [153, 152]. However, we argue that correctly modeling these compliance requirements is not enough. This is due to the human factor in this situation where run time monitoring is needed to ensure compliance.

2.1.4 Timing

Service level agreement constraints fall under the fourth category. Also, it might be required that certain activities must be completed within k units of time. Moreover, some compliance requirements might stress that customer records must be kept for at least m

years. Typically, these rules are monitored at run time. Thus, they are out of scope of our work.

Collectively, we can notice two differences between compliance rules under the control and data flow categories on one hand and the resource and time categories on the other hand. Resource and time constraints are of local nature. That is, checking and enforcement is achieved at process design time by explicitly modeling the compliance control. For instance, explicitly modeling a separation of duty between two tasks or explicitly adding a timing constraint on a specific activity. On the other hand, control and data rules most of the time, as will be shown later, are concerned with dependencies which requires the study of the process behavior in order to determine compliance. Thus, in these situations, automated checking is needed to guarantee compliance.

2.2 Requirements

In this section we discuss the set of requirements that guided the contributions of the thesis. To come up with these requirements, we checked several compliance requirements documents, e.g., the guideline for Anti Money Laundering [25]. Also, we integrated requirements from other proposals for compliance management frameworks [81, 68]. The requirements take into consideration the two worlds of compliance rules and business processes that have to be linked together. Also, it is oriented to providing the highest degree of automation for compliance support. The requirements aim at an intelligent systems that extends compliance support beyond a yes/no answer about compliance status of business processes to the support of explaining violations and trying to automatically resolve them.

2.2.1 Req. 1: Formal Specification of Compliance Rules

In order to allow automated reasoning about the compliance status of business processes formal approaches have to be adopted. Checking approaches should be selected/developed in the way that provides the highest degree of automation. Moreover, analysis algorithms must be efficient. Also, the chosen formalism must be capable of supporting the level of abstraction needed by compliance rules. Usually, compliance rules address a specific situation, e.g., dependency between activities, and abstract from other details. On the contrary, process models are detailed, as they need to be operational, and are more generic than compliance rules. For instance, a compliance rule might be concerned with the situation of handling fraudulent insurance claims, while an insurance claim handling process has to deal also with sound claims.

The checking approach must be transparent to the user. That is, the mapping of business process models and compliance requirements into a specific formalism must be achieved automatically. This allows a wider set of users to participate in the compliance rules modeling and checking.

2.2.2 Req. 2: Compliance Rules are First Class Citizens

Compliance rules must be explicitly maintained in separate constraints repositories. Although rules are to be checked against processes, they must not be implicitly attached to and hidden within process definitions. The separate maintenance of rules is needed for several reasons.

Firstly, a compliance rule is subject to checking against several process models. So, if compliance rules are replicated as attachments to each related process model, there is a chance for inconsistency if compliance rules are updated only in a subset of the related process models. Thus, compliance rules must be maintained separately and be linked to process models. Another necessity for separately maintaining compliance rules is the need to attach arbitrary metadata to them. For instance, metadata about the source legislation, the date from which it is effective, comments from compliance officers and other stakeholders can be attached to compliance rules. Moreover, Compliance rules might not be active all the time. For instance, compliance rules might be deactivated. An example would be disabling some internal policies to provide flexibility. All these situations call for keeping compliance rules separated from process definitions.

Separately storing compliance rules also allows support for sophisticated maintenance approaches like versioning that allows to trace the progress of compliance rules and also allows to foresee any future enhancement. Moreover, it is possible to group related rules to allow further analysis on them, e.g., consistency checking as will be discussed in $Req.$ 3. Also, it is possible to come with complex rules out of simple rules in a dynamic way. For instance if we have three basic compliance rules $r1$,$r2$ and $r3$, we can come up with a complex rules $cr1 = (r1 \wedge r2) \vee r3$. In this case, any changes to the basic rules will be propagated to $cr1$ automatically, thus, avoiding problems of outdated rule definitions.

2.2.3 Req. 3: Consistency Checking of Compliance Rules

With the possible divergent sources of compliance requirements, it is likely to have inconsistencies among a set of compliance rules. Also, the human interpretation of juridic documents might increase the chance of inconsistencies. Thus, it is necessary to establish mechanisms to decide about inconsistencies among compliance rules. Inconsistency can be attributed to conflict, redundancy or both. Conflicts among a set of rules indicates that it is not possible at all to come up with a process model that realizes all of them. On the other hand, redundancy is about having two or more rules describing the same situation.

Without the ability to decide about consistency of a set of related rules, a precious time and effort could be spent trying to check/enforce conflicting or redundant compliance requirements. It might take several cycles of updating process models before figuring out that the compliance requirements are conflicting, especially when checks of conflicting rules are done by different people. This necessitates resolving the conflict among compliance rules before proceeding to checking them on process models.

Also, it is necessary to establish resolution approaches for conflicts among compliance

rules. However, not in every case the resolution is automated. One possibility of resolution is to identify priorities between conflicting rules.

2.2.4 Req. 4: Correlating Processes to Compliance Rules

(Semi) automatic identification of process models subject to checking in a repository of business process models is a must. Within large business process repositories, it is tedious and error prone to manually scan process models to decide about their relevance to certain compliance rules. Providing tools and techniques that help systematically access a process repository and query for processes based on a specific criteria is considered as a valuable aid in the process of compliance management to establish correlation between processes and rules.

This correlation allows the automated checking of rules against processes. Yet, in a loosely coupled fashion. That is, whenever a rule is changed the compliance officer is informed about process models needed to be checked. Also, it is guaranteed that all processes subject to investigation are checked against the same version of the compliance rule. Similarly, a change of the process model determines which rules need to be (re)checked.

2.2.5 Req. 5: Providing Information About Compliance

$Req.\ 1$ stated that compliance rules must be expressed in a formal language to automatically check whether they are satisfied by a business process. In the current requirement, we are concerned with a more advanced situation. That is, if a process model p is compliant with a rule r, the compliance approach must be able to decide about the nature of the compliance.

Usually, compliance rules are on the form $s \rightarrow q$. Where s is the condition of the rule and q is the consequent. When a rule r is correlated to a process model p, it is required that p will behave in a way that makes s true and thus q must also be true, in order to be compliant. However, if the process p never exhibits s then p is also compliant. In this case p *vacuously* satisfies r. Thus, it is important that the compliance checking approach to inform the user, e.g. compliance officer, if a rule r is vacuously satisfied by a process p as this might help identify logical deficiencies in the design of p. The resolution of these deficiencies is out of scope.

2.2.6 Req. 6: Providing Useful Feedback in Case of Violation

The localization of problematic parts of the process models helps business experts focus their discussion and take corrective actions. Thus, the binary decision about the compliance status of a business process to a compliance rule is not helpful enough. Whenever a process model fails the compliance checking with a rule, an explanation of the possible violation(s) must be reported to the user. The reported violation must be in a form that is readable to the user who is not necessarily a technical expert. Thus, the explanation must

be in business terms rather than in technical terms. This complements $Req.$ 1in the sense that underlying formal checking must be transparent to the user.

The violation explanation approach must be exhaustive. That is, every possible violation of a compliance rule must be detected. If only a subset of violations is found, this will necessitate the call for a second round of verification after fixing the so-far-found error. But, if all violations are guaranteed to be found, they can be fixed in a single round. This exhaustive violation explanation saves the time and effort of business experts and compliance officers.

The violation explanation procedure should be automatically invoked at the time the verification procedure determines non-compliance. The final result to the user should be either that the process is compliant, or it is not compliant along with the explanation.

2.2.7 Req. 7: Resolution of Violations (Optional)

It is preferable to suggest remedies to compliance violations when possible. These remedies are in the form of process model reforming to be compliant. While resolving violations can be seen as a pure human expert's task, it might be possible to partially automate violation resolution.

In the situation where trivial updates to the process need to be taken to enforce compliance, automation can relief the expert and let him focus on the more complex situations. On the other hand, when automated resolution is not possible, the user is informed about what is missing in order to resolve violations. The identification of missing information can be seen as an aid to estimate the effort needed to restore compliance. This is especially true in situations where the business objective behind a business process *contradicts* the compliance rule. For instance, providing flexibility of receiving money after sending goods against the firm requirement of not sending out goods before receiving the payment.

Automating violation resolution depends on the domain knowledge. Thus, expert system approaches could be used to extract business expert knowledge and encode them in a way that allows their reuse. While extracting the knowledge is out of scope, the encoding of this knowledge should be in a formal and technology independent way. The formality allows automated reasoning; the technology independent representation allows the reuse of this knowledge in different situations and for different purposes.

Suggested remedies must respect correctness and operation-ability of business processes. This stems from the different natures of rules and processes as discussed earlier. Rules are focused on a specific situation, while processes contain sufficient details to be operational. The resolution approach must fill this gap. That is, violation resolution must suggest remedies in details sufficient for keeping the process operational. In the mean time, the merge of these remedies within the process should keep it consistent and correct. For instance, if a violation can be fixed by inserting some activity in the process model, it is necessary to be sure that all prerequisites for this activity will be present at the activity execution in order to avoid deadlocks.

2.2.8 Req. 8: Triggering of Compliance Checking

Usually checking is triggered by users. The compliance support system should be proactive in telling the user about the need to (re)check. This requirement is based on $Req.$ 4. Each time a rule or a process is changed, the system may suggest a rerun of checking. This allows an instant response to changes in the rule repository or the process repository and providing a tight follow up on the compliance status of processes.

2.2.9 Req. 9: Graphical Representation of Compliance Rules

While formal representation of compliance rules allows for automated reasoning, it limits the accessibility only to technical experts. However, for business people, it is necessary to discuss about these compliance requirements before putting them into production. Thus, compliance requirements need to be presented in a way comprehensible to business experts and stakeholders. On the other hand, discussing compliance requirements in their juridic form is not preferable as well. In many cases, the terms are ambiguous and are not in the business terms. Thus, an intermediary representation of these compliance requirements, preferably in business terms, is needed to ensure the highest possibility of uniform perception and to reduce chances of conflicts.

2.3 Contribution of the Thesis

In this section we layout the contribution of the thesis to come up with an approach for compliance management of business processes at design time. Our approach realizes the set of requirements discussed in Section 2.2. However, we might assume some techniques or a level of maturity to exist in order to the proposed approach to work, we discuss relaxations to these assumptions in Chapter 9.

The choice of a verification approach to a large extent controls the level of automation that can be achieved. Thus, a careful choice of that verification technique will drive the decision about subsequent steps of, for instance, automated reasoning, violation explanation and consistency checking.

Of course, another factor to choose a specific verification approach, and thus a specific formalism, is the targeted compliance aspects to be checked. As stated in Section 1.2, we are concerned with static verification of control and data flow compliance aspects of business processes. In this regard, for a design time checking there are three possibilities: 1) simulation 2) model checking 3) logical inference via theorem proving. Simulation effectiveness is bounded by the experience and skills of the user. Thus, it is a manual approach that is not guaranteed to cover all possible executions of a process model. The other two approaches provide a level of automation above simulation.

Model checking [24] algorithms depend on the exhaustive state space search in order to prove the satisfiability of a property specified as a temporal logic formula. In some cases, model checking performance suffers from state space explosion. On the other hand,

theorem provers use logical inference techniques to decide whether a property expressed with first order logic is derivable from the knowledge about the investigated system. Thus, theorem provers do not suffer from the state space explosion problem. However, in many situations, human intervention is required to help theorem provers reach a decision [102]. This intervention is expected from a person who has a deep knowledge about logic and capable of handling complex logical formulas.

For the case of compliance rules verification, we believe that model checking is preferable over theorem proving. Firstly, the nature of compliance requirements addressed by this thesis, e.g., dependency between activities, can be well represented as temporal logic formulas. Secondly, the execution semantics of business processes have been always interpreted in terms of states which makes the choice of model checking intuitive. Thirdly, the problem of state space explosion for model checking can be worked around in a number of ways: 1) Symbolic approaches for representing the state space of systems provide a compact way that avoids state explosion. 2) Employing state reduction techniques based on the verified property. Fourthly, model checking is fully automated.

In the business process management community, process models' behavioral soundness have been heavily studied, e.g. [3, 30]. One benefit is that formal execution semantics have been assigned to the different process modeling elements e.g., semantics of EPCs [137], semantics of BPMN [31], etc. However, these semantics were only concerned with control flow aspects. As we are concerned with data access semantics also, we had to provide a formal access semantics that integrates well with existing control flow semantics. The discussion about process models and their execution semantics is given in greater details in Chapter 4.

To define a compliance rule to be checked against a process model via model checking, the user has to represent the compliance rule as a temporal logic formula. It is not possible to assume that every person involved in compliance management is aware of such complex mathematical notation. Moreover, in $Req.$ 9 "Graphical representation of compliance rules", we argued that compliance rules should be represented in a way readable to a user who is usually more on the business side. In [39, 76] the authors proposed a visual language that allows the users to express compliance rules in a graphical way while encapsulating the formal property and generating it transparently to the user. We believe that this approach provides a balance between expressiveness and formality on the one hand and ease of use on the other hand. Thus, we will provide the user with a set of visual patterns to model compliance rules and an approach to extend these patterns. The approach is to define a visual pattern and assign it a temporal logic template. With this approach, we have a formal representation of compliance requirements, $Req.$ 1, a visual representation easy to use by the compliance officer and business experts, $Req.$ 9, and we are able to automatically decide about (vacuous) satisfiability of compliance rules, $Req.$ 5.

To represent a compliance pattern, compliance rule type, we use BPMN-Q [7, 124]. Originally, BPMN-Q was designed as a visual language to query repositories of business

process models. BPMN-Q provides a set of abstract concepts, as will be discussed in Section 4.2, that are adequate to express compliance requirements. While BPMN-Q is similar to BPMN [1] in notation, we explain in Chapter 4 that our approach is indeed generic.

With the use of BPMN-Q queries as a means to express compliance rules, we can store these compliance queries as separate artifacts, $Req.\ 2$. Arbitrary metadata describing any kind of attributes of interest to the compliance officer can be attached to queries. A metamodel describing how to track compliance related artifacts can be found in [64].

To correlate compliance rules to other compliance rules and to business processes, $Req.\ 4$, we currently assume a common set of tags that are used to annotate both compliance rules, i.e. queries, and business processes. In the simplest form, tags are short text that have common interpretation within the organization. For instance, a tag "claim handling" is interpreted equally both by business expert and compliance officer. The availability of common interpretation of these tags is considered as a level of maturity that we assume to exist within the organization. With the use of tags, we can automate the correlation between processes and rules, yet, in a loosely coupled way. Any change to the process, the rule, their tags would flag the need to rerun checks, cf. $Req.\ 8$.

As rules can be correlated to each other, it is possible to identify sets of related rules by means of finding a common set of tags. To this end, consistency checking among them can be conducted, $Req.\ 3$. As we said earlier, the choice of the formal language determines the level of automation. As temporal logic is the formalism to express compliance requirements, consistency checking to a large extent benefits from well established approaches in that regard. However, as we will show, these approaches are useless unless domain specific knowledge is well reflected in checking for consistency.

Whenever a compliance rule is not satisfied by a process model, the user must receive an intelligible feedback explaining how the violation occurred, $Req.\ 6$. In this regard, we benefit from the querying nature of BPMN-Q and define violation scenarios as anti patterns, in contrast to compliance patterns. For each compliance pattern a set of anti patterns is defined. Each anti pattern is represented also as a BPMN-Q query. When the anti pattern query is structurally matched to the business process, the matching part of the process highlights execution paths that caused the violation.

Finally, to suggest remedies, $Req.\ 7$, we employ another toolset where we depend on a structural analysis of business processes. Based on this analysis, we devise a set of compliance resolution algorithms that depend on automated planning [97].

Figure 2.1 summarizes our approach for compliance management. First and foremost, the user, e.g., compliance expert, models the compliance rule by instantiating a compliance pattern. In addition to instantiation, the user assigns a set of tags to the compliance rule. With the tags assigned, the system can identify related compliance rules and business processes respectively. The details of modeling compliance rules, checking their consistency and checking them against business processes are given in Chapter 5.

Whenever, a compliance rule checking is negative, the system generates the corre-

sponding anti patterns, matches the anti patterns to the process models and present the user with the parts of the process causing the violation. Chapter 6 covers the details of obtaining anti patterns and representing them as queries.

The next and last step is to suggest remedies, if possible, to the user. This is covered in Chapter 7.

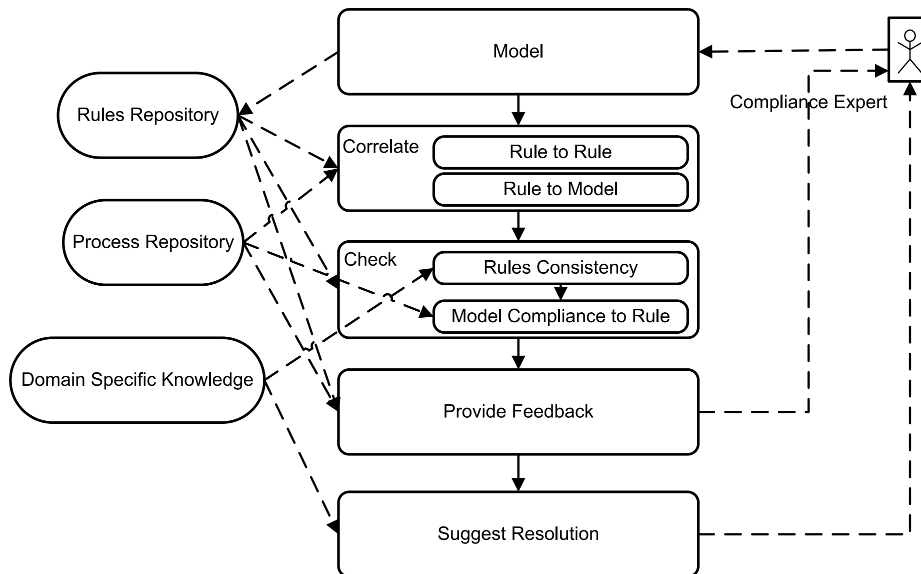


Figure 2.1: Compliance management approach

Assumptions

We assume a set of facts upon which our approach depends:

- A central process repository: As indicated in Figure 2.1, we assume process models subject to compliance check to reside in a central repository.
- Correct process models: we assume investigated process models to be correct in the sense that they are free from modeling errors causing deadlocks, live locks. We can justify our assumption with the fact that compliance requirements are some sort of a next step correctness criteria. Thus, it is logical to check first level correctness, e.g., deadlock freedom, before checking advanced correctness criteria imposed by compliance rules.
- A common set of tags to be assigned for process models and rules to achieve correlation: This is a reasonable assumption as it is common that business processes and compliance requirements refer to the same business concepts and terms. Actually, in the phases of discovery and extraction of compliance requirements, cf. Figure 1.1, the identification and internalization of requirements is based on identifying relevant business terms [71].

- A common set of activity and business object labels: this is related to the above assumption. We assume that whenever a business activity label appears in a process model or compliance rule, they reflect the same business semantics. For instance, "Process purchase request" means the same thing for both business process modeler and compliance rule modeler.

- Explicit domain knowledge: Figure 2.1 indicates the importance of domain-specific knowledge in order to automate the checking, explanation and resolution steps. We assume this knowledge to be present and encoded in the way we discuss it in Section 4.4.

In Chapter 9 we discuss approaches to relax these assumptions.

Chapter 3

Related Work

While this thesis is concerned with design-time compliance management of business processes, in this chapter, we widen the scope of related work to include support of compliance at the different phases of a business process life cycle. We will extensively discuss approaches concerned with design-time compliance management and will evaluate each against compliance requirements introduced in Section 2.2. Section 3.1 discusses other compliance management frameworks proposed for business processes. Design-time compliance management approaches are detailed in Section 3.2. Section 3.3 discusses approaches for monitoring the adherence to compliance requirements at process execution. Compliance auditing is discussed in Section 3.4.

3.1 Compliance Management Frameworks

In [161] authors discussed how risk management can be integrated in the business process management life cycle. The authors have identified a set of risk factors and defined general strategies to deal with them. Noncompliance to regulations is a risk factor. Thus, it is necessary to establish measurements for risk of that type, to document it, and to establish control objectives that ensure compliance and avoid that risk. In this section we give an overview on compliance management frameworks that addressed the issue of documenting and tracking compliance requirements. Note that we are not concerned with how control objectives are realized or checked; rather, we focus on the concepts of risk, regulation and internal controls.

In [121] authors introduce the notion of a control directory as a means to manage compliance requirements. Moreover, the authors correlate the control management life cycle with the business process management life cycle. Within a control directory, each entry is described by the control requirement, the risk the organization takes in case of violation and the internal control that must be established in order to be compliant. The work in [95, 94] extends the work discussed above by describing the different roles

involved in compliance management, namely business process expert, compliance expert and the external auditor. Moreover, the authors discuss different recovery actions in case of violation that can be attached to each control objective. Finally, the authors classify the control objectives into patterns that they claim to occur frequently in regulations. The framework is oriented to runtime monitoring of control objectives. The authors agree with our view about the need to maintain a separate control directory, what we call compliance rules repository. However, the framework does not discuss the need to develop mechanism to decide inconsistency between related compliance requirements, $Req.\ 3$. While the framework supports run time compliance management, $Req.\ 6$ and $Req.\ 7$ regarding providing useful feedback and supporting automated resolution of violations are not relevant. The framework puts the task of correlating processes to compliance rules totally on the human side, $Req.\ 4$. The framework proposes the formal contract language (FCL) to represent compliance rules. The language, as the name tells, is a technical one. We believe that the use of technical representation only limits the chance to argue about the rules.

[127] discusses an ontology based approach to represent both processes and compliance requirements. The approach classifies compliance requirements into syntactic, semantic and pragmatic. The compliance checking then is reduced to the problem of checking whether the process ontology instance realizes the compliance ontology. The work in [69] follows a similar approach. The two frameworks, thus, support $Req.\ 1$, which allows to formally reason about compliance. Also, developing compliance specific ontology could be seen as a satisfaction of the separate maintenance of compliance rules, $Req.\ 2$. However, it is not clear whether the frameworks allow to attach metadata to track the sources of the compliance requirements and risk factors attached to them. The frameworks support neither providing useful feedback in case of violation, $Req.\ 6$, nor automated resolution of violations, $Req.\ 7$. However, consistency issues among compliance rules, $Req.\ 3$, can be identified while building the compliance ontology. Correlation between processes and rules, $Req.\ 4$, is achieved on the ontology level.

Authors of [155] provide a methodology to semi automatically extract and document compliance requirements. To extract compliance requirements, natural language processing techniques are used to identify relative terms. Later on, the extracted terms are put into a taxonomy. This framework is oriented to support the automation of compliance requirements extraction from legislative documents. Thus, it does not intersect with the scope of this thesis. However, consistency checking mechanisms can be applied early at that stage.

[96] discusses the challenge imposed on organizations to come up with internal controls to ensure that business operations comply with requirements. The internal control is itself a process that includes: identification of significant accounts, definition of control objectives, $Req.\ 2$, identification of relevant processes, $Req.\ 4$, and risks behind not obeying them. Challenges of compliance management can be attributed to two factors. First, the realization and testing of internal controls is known to be an expensive and a

time-consuming operation. Second, the compliance requirements are usually very high level and without any recommendation on how to realize these requirements. The authors discuss high level concepts like, a control objective, a business process and a risk. The authors relate these concepts to each other. However, the authors do not discuss how or when to check compliance rules against business processes. Moreover, the authors do not propose a specific formalism to express compliance rules, thus, requirements $Req.$ 1, $Req.$ 6, $Req.$ 9 and 7 were not addressed. Moreover, consistency checking among compliance requirements were not addressed.

The work in [69, 70] discusses a framework to define and integrate compliance requirements within the organization by means of policies. As compliance is a vertical concern, i.e., it is approached at different levels of details, the authors propose the use of ontologies as a means to control the level of abstraction needed for both business processes and compliance requirements. As compliance requirements are abstract, e.g., laws, the authors propose to integrate them in the organizations goals and strategies. This will reflect compliance requirements on finer-grain levels like operational processes, business objects and security policies. The authors suggest the use of business rules as a means to realize policies and to monitor the enforcement of compliance requirements on process models and on process execution. To this end, consistency checking among compliance rules, $Req.$ 3, the need to explain violations and provide useful feedback, $Req.$ 6 and support for suggesting resolutions to violations, $Req.$ 7 were not recognized by that framework.

Ly et al. [81, 84] introduce SeaFlows, a compliance management framework supporting so-called lifetime compliance. The framework points out the need for a separate constraint repository. SeaFlows distinguishes itself in support to design time validation and the support of controlling instance adaptation at runtime. To support design time validation, SeaFlows allows more than the binary yes/no decision about compliance. That is, the framework can identify full compliance, definite violation and conditional violation. The framework supports providing the user with textual description of violations. However, it is not clear how the framework supports consistency checking among compliance rules, $Req.$ 3. Moreover, the correlation of rules to processes is left to humans with no automated support, $Req.$ 4.

In [19] the authors develop an enterprise-wide approach to help assess risk and achieve agreement among different stakeholders about regulations and the controls to be created to realize these regulations. The authors determine the relationship between the number of regulations, disagreement between stakeholders, risks, controls, costs and compliance degree. Based on these terms and their relationships, authors came up with a model to track compliance by means of identifying *business units* that have to show compliance to certain *regulations*. To achieve compliance, *control activities* are established. These controls are realized via *IT solutions*. The framework focuses on achieving agreement among different stakeholders regarding a specific regulation on a high level. Thus, none of the compliance requirements discussed in Section 2.2 are addressed by that framework.

Service oriented architecture (SOA) is seen as an implementation architecture for business processes. To ensure governance of compliance, the compliance-driven models, languages, and architectures for services (COMPAS) [2] project aims at providing a life-time support for compliance for processes implemented as services. The overall architecture and the research objectives of the project is described in [27]. To support compliance at process design time, COMPAS follows a compliance-driven approach. That is, process fragments are designed to be compliant with rules of interest. To come up with operational processes, process fragments are enriched with sufficient details that make them operational and are deployed. At runtime, compliance is monitored by means of logging events from both processes and execution engines. Later on, these events are audited to assess the compliance state of the organization.

Evaluating COMPAS against our requirements discussed in Section 2.2, we can see that $Req.$ 1, formal specification of compliance rules, is not considered as the framework tends to build process fragments that are compliance-driven. However, due to the changing nature of compliance requirements, it is necessary to identify the compliance status of already developed processes to new requirements. Consequently, $Req.$ 6, providing useful feedback, and $Req.$ 7, resolution of violation, are not addressed as the framework tends to compliance-driven process design. Also, the notion of consistency checking between different compliance rules is not addressed, $Req.$ 3.

Table 3.1 summarizes the evaluation of the above mentioned frameworks against our requirements for a compliance management framework. For each approach we describe the formal language being used to represent compliance rules, the support for separate rule base, the awareness of the need to check consistency, the approach to correlate rules to processes, the ability to report about vacuous compliance, the ability to explain violations, the ability to suggest remedies to violations and the representation of rules to users. The use of +,+/-, or - indicates full, partial or lack of support to a specific feature.

Framework	**Formal language**	**Separate rule base**	**Consistency among rules**	**Correlating rules to processes**	**Vacuous compliance**	**Explanation of Violation**	**Resolving violations**	**Rule representation**
Sadiq et al. [121]	FCL	+	-	Manual	-	-	-	Textual (FCL rules)
Namiri and Stojanovic [94, 96]	-	+	-	Manual	-	-	-	Pattern based
Schmidt et al. [127]	OWL	+	+/-	Automated (via ontologies)	-	-	-	Textual (via ontologies)
Yip et al. [155]	SWRL	+	+/-	Automated (via ontologies)	-	-	-	Textual (via ontologies)
ElKharbili et al. [69, 70]	Business rules	+	+/-	Automated (via ontologies)	-	-	-	Textual (via ontologies)
Seaflows, Ly et al. [81, 84]	Not specified	+	+/-	Manual	-	Textual explanation	+/-	Not specified
COMPAS, Daniel et al. [27]	Not specified	+	-	Manual	-	-	-	Domain specific languages

Table 3.1: Evaluation of compliance management frameworks against requirements in Section 2.2

All frameworks recognize the need to separately maintain compliance rules. However, only a subset discusses the need to establish consistency checking mechanisms among them. Depending on the formalism for capturing compliance rules, consistency checking can be gained. For instance, using business rules, one can benefit from well established approaches for consistency checking [49].

None of the approaches envisioned the support for vacuous satisfiability, violation explanation and violation resolution except for the SeaFlows [84], where the framework provides textual explanation of compliance violation. Textual explanation might be hard to reflect, by the user, on the process level, in case the process is large or that it has multiple appearances of the violation. In the latter case, there is a threat of incompleteness of identifying problematic parts of the process.

Most of the approaches do not distinguish between the formal representation of the rule and the representation for business users. This is a limitation as it hinders the understandability of non-technical users. When rules are negotiated on a business level, compliance expert has to provide a manual mapping from formal representation to a notation understood by business people.

3.2 Design Time Compliance Checking

Process design time compliance can be categorized into compliance-driven process design and compliance verification. Compliance-driven design aims at enforcing compliance and ensuring it while a business process is being designed. In compliance verification, processes are designed independently from compliance awareness and are checked later for compliance.

Compliance-driven design guarantees violation prevention as both the business expert and the compliance expert are involved in that step. However, compliance-driven process design cannot be fully automated. This approach is useful in specific cases where processes are derived from a certain specification, e.g., business contracts, and are kept for a predefined duration, e.g., the duration of a business contract.

On the other hand, compliance verification allows more flexibility and a higher degree of automation. Process design and compliance checking are decoupled. This allows to automate and repeat the check each time a compliance requirement is added or changed, or a process is added or changed. However, this approach requires a higher degree of maturity in order to automate in terms of means to correlate compliance requirements to business processes. For this approach, automated verification tools can be employed to accelerate the checking process and have accurate results. However, still the resolution of violation is to a large extent on the human side.

In the rest of this section we will explore literature following the above classification. We start by approaches supporting compliance-driven design and then we discuss approaches for compliance verification. For each approach we describe the language or

tool used to represent the compliance rules, the reasoning approach, and the category of compliance requirements, cf. 2.1, addressed.

3.2.1 Compliance-driven Design

Goedertier and Vanthienen introduced an approach to support compliance-driven business process design in [50]. The authors propose the use of business rules as a means to guarantee compliance with regulations in a flexible way. Business rules reflect the requirements imposed by either external regulations or internal policies. Thus, there is no need to hard code these requirements directly in the process control flow. To express policies, the authors developed "Process ENtailment from the ELicitation of Obligations and PErmissions" (PENELOPE) as a declarative language to capture obligations and permissions imposed by business policies in the form of temporal deontic expressions that are focussed on sequencing and timing constraints between activities in a business process. To guarantee a compliant process, the authors provide an algorithm that generates a compliant process model from a set of PENELOPE rules. However, the authors clearly indicate that the generated compliant process model is not intended for execution; rather it is intended as an aid to the user to verify executable processes against it. The approach focuses on compliance with control flow aspects. We believe that the generation of a process template is an unnecessary step. Processes could be verified directly against rules in PENELOPE. Nevertheless, PENELOPE can identify conflicts among rules, $Req.$ 3.

Another approach to support compliance-driven design is presented by Milosevic et al. in [92, 93] where authors use the formal contract language (FCL), introduced in [52], to express contract permissions, obligations and prohibitions as a set of FCL rules. The generation of compliant business processes is achieved in a progressive manner. At the abstract level, interactions among business partners of the contract are identified. In the next step, internal details for each partner's process are added. These details depend on the activities mentioned on the partner's side in the business contract. The authors claim to cover the four different aspects of compliance for business processes, cf. Section 2.1. This is achieved via annotating individual tasks within business processes with predicates describing each aspect. For instance, if a task duration is not allowed to exceed 3 days; it is annotated with predicate `max_duration(task,3,days)`. The annotation is meant to help in monitoring the generated processes at run time.

In the field of service composition, a framework for guiding service compositions based on PROPOLS [55] proactively suggests next step activities in a composition in order not to violate temporal business rules. While building a service composition, e.g. a business process, is considered as a human task. PROPOLS interactively suggests, based on predefined temporal business rules, to insert, delete and/or reorder activities to be compliant. Design mistakes, i.e., contradicting to the business rules, are identified on the fly and the designer is informed about the error. To achieve this, a finite state automata (FSA) is derived from the set of temporal business rules and the currently developed process schema. Thus, deviations and possible future moves can be identified.

The approach focuses on control flow aspects regarding sequencing and occurrences of activities. While it is beneficial to proactively guide the designer, the in-progress model checking would be of too much overhead regarding processing time to reconstruct the FSA with each update made to the process model. The algorithm to suggest valid next steps depends on traversing all paths within the partially developed process FSA until the currently selected activity. This incurs a huge overhead as this step is repeated with each insertion of a new activity. For suggesting remedies to violations the same approach of traversing process FSA is employed to identify erroneous paths. However, these paths are identified on the state level rather than on the process structure level, in contrast to $Req. 6$. Moreover, the performance of path finding will degrade severely with the existence of concurrent paths within the process definition. Additionally, the suggested remedies are bound only to the knowledge encoded in the business rules. However, there might be other activities needed to be inserted or deleted to keep the business process operational. The need for this domain knowledge is not recognized by the approach, we elaborate more on that in Chapters 4 and 7. In [156, 157] the authors extend the use of PROPOLS to a semi-automatic compliance-driven business process design.

During the execution of business processes, data elements are manipulated to determine next steps. In general, business processes manipulate business objects, e.g., order, insurance claim. These objects are described, among other aspects, by their allowed states and transitions among them. To this end, Kuster etl al. [74] describe an approach to generate business process models that are compliant with business objects life cycles. This is useful as usually the definition of object life cycles are driven by organization's internal policies and external regulations. To generate compliant processes, individual activities in a process are the means to bring a business object from one state to the other. The generated process models are guaranteed to leave the data objects in one of the prescribed final acceptable states. However, the approach is not fully automated as in case of several input object life cycles, synchronization points among them have to be defined manually.

Schleicher et al. [125] propose the preparation of process templates that are implicitly compliant. Later on, these templates are modified by process experts to adapt them to certain business needs. To ensure compliance, the adapted templates have to be rechecked for compliance to assert that no violations are possible due to modifications made by the process expert. The authors devote an algorithm to ensure compliance after inserting details in the process template to be operational.

Evaluation

Most of the compliance-driven approaches aim at generating process templates that are compliant with certain regulations. These templates are adapted by users for certain business objectives by means of adding more details to make them operational. To this end, a posteriori checking is needed to ensure that the business processes are still compliant and no violation is introduced due to an unintentional breaking of compliance

by a designer. Also, each time regulations change there is a need to a posteriori check to ensure compliance. Thus, we believe that compliance-driven process design is useful in specific cases where processes can be fully derived from a specification, like a business contract. Otherwise, there is always a need to recheck.

3.2.2 Compliance Verification

In [39, 40, 41], Forster et al. proposed the Process Pattern Specification Language (PPSL) as an approach to visually express quality, compliance and constraints regarding the process behavior. PPSL is an extension of UML Activity Diagrams [100] that extends edges with the `<<after>>` stereotype to indicate that two activities are not necessarily are required to be in strict order. Moreover, they are able to express complex constraints with all types of logical connectors. To verify properties, PPSL patterns are translated into temporal logic that are model checked against the process model. Execution semantics of process models are given via so-called dynamic metamodeling [131, 36] which in a way is similar to the token flow semantics of Petri nets. The approach focuses on control flow aspects of compliance rules. PPSL uses temporal logic as a formal background, cf. $Req.$ 1. Also PPSL patterns are visually represented in way close to that of UML Activity Diagrams, $Req.$ 9. This gives a great chance for business expert to define and argue about compliance requirements. From an expressiveness point of view, patterns are not capable of expressing negation. That is, certain activities must never be executed when other activities have already been executed. Moreover, the approach is limited to defining rules concerning control flow aspects only. Also, no support is given beyond verification. That is, there is no helpful feedback given to the user in case of violation, cf. $Req.$ 6. Let alone the suggestion of remedies to violation, $Req.$ 7.

Taking business contracts as the source for compliance requirements, the authors in [53] used FCL to formally measure the compliance between a business contract and a business process. As stated earlier, it is possible to express obligations prohibition and permissions within a FCL rule. Moreover, it is possible to express several levels of exception handling when the basic behavior is violated by the business process. FCL is further used in [80, 79] to help business process designer quantitatively assess the degree of compliance of a process with a rule. Based on a compliance rule, a process can be either ideal, sub-ideal or non-ideal. To reason about the idealism of a process against an FCL rule set, the authors in [54] provide an algorithm that follows a theorem proving approach to verify the rule against the process. To reach this, activities with a process are annotated with certain effects [51] upon which the algorithm reasons. The current reasoning algorithm does not support loops within process models. A similar approach to verify contract regulated service composition can be found in [77]. The level of idealism reported for a business process against a rule can be seen as a support for explaining the compliance status of a business process, $Req.$ 6. However, the approach does not support the notion of violation resolution, $Req.$ 7. Moreover, FCL rules are represented purely as textual formulas. This limits to a great extent the ability of business experts to understand,

develop and argue about these compliance requirements.

Approximate compliance checking for annotated process models is discussed in [148, 58]. The authors argue that verifying a process against a compliance requirement is an NP-hard problem and thus approximate approaches are needed as a work around that limitation. The authors provide an approach that is in general neither sound nor complete. That is, in some cases they are able to find *some* of the violations. In some other cases they cannot identify non-compliance. Also, the verification algorithms are limited to process models without loops. The verification is in a form of theorem proving and thus needs to propagate the effects of activities throughout states. One merit of that approach is the ability to trace back from the violating state to the activity that caused the violation. However, this is only useful to compliance rules of local nature, e.g., restrictions on resource allocations like separation of duty. However, violation to execution ordering compliance rules cannot benefit from this diagnosis feature. The approach seeks compliance rules with effects local to specific activities within the process.

Another approach that addresses compliance verification using ontologies is presented in [56]. The paper describes a prototypical implementation of checking compliance of business processes, annotated with ontological terms, against business rules capturing compliance requirements. Upon violation, the tool is able to highlight the activity that caused the violation. The approach addresses compliance issues related to existence of specific activities within the process. Compliance requirements related to the execution ordering between activities are not covered. There is no support for resolving violations.

In [76], a formal approach based on model checking was given to check for compliance of processes defined in BPEL against constraints defined in the Business Property Specification Language (BPSL) that are translated to linear temporal logic. The approach supports both control and data flow aspects for compliance checking. Similar work that verifies BPEL processes is described in [158] where authors propose their own language PROPOLS to capture patterns to be checked against a business process. Upon a violation detection, the authors in [76] show the user execution traces on the state level that caused the violation. As known from standard model checking algorithms, these execution traces, also called counter examples, are automatically given by the model checker. In comparison to $Req.$ 6, providing useful feedback in case of violation, the state-level counter example is not useful to the user as it is presented in technical terms. Moreover, counter examples are not complete, as they report the first met violation only, we elaborate more on that in Chapter 6. The requirement to provide remedies for violation is not addressed by that work. Moreover, BPSL patterns seem not to support modeling negation.

Ly et. al [82, 83] discuss an approach for managing and verifying semantic constraints within adaptive process management [150, 139]. The authors describe semantic constraints as a form to express compliance rules. The approach extends earlier work on supporting adaptivity and change propagation from process models to instances, and vice versa, from a syntactical correctness [112, 114] to a semantical one. The extension is in the form of formalizing and integrating domain knowledge within process management

systems. To verify constraints, the authors depend on execution traces. An execution trace is a finite sequence of events. An event corresponds to the completion of a task. The constraints are checked against these traces for satisfiability. The approach is concerned with control-flow aspects of compliance. The approach for verification is limited to processes without loops. Also, the need to provide useful explanation of violation is not supported. Let alone the suggestion of remedies.

In [72], the authors present an approach to model business processes and verify their behavior against compliance rules based on a formal language called Reo. Process modeling constructs, e.g., activities, choice, parallelism, are mapped to Reo circuits. A Reo circuit is a sort of logical circuit. Constraint automata (CA), a special type of finite state machines, are derived from Reo circuits in order to enable the reasoning about the process behavior. The approach is claimed to cover all aspects of compliance modeling by means of utilizing special extension of the (CA) for the specific aspects, for instance TCA extension can be used to address compliance rules related to timing and deadlines of activities. Evaluating the approach against our requirements, the automated support is limited to verification. That is, there is no support for providing useful information in case of violation. Also, the compliance rule is assumed to be written directly in the respective formal language used for verification. So, a user-friendly representation is not considered. Also, there is no means to support resolution violation.

In [46] Ghose and Koliadis propose an approach to check compliance of business processes and to resolve violations. In order to determine the compliance status of a business process, activities within the process must be annotated with their immediate effects. An annotated process is then mapped to so called semantic process networks (SPN). SPN is a directed graph that serves as an input to the verification algorithm against compliance rules. Authors provide their own algorithms to verify properties related to execution ordering of activities. To resolve violations, heuristics are used. The approach is focused on control-flow aspects. Also, the approach is not fully automated as a business analyst is required to manually accumulate activities' effects throughout the whole process model, in contrast to automated propagation discussed in [148]. This cumulative propagation of activity effects is needed by the verification algorithms since it is designed to check compliance locally at each single activity. Of course, with medium to large process models, the chance to forget or to mis-propagate the effects is very likely. Thus, the result of verification cannot be trusted. While explaining violations is not addressed. There is a limited support for resolving violations by suggesting adding, removing, or reordering of activities to gain compliance. These suggestions do not take into consideration dependencies between activities. For instance, the approach might suggest to reorder activities A and B in a way that A executes after B while there is a data dependency from B on A. In such case, reordering will make the process compliant but will not be operational since the execution will deadlock due to unfulfilled data inputs of B.

Kumar and Lui in [73] address the resource aspects of compliance. Using a pattern

based approach, the authors express role based access controls to tasks in a process model. The authors discuss how variants of the separation of duty principle can me modeled in Prolog and verified against process models, where activities are annotated with roles that will perform them. A similar work is discussed in [153] where Wolter et al. provide an extension of BPMN [1] that allows the user to visually specify task based authorization constraints. In [152] Wolter et. al use model checking to verify whether a process model satisfies security constraints like separation and bind of duty. In [89] Mendling et. al study the different ways to express separation of duty constrains using BPEL4People [4].

Table 3.2 summarizes our evaluation of compliance verification approaches. The evaluation is based on satisfaction of compliance requirements discussed in Section 2.2.

Approach	Checking approach	Representation	Violation explanation	Violation resolution	Covered compliance aspects	Limitations
Forester et al. [39, 40, 41]	Model checking	Visual Patterns translated to temporal logic formulas	-	-	Control flow	Negation is no covered in supported rules.
FCL-based contract verification [53, 80, 79, 54]	Theorem proving-like proprietary algorithm	Textual formulas	partially addressed by the different levels of idealism	-	all	reasoning algorithm are limited to processes without loops
Static Compliance checking, Liu et al. [76]	Model checking	Visual representation in BPSL translated to temporal logic formulas	Only on the state level	-	Control and data flow	BPSL rules cannot express negation and the violation explanation is not complete
PROPOLS, Yu et al. [158]	Model checking	Textual patterns	Only on the state level	-	Control flow	The violation explanation is not complete
Ly et. al [82, 83]	Proprietary algorithm	Textual	-	-	Control flow	The verification algorithm is limited to process models without loops
Reo, Kokash and Arbab [72]	Model checking	Textual	-	-	All	
Ghose and Koliadis [46]	-	Textual	-	Heuristic pattern-based resolution suggestions	Control flow related to ordering of activities	The approach cannot be automated
Approximate compliance checking [148]	First order logic	Textual	Locating activities causing violation	-	Compliance requirements of local effects, e.g., separation of duty	The approach is not complete with respect to identifying violations
Role patterns, Kumar and Lui [73]	Logical inference	Prolog rules	-	-	Resource allocation	
Wolter et al. [153, 152]	Model checking	Visual patterns translated to temporal logic formulas	-	-	Resource allocation	

Table 3.2: Evaluation of compliance verification approaches

3.3 Runtime Compliance Monitoring

In this section we give an overview of business process compliance monitoring at process runtime. With monitoring we mean instant monitoring of running process instances. Monitoring takes place by a process execution engine that notifies interested agents about the occurrence of specific events during the process execution.

A business contract defines the duties and the rights of the contracting parties. In real life, violations are subject to penalties. In an e-business environment it is crucial to define and monitor the satisfaction of so-called e-contracts. [91, 52] discuss the problem of following up the execution of e-contracts. One of the mechanisms it discusses is the *monitoring* of contract execution. By monitoring of significant events, the comparison of actual behavior to the expected behavior is possible. In case of violation, several resolution mechanisms are offered depending on the severity of the violations. For instance, a notification can be used to alert contract participants about possible upcoming violations, e.g., in contract between an enterprise and an Internet service provider the service might be allowed to fail no longer than 4 hours a week; otherwise ISP is subject to penalties. In this case, a contract execution monitor might send reminders for the ISP if the 4 hours allowance is about to expire. When a deviation takes place, i.e., the notification did not help prevent that deviation, mediation mechanisms try to find an agreement between contracting parties in order to put contract execution back into order. Finally, arbitration is the step to do when no mediation possible. Similar approaches can be found in [62, 116, 6].

In [149], Weigand et al. similarly take business contracts as the basis to align workflows of partner enterprises into a virtual workflow for the virtual enterprise. The virtual enterprise is modeled using a component definition language (CDL). Each component represents a partner. Later on, the workflow of each enterprise is linked to the business contract. Thus, a contract defines the legal interactions between components. To express the mutual commitments among parties, business objects are used to describe the semantics of cooperation in business terms, e.g., Customer, Order. To work on these business objects, business tasks are defined in a way that accesses and transforms these objects. Finally, business tasks are assigned to roles within organizations and their order of execution is defined, the workflow. To monitor the execution and adherence to the business contract, control workflows are defined. Control workflows define the sequence and direction of exchanged messages, according to the contract. Thus, a global contract object coordinates the execution of the various cooperating workflows. Although the approach provides a practical solution for virtual enterprises, the authors did not illustrate which partner owns the contract object.

Giblin et al. [47, 48] propose REALM as a metamodel to express compliance policies covering all aspects of compliance discussed above. A policy is a rule set that has a scope of applicability. Rules are expressed by means of real-time temporal object logic. This gives expressiveness not only to capture ordering between events; rather, the actual times of their occurrences. This is necessary to express timing constraints, e.g., an activity must

not take more than two days. In addition to enforcing these rules on process definitions, they are monitored at runtime. This is important for time constraints, as design time verification might not be sufficient.

Ontology languages are used by ExPDT [63] to express privacy compliance rules. With ExPDT, security policies concerning both data and access to them by business processes can be expressed. The language seems to cover three aspects of compliance. That is, one can express a ExPDT rule indicating conditions on the user, the data and the action taken in the business process. The rules are monitored at runtime to enforce compliance.

The work in [78] discusses the achievement of IT-controls as means to monitor compliance of SOA-based business systems. The IT-control is a software artifact that can interact with automated business processes. The approach focuses on security related controls. To enable monitoring, the authors propose an architecture that extends a SOA by components for signaling and monitoring events caused by services, for their aggregation and analysis with respect to control objectives

3.4 Compliance Auditing

Usually, the proof of compliance of some organization to regulations is done by an external auditor. The external auditor randomly collects, based on experience, audit trails and logs from the different sectors of the organization for inspection. This step is usually manual and its duration depends on the auditors team, their experience and the maturity of the organization. However, it is possible to speedup this step by providing automated tools that scan system logs, especially process execution engines, to collect evidences about the compliance status of the company.

Process mining [135] is an approach to synthesize process models out of execution logs. To this end, process mining enables organizations to assess the way their processes were actually executed. Based on the extracted processes, auditing can be automated. In [140], Aalst et al. provide an automated approach to detect violations based on workflow logs. Audit checks are formalized in linear temporal logic (LTL) formulas that are model checked against the mined processes. The approach supports the check of control flow aspects of compliance.

The work in [141] uses process mining to discover anomalous process execution regarding security constraints. Security constraints are interpreted as requirements to have certain activities executed in a certain order. It is a two-step approach. The first step is to mine an *acceptable* log. The result of the step is a structured workflow net. To discover anomalous executions, the second step, logs of newer process instances are checked for conformance against the mined workflow net. Although the paper assumes that logs are described not only in terms of activities, but also performers and time stamps for start and end of an activity, security breaches due to, e.g., violation of separation of duty constraints were not discussed.

Doganata and Curbera [32] discuss an approach to semi automatic auditing in cases where there is no process execution engine, so-called unmanaged processes. Business provenance is the means to automate auditing of unmanaged processes.

3.5 Other Approaches

In general, the identification of compliance requirements relative to the organization is a pure human task. Compliance experts along with lawyers have to inspect legislative documents to extract control requirements. In [71], Kiyavistskya et al. provide an approach to help automate the step of compliance requirements identification. The approach is focussed on regulations regarding data privacy and obligations. The authors propose a three-step approach. First, regulation text is annotated to identify fragments describing actors, rights, obligations, etc. Second, a semantic model is constructed from these annotations. Third, the semantic model is transformed into a set of functional and nonfunctional requirements.

Saeki and Kaiya [123] propose an approach to guarantee compliance of developed software systems, e.g., process models in early stage of requirements elicitation. The authors follow an incremental approach where new requirements are examined against the available set of regulations and requirements. In case of conflicts, the user, software engineer, is informed and the tool suggests changes to the requirement in order to keep compliance with regulations.

In [5] Agrawal et al. propose a solution to automate compliance with Sarbanes-Oxley [99] act based on database technology. The approach targets the continuous evaluation of the compliance status by two different ways depending on the maturity of the organization. The first way is via automated auditing of activity logs that are stored in database tables. An auditor can either issue a mining or a querying task against the logs. With mining tasks the auditors reconstruct the actual way business routines were performed and thus can manually determine the deviation from expected behavior. On the other hand, with querying, the auditor can express violations and check the log for their occurrences. The other approach for continuous evaluation is active enforcement. In this case, the business routines are modeled as workflows in a way that guarantees compliance. At enactment time, actual instances are compared against the prescribed workflows to check for compliance. Database technology is used to reflect the prescribed workflows as a set of constraints expressed on the database level. Thus, there is no need to deploy process execution engines.

zur Muehlen et al. [162] argue that a combination of a process modeling language and a business rules language is necessary to cover all concepts needed for compliance management. Specifically, zur Muehlen et al. propose to use a combination of BPMN and Simple Rule Markup Language (SMRL) to model compliance requirements.

In [45] the authors provide a framework to track the compliance with health care regulations. The framework proposes to firstly model business processes of a hospital

and model regulations. Later on, the two models are linked together. The paper focuses on a requirements management framework to establish different types of links between hospital business processes and privacy legislations imposed by health care regulation bodies. While establishing these links, the framework helps identify missing steps, within a hospital business process, to be compliant with regulations.

Chapter 4

Foundations

In this chapter we discuss background and foundations for our work. We discuss concepts and techniques that describe the domain of business processes as well as formal approaches that will be used to address the compliance problem. Some of these foundations are *contributions* of the author in the course of approaching the compliance problem.

The rest of this chapter is organized as follows. First, we briefly introduce the concepts of business process models in Section 4.1. Sections 4.2 and 4.3 discuss the founding contribution of business process model query language. The notion of domain knowledge, another founding contribution, is detailed in Section 4.4. Finally, model checking is discussed in Section 4.5.

4.1 Business Process Modeling

Currently, there are two major paradigms for modeling business process models. Traditional modeling is concerned with imperatively describing process models. Activities and their control flow and data flow dependencies are described. A process definition, model, is enumerating all allowable execution scenarios for process instances. On the other hand, declarative approaches for process modeling [106, 142] give more flexibility for process modeling by just constraining forbidden situations and allowing everything else.

In this thesis, we are concerned with imperative process models. We believe that they are the dominating modeling paradigm. However, we include a discussion about incorporating compliance requirements with declarative process models in Chapter 9.

Imperative process models can also be classified into graph based modeling languages, e.g., BPMN [1], EPCs [67] and UML ADs [100] and text based languages that depend on process algebra [17], e.g, π-Calculus [90]. For the same reason of dominance, we are concerned with graph based modeling languages. We discuss how the contributions of the thesis can fit with text-based modeling languages in Chapter 9.

4.1.1 Process Modeling Concepts

Currently, there is a number of graph-based business process modeling languages, e.g., BPMN [1], EPCs [67] and UML ADs [100]. Despite their variance in expressiveness and modeling notations, they all share a core set of concepts. All process modeling languages are capable of expressing concepts of activities, the individual steps that represent the added value of a business process. They are also capable of expressing execution ordering between activities, control flow. Sequential order, choice, and parallel threads are all control flow ordering that are expressible by these languages. Moreover, these languages provide support for the modeling of data flow in addition to that of control flow. By data flow, we mean the data elements and their values (states) that are needed as input for an activity, as a routing condition and/or as an output from an activity.

In this thesis, we focus on the core set of concepts; from which, business process models can be composed. Within graph-based process modeling languages, elements for modeling business processes are divided into two categories, nodes and edges respectively. Nodes, in turn, are divided into control objects, nodes that would correspond to active elements in a process, and data objects, passive data elements that are read or updated during the execution of a process. A control object is the general concept for activities, events, and control routing (gateways). Activities represent process steps, tasks, or a functions that are atomic and executable during a process instance *. Similarly, events are things/flags that are signaled during the process execution. Finally, control routing gives more expressiveness for the ordering of control objects other than sequencing. The XOR and AND control routing carry the respective execution semantics implied by their names that are largely accepted in the business process management community, formal execution semantics will be considered in subsequent sections.

Activities might occur multiple times within a process model, e.g., two nodes have the same label. However, it is of great importance for our work to distinguish these activities, especially when we discuss explanation of compliance violation in Chapter 6. Thus, we assume that each activity is assigned a unique identifier.

Data objects are used to represent data elements accessed during the process execution. In general, data elements can have domains that are infinite in nature, e.g., the domain of integer numbers. In this thesis, we use data objects to represent business objects handled throughout the process. A business objects has a finite set of states that represents an acceptable abstraction over underlying, possibly infinite, domain of values. Thus, at any point of the process execution, a data object can assume one state.

Edges are used to connect the nodes to create a process model. There are basically two types of edges, control flow and data flow edges. Control flow edges connect between control objects to model the ordering between them. On the other hand, data flow edges connect either data objects to control objects, to model the reading of data, or they connect control objects to data objects, to model the updating of data.

Process models described above, can be formalized as follows:

*For this thesis, we do not need to distinguish tasks from sub-processes

Definition 4.1. *[Business Process model]* A business process model *is a tuple* $\mathcal{P} = (\mathcal{A}, \mathcal{E}, \mathcal{D}, \mathcal{G}, \mathcal{S}, \mathcal{D}ata\mathcal{S}tate, \mathcal{CF}, \mathcal{DF}, \mathcal{CFC}, \mathcal{ID})$*where:*

- $\mathcal{A}$ *is a finite set of* activities*,*
- $\mathcal{E}$ *is a finite set of* events *where* $\mathcal{E}_s$*,* $\mathcal{E}_e$ *represent the* start *and* end *events, respectively. Other events are called intermediate events,*
- $\mathcal{D}$ *is a finite set of* data objects*,*
- $\mathcal{G}$ *is a finite set of* control flow routing gateways*. This set is further subdivided into* $\mathcal{G}_\mathcal{X}$ *and* $\mathcal{G}_\mathcal{A}$*, representing XOR and AND gateways respectively.* $\mathcal{G}_\mathcal{X} \cap \mathcal{G}_\mathcal{A} = \emptyset$*,*
- $\mathcal{S}$ *is a finite set of* data states*,*
- $\mathcal{D}ata\mathcal{S}tate \subseteq \mathcal{D} \times \mathcal{S}$ *is the* relation between data objects and states*,*
- $\mathcal{CF} \subseteq (\mathcal{A} \cup \mathcal{G} \cup (\mathcal{E} \setminus \mathcal{E}_e)) \times (\mathcal{A} \cup \mathcal{G} \cup (\mathcal{E} \setminus \mathcal{E}_s)$ *is the* control flow *relation,*
- $\mathcal{DF} \subseteq (\mathcal{D}ata\mathcal{S}tate \times \mathcal{A}) \cup (\mathcal{A} \times \mathcal{D}ata\mathcal{S}tate)$ *is the* data flow *relation between activities, data objects, and states,*
- $\mathcal{CFC} : \mathcal{CF} \rightarrow 2^{2^{\mathcal{D}ata\mathcal{S}tate}}$ is a function associating control flow edges with data conditions. *An empty condition is interpreted as* $true$*. Each data condition is assumed to be in a disjunctive normal form,*
- $\mathcal{ID} : \mathcal{A} \cup \mathcal{E} \cup \mathcal{G} \rightarrow \mathbb{N}$ is a function that assigns a unique ID for each node.

Definition 4.1 allows to assign explicit data conditions to control flow edges. However, to describe complex conditions, we require the data condition to be in a disjunctive normal form. For instance, a condition that refers to two data objects “Order” and “Bill” can be $(((Order, confirmed) \wedge (Bill, issued)) \vee ((Order, cancelled)))$.

Definition 4.1 allows process models of arbitrary structure. In order to correctly reason about the behavior of these models, we need to consider well formed process models. We use the notation $in(x)$ to denote the set of incoming control flow edges for a control node x, a control flow node is either an activity, an event or a gateway. Similarly, $out(x)$ denotes the set of outgoing control flow edges for a control node x. Moreover, the set $state_d = \{s : (d, s) \in \mathcal{D}ata\mathcal{S}tate\}$ identifies for each data object d the set of data states it can assume.

Definition 4.2. *[Well formed process model] A business process model* $\mathcal{P} = (\mathcal{A}, \mathcal{E}, \mathcal{D}, \mathcal{G}, \mathcal{S}, \mathcal{D}ata\mathcal{S}tate, \mathcal{CF}, \mathcal{DF}, \mathcal{CFC}, \mathcal{ID})$ *is* well formed *if and only if:*

- There is exactly one start even*t,* $|\mathcal{E}_s| = 1$,
- There is exactly one end event, $|\mathcal{E}_e| = 1$,
- A start event has no incoming edges, $\forall e \in \mathcal{E}_s : |in(e)| = 0$,

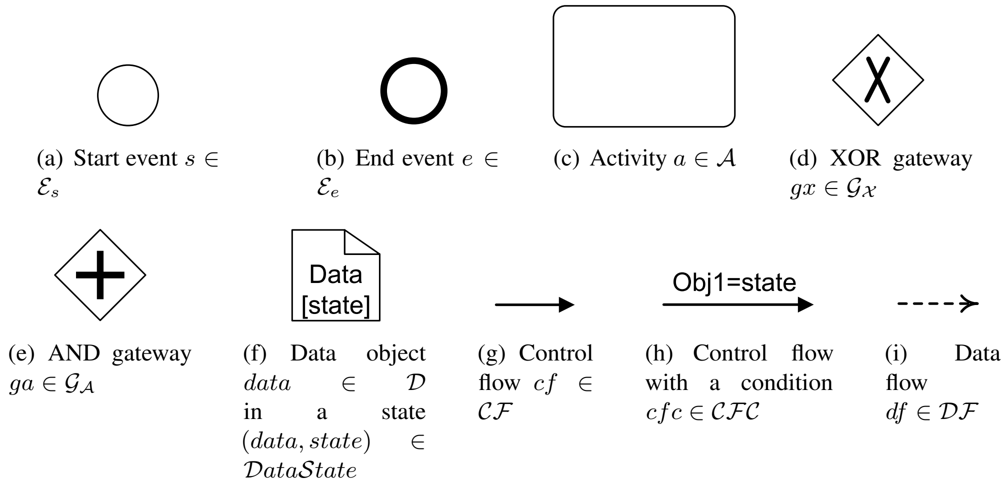
- An end event has no outgoing edges, $\forall e \in \mathcal{E}_e : |out(e)| = 0$,
- Any activity, non-start event has exactly one incoming control flow edge, $\forall n \in \mathcal{A} \cup (\mathcal{E} \setminus \mathcal{E}_s) : |in(n)| = 1$,
- Any activity, non-end event has exactly one outgoing control flow edge, $\forall n \in \mathcal{A} \cup (\mathcal{E} \setminus \mathcal{E}_e) : |out(n)| = 1$,
- Explicit splits and joins, $\forall g \in \mathcal{G} : (|in(g)| = 1 \land |out(g)| > 1) \lor (|in(g)| > 1 \land |out(g)| = 1)$, no mixed gateway,
- Each data object has an initial state. $\forall d \in \mathcal{D} \exists i \in state_d$ *where* i *indicates the initial state of the data object,*

In addition to the properties discussed in Definition 4.2, we require process models to be correct. By correctness we mean lack of deadlocks and livelocks either due to control flow modeling error [3, 122] or data flow modeling errors [120, 8, 134]. Moreover, no dead activities are present in the model due to never-satisfied data conditions. This is a reasonable assumption since compliance requirements are domain-specific correctness criteria while deadlock and livelock freedom is a domain-independent one.

We define $\mathbb{P}$ to be the set of all process models, the repository of process models (cf. Section 2.3). Every process model $p \in \mathbb{P}$ is well formed.

4.1.2 BPMN as a Process Modeling Language

Business Process Modeling and Notation (BPMN) [1] is the defacto standard for modeling business processes. We use the BPMN notation to visually represent the modeling concepts described above. Figure 4.1 describes the concrete syntax of these concepts in the BPMN notation.



(a) Start event $s \in \mathcal{E}_s$

(b) End event $e \in \mathcal{E}_e$

(c) Activity $a \in \mathcal{A}$

(d) XOR gateway $gx \in \mathcal{G}_\mathcal{X}$

(e) AND gateway $ga \in \mathcal{G}_\mathcal{A}$

(f) Data object $data \in \mathcal{D}$ in a state $(data, state) \in \mathcal{D}ata\mathcal{S}tate$

(g) Control flow $cf \in \mathcal{CF}$

(h) Control flow with a condition $cfc \in \mathcal{CFC}$

(i) Data flow $df \in \mathcal{DF}$

Figure 4.1: Process modeling constructs expressed in BPMN notation

4.1.3 Execution Semantics

In order verify properties of the process behavior, formal execution semantics must be assigned to the different modeling concepts of a process model. In literature, Petri nets [113] have been widely used to formalize execution semantics of control flow aspects of different process modeling languages [143, 137, 31], to name just a few . Moreover, there are approaches to formalize data access semantics in process models [120, 133, 8].

In this thesis, we use the formalization of control flow introduced in [31]. Also, we use the data access semantics we developed in [8]. The advantage is that both approaches use place/transitions Petri nets as the formal background.

Definition 4.3 (Petri net)**.** *A* Petri net *is a tuple* $PN = [P, T, F, m_0]$ *where* P *and* T *are finite disjoint sets of* places *and non-empty set of* transitions*, respectively,* $F \subseteq (P \times T) \cup (T \times P)$ *is a* flow relation*, and* $m_0 : P \rightarrow \mathbb{N}$ *is an* initial marking.

For a node $x \in P \cup T$, we define ${}^{\bullet}x = \{y \mid (y, x) \in F\}$ and $x^{\bullet} = \{y \mid (x, y) \in F\}$. A transition t is *enabled* by a marking m (denoted $m \xrightarrow{t}$) if $m(p) > 0$ for all $p \in {}^{\bullet}t$. An enabled transition can *fire* in m (denoted $m \xrightarrow{t} m'$), resulting in a successor marking m' with $m'(p) = m(p) + 1$ for $p \in t^{\bullet} \setminus {}^{\bullet}t$, $m'(p) = m(p) + 1$ for $p \in {}^{\bullet}t \setminus t^{\bullet}$, and $m'(p) = m(p)$ otherwise. Moreover, we define a function $TID : T \rightarrow \mathbb{N}$ that assigns each transition an identifying number. This function plays an important role for the mapping of process models to Petri nets.

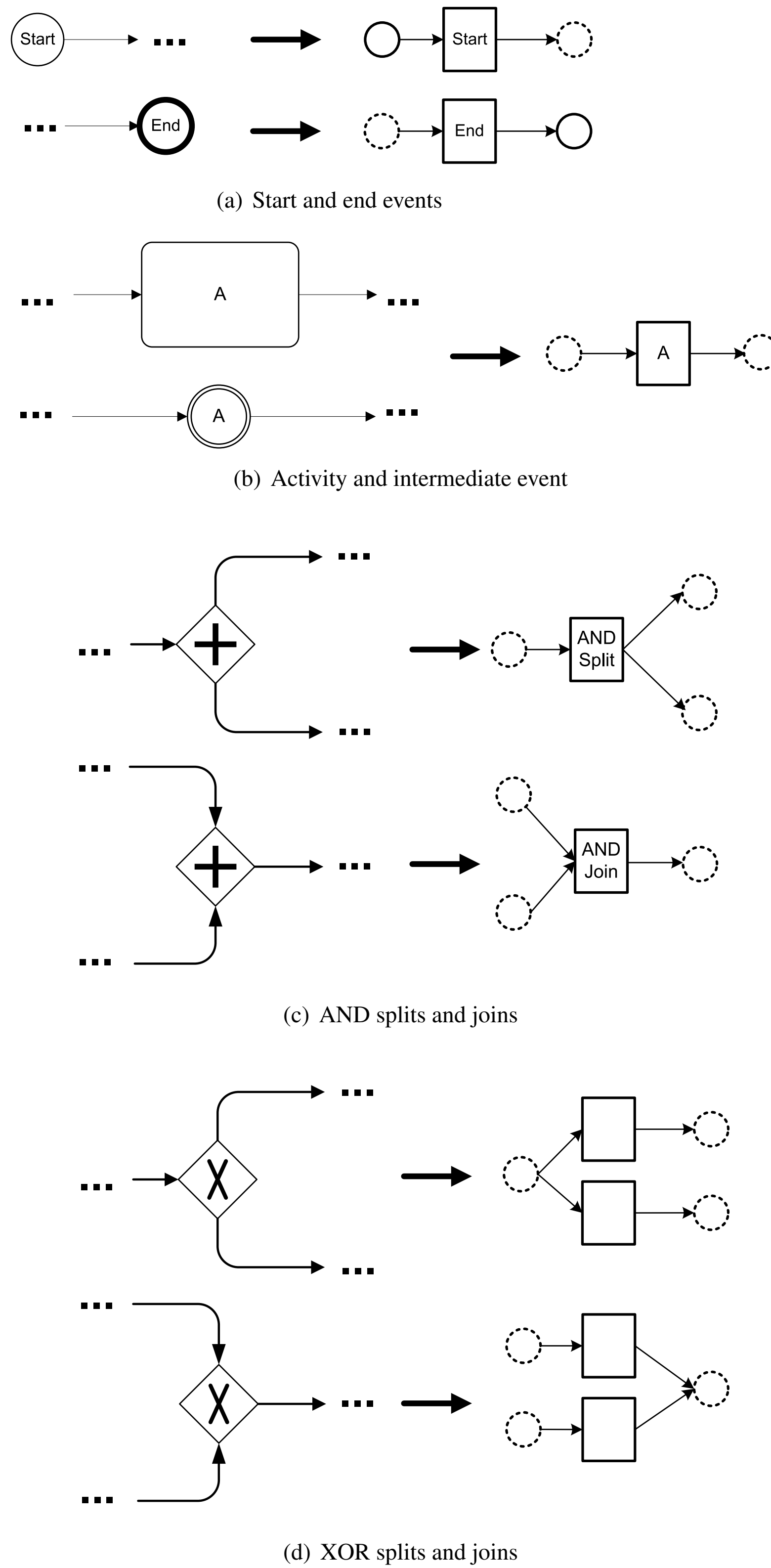
4.1.3.1 Control Flow Formalization

In this section we describe the mapping [31] of the business process control flow to place/transition Petri nets [113]. According to Definition 4.1 and Definition 4.2, we can have the following mapping patterns as shown in Figure 4.2.

Each event or activity is mapped to a single transition t and two places p_i, p_o where $p_i \in {}^{\bullet}t$ and $p_o \in t^{\bullet}$. The difference is that the input place for the start event transition has no incoming edges, denoted by the solid border of the place. Similarly, the output place of the end event transition has no outgoing edges. Other types of events and activities can share their input/output places with other preceding/succeeding nodes respectively.

An AND split gateway is mapped to a transition t, an input place p_i and a set of output places $\{p_{o1}, p_{o2}, \ldots, p_{on}\}$ according to the number of outgoing edges of the split. Both input/output places are shared with preceding/succeeding nodes respectively. On the other hand, an AND join is mapped to a transition t, a set of input places $\{p_{i1}, p_{i2}, \ldots\ p_{in}\}$ and a single output place p_o.

An XOR split is mapped to a shared place p_s, a set of transitions $\{t_1, t_2, \ldots, t_n\}$ according to the number of outgoing edges of the XOR split and a set of places $\{p_{o1}, p_{o2}, \ldots, p_{on}\}$ also according to the number of outgoing edges of XOR split. The place p_s is shared among the set of transitions. Place p_{oi} is the output place of transition t_i. An XOR join is mapped to a shared place p_j, a set of transitions $\{t_1, t_2, \ldots, t_n\}$ and a



(a) Start and end events

(b) Activity and intermediate event

(c) AND splits and joins

(d) XOR splits and joins

Figure 4.2: Control flow execution semantics by mapping to Petri nets, adapted from [31]

set of places $\{p_{i1}, p_{i2}, \ldots, p_{in}\}$. Each transition t_x and place p_{ix} represent an input edge of the XOR join.

To correlate transitions to their original nodes in the process model, we assign the same identifier for the transition as the node, i.e., $TID(t_n) = ID(n)$ where $n \in \mathcal{A} \cup \mathcal{E} \cup \mathcal{G} \wedge t_n \in T$.

4.1.3.2 Data Flow Formalization

First of all, we assume a single copy of each data object that is handled within the process. This single copy is assumed to exist from the moment of process instantiation. At instantiation, the data object has the specific initial state $initial$. Here, we can establish the analogy between a computer program and business process; where all variables are statically allocated at the instantiation of the program. Moreover, each variable is in a known initial value. Multiple data object shapes with the same label are considered to refer to the same data object.

On the other hand, each data object is in a certain state at any time during the execution of the process, e.g., an order is created, processed, confirmed, cancelled, etc. This data object changes its state only through the execution of activities. It can be specified which state a data object must be in before an activity can start (precondition) and which state a data object will be in after having completed an activity (effect). This is represented via data flow. A directed data flow from a data object to an activity symbolizes a precondition and a data flow leaving an activity towards a data object symbolizes an effect.

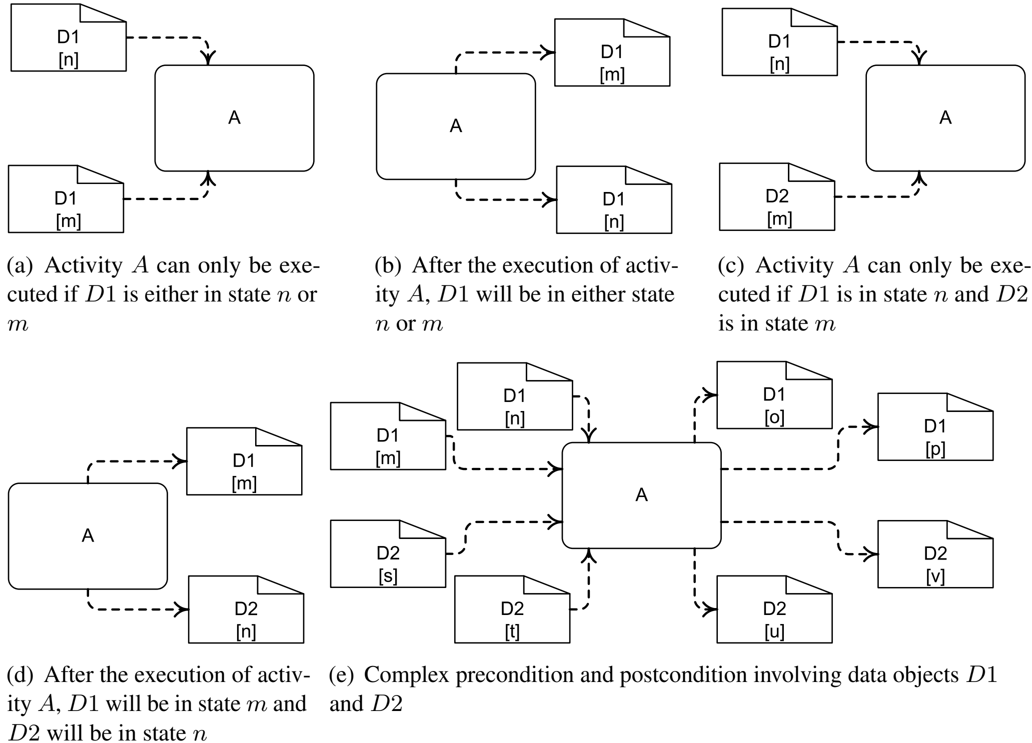
While often it is required that a data object is in exactly one state before being able to execute an activity, it might also be allowed that the data object is in any state of a set of states. This is symbolized through multiple data flow edges targeting an activity and that each originate in a data object shape referring to the same data object but in different states (cf. Figure 4.3(a))[†]. An analogous representation applies for alternative effects (cf. Figure 4.3(b)).

If multiple data objects are required as input (as in Figure 4.3(c)), then it is interpreted as a precondition that data object $D1$ is in state n *and* data object $D2$ is in state m. The same principle is applied in case of outputting multiple data objects Figure 4.3(d). Figure 4.3(e) describes the union of the other situations when an activity has complex pre/postconditions on data objects. In that specific case, activity A requires that data object $D1$ to be in *either* state n or m *and* data object $D2$ to be in *either* state s or t. After activity A completes, it changes the state of data object $D1$ to *either* o or p *and* changes the state of data object $D2$ to *either* state u or v.

Another interpretation for multiple data objects input/output could be the disjunction. I.e., an activity requires either $D1$ *or* $D2$ in order to execute. Although this might be the case in some situations, our interpretation is also possible. More modeling concepts are needed to let the modeler express his intent, e.g., by inputting of $D1$ and $D2$ to

[†] It is also possible to represent the different input states as single shape for data object $D1$ and the states are [m],[n]

AND/XOR join and then inputting that to the activity. However, employing any of the interpretations will not affect the contribution of this thesis.



(a) Activity A can only be executed if $D1$ is either in state n or m

(b) After the execution of activity A, $D1$ will be in either state n or m

(c) Activity A can only be executed if $D1$ is in state n and $D2$ is in state m

(d) After the execution of activity A, $D1$ will be in state m and $D2$ will be in state n

(e) Complex precondition and postcondition involving data objects $D1$ and $D2$

Figure 4.3: Assumptions about data access semantics

In order to reason about the change of data values through the process execution, formal data access semantics must be given to data objects and their reading/updating by activities. We assign this formal semantics by mapping [8] data objects states and access by activities to place/transition Petri nets as well. Fig. 4.4 illustrates the mapping. Each data object is mapped to a set of places. Each place represents one of the states the data object can be in. Activities with preconditions or effects are modeled as transitions. Depending on the kind of preconditions and effects, an activity can be represented by one or a set of transitions in the data flow model. Arcs connect these places with transitions, again depending on the preconditions and effects.

The simplest case is represented as in Figure 4.4(a). Here, an activity A reads a data object in $state\ 1$ and changes it to $state\ n$. This is represented as consuming a token from place *[Data object, state1]* and producing a token on *[Data object, staten]*. The case in Figure 4.4(b) represents a generalization of the previous case where an activity can have a complex precondition and can update the state of the data object to only one of the output states. In this case, activity A is represented with multiple transitions to reflect all possible combinations between precondition and effect states for the data object. The case in Figure 4.4(c), executing activity A does not have any effect on the data objects.

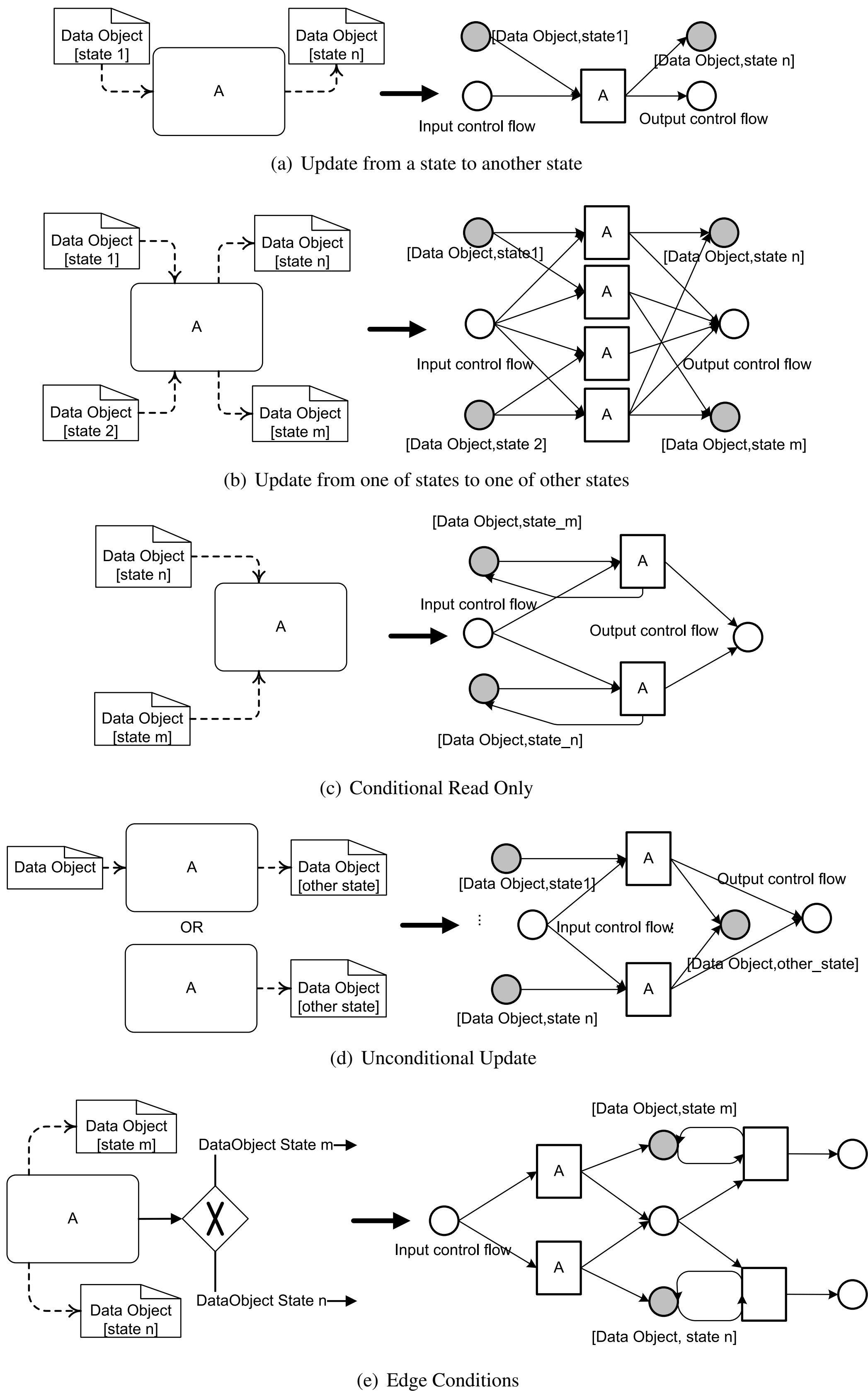


Figure 4.4: Data access semantics by mapping to Petri nets

However, it requires the data object to be in one of the specified states. This is modeled using a bi-flow in the Petri net: The transition consumes and produces a token from the same place. Notice that activity A might also be represented by many transitions if a data object is expected to be in one of many states.

The case in Figure 4.4(d) displays that the data object is changed to a certain state (*other_state*) when executing activity A. Multiple transitions are used as the data object can be in a number of previous states. For each previous state a transition models the change to the new state. In this case, it does not make any difference if the data object is used as input (without constraint on the state) or not at all. Finally, the case in Figure 4.4(e) illustrates how data object states can also be used in branching conditions.

4.1.3.3 Formalizing Process Model to Petri Net Mapping

So far, we have shown how the control flow as well as the data flow of a well formed process model, see Definition 4.2, can be mapped into a place transition Petri net. However, in order to reason about the behavior of the net, and so about the behavior of the process, we have to determine the initial marking of the net. In our case, the start event of the process determines the initiation of a process instance control flow. On the other hand, for data aspects, we assume that at the very beginning of the process each data object is in an $initial$ state. Thus, the initial marking of the net is determined by putting a token in the input place of transition t_{start} corresponding to the start event. And for each data object $initial$ state place a token is put.

Definition 4.4 formalizes the generation of a Petri net from a given well-formed process model. The formalization consolidates the different mappings provided in Figures 4.2,4.4. The set of places in the resulting Petri net consists of a place for each data state $s \in \mathcal{S}$ in the given process model. In addition, control flow places are added for start events, control flow between nodes in the model, end events respectively.

Activities that require neither data input nor data output are represented with a single transition each. On the other hand, those having either input or output data requirements are represented with multiple transitions. Each of the transitions represents one of the possible input/output conditions for the activity.

Connectivity between places and transitions comes from the control and data flow dependency between nodes in the process model. For input data dependency, each data state place is connected as an input place for the corresponding activity transition. The same case for output data dependency, where transitions are connected as incoming transitions for the corresponding output data states.

Definition 4.4. *[Process model to Petri net mapping] Let* $\mathcal{P} = (\mathcal{A}, \mathcal{E}, \mathcal{D}, \mathcal{G}, \mathcal{S}, \mathcal{D}ata\mathcal{S}tate, \mathcal{CF}, \mathcal{DF}, \mathcal{CFC}, \mathcal{ID})$ *be a well formed process model.*

The Petri Net (P,T,F,m_0) resulting from mapping is:

$P = \{p_{(d,s)} | (d,s) \in \mathcal{D}ata\mathcal{S}tate\} \cup$ states of data objects

$\{p_s | s \in \mathcal{E}_s\} \cup$ start event

$\{p_e | e \in \mathcal{E}_e\} \cup$ end event

$\{p_{(x,y)} | (x,y) \in \mathcal{CF}\}$ control flow

$T = \{t_a | a \in (\mathcal{A} \cup \mathcal{E} \cup \mathcal{G}_{\mathcal{A}}) \wedge \nexists(_, a), (a, _)^{\ddagger} \in \mathcal{DF}\} \cup$

data-independent activities

$\{t_{(d,s,a,s')} | a \in \mathcal{A}, d \in \mathcal{D}, s, s' \in \mathcal{S} \wedge s \neq s' \wedge ((d,s),a),(a,(d,s')) \in \mathcal{DF}\} \cup$

activities that change data objects states

$\{t_{(d,s,a)} | a \in \mathcal{A}, d \in \mathcal{D}, s \in \mathcal{S} \wedge ((d,s),a) \in \mathcal{DF} \wedge \nexists(a,(d,_)) \in \mathcal{DF}\} \cup$

read only activities

$\{t_{(x,y)} | x \in \mathcal{G}_{\mathcal{X}} \wedge (x,y) \in \mathcal{CF} \wedge |in(x)| = 1 \wedge CFC((x,y)) = true\} \cup$

branches of XOR splits without data conditions

$\{t_{(x,y,cond)} | a \in \mathcal{G}_{\mathcal{X}}, \wedge (x,y) \in \mathcal{CF} \wedge |in(x)| = 1 \wedge cond \ \in CFC((x,y))\} \cup$

branches of XOR splits with data conditions

$\{t_{(x,y)} | y \in \mathcal{G}_{\mathcal{X}} \wedge (x,y) \in \mathcal{CF} \wedge |out(x)| = 1\}$

branches of XOR joins

$F = \{(p_s, t_s) | s \in \mathcal{E}_s \wedge p_s \in P \wedge t_s \in T\} \cup$

start event

$\{(t_e, p_e) | e \in \mathcal{E}_e \wedge p_e \in P \wedge t_e \in T\} \cup$

end event

$\{(t_x, p_{(x,y)}) | (x,y) \in \mathcal{CF}\} \cup$

control flow

$\{(p_{(x,y)}, t_y) | (x,y) \in \mathcal{CF}\} \cup$

control flow

$\{(p_{(d,s)}, t_{(d,s,a)}), (t_{(d,s,a)}, p_{(d,s)}) | (d,s) \in \mathcal{D}ata\mathcal{S}tate, a \in \mathcal{A},$
$((d,s),a) \in \mathcal{DF}, \wedge \nexists(a,(d,_)) \in \mathcal{DF}, p_{(d,s)} \in P, t_{(d,s,a)} \in T\} \cup$

‡We use (a,_) to indicate that a is related to some element. Similarly is (_,a)

read only activities

$\{(p_{(d,s)}, t_{(d,s,a,s')}), (t_{(d,s,a,s')}, p_{(d,s')})|(d,s),(d,s') \in \mathcal{D}ata\mathcal{S}tate, a \in \mathcal{A},$
$((d,s),a),(a,(d,s')) \in \mathcal{DF}, p_{(d,s)}, p_{(d,s')} \in P, t_{(d,s,a,s')} \in T\} \cup$

activities that update data object state

$\{(p_{(d,s)}, t_{(x,y,cond)}), (t_{(x,y,cond)}, p_{(d,s)})|(d,s) \in cond, cond \in \mathcal{CFC}((x,y)), x \in \mathcal{G}_{\mathcal{X}} \wedge$
$|in(x)| = 1\}$

XOR split edges conditions

The initial marking m_0 *is determined as* $\forall p \in P \wedge \bullet p = \emptyset\ m_0(p) = 1$

control flow initial marking

$\forall p \in P$ *where* p *is an initial state data place* $m_0(p) = 1$

data flow initial marking

In this section we discussed business process modeling elements and their execution semantics. In the next two sections, we introduce the concepts process models query language BPM-Q.

4.2 BPM-Q

The Business Process Model Query (BPM-Q) language was designed to help business process designers access repositories of graph-based business process models [7, 117]. Thus, the language supports querying on the core concepts discussed in the previous section. Moreover, the language introduces a set of new *abstraction* concepts.

A BPM-Q model is called a query. In addition to the core business process modeling concepts, BPM-Q introduces new concepts, as will be discussed later. A query declaratively describes a structural connectivity that must be satisfied by a process model. To answer a query, it is matched to the process models in the repository. The matching determines the process (sub) graph that is the *refinement* of the query, details of matching will be discussed later.

BPM-Q was applied to detect modeling errors [11, 75], e.g., deadlocks. Also, it was applied to find semantically similar process models [10].

In this thesis, we use BPM-Q concepts to model compliance requirements. Thus, we discuss the subset of BPM-Q concepts that are useful to the compliance use case, interested readers can refer to [7, 117] for mode details.

In the following subsections we introduce a subset of the new concepts in BPM-Q useful for the compliance use case. Also, we describe the steps of matching a query to a process model.

4.2.1 BPM-Q Concepts

In addition to the core concepts of process modeling, activities, events, control flow, data flow, etc. BPM-Q has two major new concepts (constructs) that will be used for modeling compliance requirements, the *path edge* and the *anonymous activity*. Both concepts are means to provide abstraction over process model details. This matches the abstract nature of compliance requirements as will be shown in Chapter 5.

- **Path edges:** a path edge connecting two nodes in a query represents an abstraction over whatever control nodes could be in between in the process model. Moreover, a path edge has an *exclude* property. When the *exclude* property is set to some node(s), the evaluation of the path edge succeeds only if there are sequences of nodes between the source and destination nodes to whom *excluded* node(s) do not belong.

- **Anonymous activities:** Are meant to abstract from activities. For instance, a query could be “what activities read/update the insurance claim data object?”. Since, the user does not know that activity, he/she can start its label with the ’@’ symbol to declare it as an anonymous activity.

4.2.2 Matching Queries to Processes

A BPM-Q query is matched to a process via a set of refinements to the query. With each refinement nodes/edges in a query are replaced with nodes/(nodes,edges) of the matching process model. This operation is repeated until either a node/edge cannot be resolved, or all nodes/edges have been resolved. The match to the query, its final refinement, is a sub-graph of the process model.

Next, we formalize the notion of a BPM-Q query and the notion of matching a query to a process. We define a BPM-Q query as follows.

Definition 4.5. *[BPM-Q query]* a BPM-Q query *is a tuple* $\mathcal{Q} = (\mathcal{A}_\mathcal{Q}, \mathcal{E}_\mathcal{Q}, \mathcal{D}_\mathcal{Q}, \mathcal{G}_\mathcal{Q}, \mathcal{S}_\mathcal{Q}, \mathcal{D}ata\mathcal{S}tate_\mathcal{Q}, \mathcal{P}_\mathcal{Q}, \mathcal{DF}_\mathcal{Q}, isAnonymous, \mathcal{X}, \mathcal{ID}_\mathcal{Q})$ *where:*

- $\mathcal{A}_\mathcal{Q}$ *is a finite set of* activities *in a query,*

- $\mathcal{E}_\mathcal{Q}$ *is a finite set of* events *in a query.* $\mathcal{E}_\mathcal{Q} = \mathcal{E}_{\mathcal{Q}_s} \cup \mathcal{E}_{\mathcal{Q}_e}$ *where* $\mathcal{E}_{\mathcal{Q}_s}$ *is the set of* start events *and* $\mathcal{E}_{\mathcal{Q}_e}$ *is the set of* end events *in a query,* $\mathcal{E}_{\mathcal{Q}_s} \cap \mathcal{E}_{\mathcal{Q}_e} = \emptyset$*,*

- $\mathcal{D}_\mathcal{Q}$ *is a finite set of* data objects *in a query,*

- $\mathcal{G}_\mathcal{Q}$ *is a finite set of* control flow routing gateways*. This set is further subdivided into* $\mathcal{G}_{\mathcal{Q}\mathcal{X}}$ *and* $\mathcal{G}_{\mathcal{Q}\mathcal{A}}$*, representing XOR and AND gateways respectively.* $\mathcal{G}_{\mathcal{Q}\mathcal{X}} \cap \mathcal{G}_{\mathcal{Q}\mathcal{A}} = \emptyset$*,*

- $\mathcal{S}_\mathcal{Q}$ *is a finite set of* states(values) *data objects assume within a query,*

- $\mathcal{D}ataState_{\mathcal{Q}} \subseteq \mathcal{D}_{\mathcal{Q}} \times \mathcal{S}_{\mathcal{Q}}$ *is the* data state *relation in a query,*
- $\mathcal{P}_{\mathcal{Q}} \subseteq (\mathcal{A}_{\mathcal{Q}} \cup \mathcal{E}_{\mathcal{Q}}) \times (\mathcal{A}_{\mathcal{Q}} \cup \mathcal{E}_{\mathcal{Q}})$ *is the* path *relation between activity and event nodes in a query,*
- $\mathcal{DF}_{\mathcal{Q}} \subseteq (\mathcal{D}ataState_{\mathcal{Q}} \times \mathcal{A}_{\mathcal{Q}}) \cup (\mathcal{A}_{\mathcal{Q}} \times \mathcal{D}ataState_{\mathcal{Q}})$ *is the* data flow *relation between activities and data objects with states,*
- $isAnonymous : \mathcal{A}_{\mathcal{Q}} \rightarrow \{true, false\}$ *determines whether* activities *in a query are anonymous,*
- $\mathcal{X} : P_{Q} \rightarrow 2^{\mathcal{A}_{\mathcal{Q}} \cup \mathcal{E}_{\mathcal{Q}}}$ *is the* exclude property for path edges,
- $\mathcal{ID}_{\mathcal{Q}} : \mathcal{A}_{\mathcal{Q}} \cup \mathcal{E}_{\mathcal{Q}} \cup \mathcal{G}_{\mathcal{Q}} \rightarrow \mathbb{N}$ is a function that assigns an identifier to each control flow node in the query.

We assume the set $\mathbb{Q}$ to be the universal set of queries. Since BPM-Q is designed to match queries to process definitions in a repository, it is necessary to identify a candidate set of process models that might have the chance to provide a match to the query, rather than scanning the whole repository [128]. To achieve this, we assume a common ontology $\mathbb{TAG}$ is used to annotate both processes [130] and queries. The function $annotate : \mathbb{P} \cup \mathbb{Q} \rightarrow 2^{\mathbb{TAG}}$ assigns tags to processes and queries.

Definition 4.6. *[Candidate match]* a process model
$\mathcal{P} = (\mathcal{A}, \mathcal{E}, \mathcal{D}, \mathcal{G}, \mathcal{S}, \mathcal{D}ataState, \mathcal{CF}, \mathcal{DF}, \mathcal{CFC}, \mathcal{ID})$ is said to be relevant to a query
$\mathcal{Q} = (\mathcal{A}_{\mathcal{Q}}, \mathcal{E}_{\mathcal{Q}}, \mathcal{D}_{\mathcal{Q}}, \mathcal{G}_{\mathcal{Q}}, \mathcal{S}_{\mathcal{Q}}, \mathcal{D}ataState_{\mathcal{Q}}, \mathcal{P}_{\mathcal{Q}}, \mathcal{DF}_{\mathcal{Q}}, isAnonymous, \mathcal{X}, \mathcal{ID}_{\mathcal{Q}})$
if $annotate(\mathcal{P}) \cap annotate(\mathcal{Q}) \neq \emptyset$*. Thus, the set* $\mathbb{P}_{\mathcal{Q}} = \{p \in \mathbb{P} : p$ *is relevant to* $\mathcal{Q}\}$ *represents the candidate match processes to a query.*

After identifying the candidate match set of processes, the query processor continues by doing the following steps in sequence for each of the candidate match processes. If any of the steps fails, the processor drops the investigated process. The steps are detailed as follows:

1. Resolving non anonymous activities.
2. Resolving data objects.
3. Resolve gateway nodes
4. Resolving events.
5. Resolving anonymous activities.
6. Substituting path edges.

We start with an unrefined query. This is a query whose nodes are all assigned the value 0 as an identifier. With each of the steps above, some nodes are resolved. Resolution is done by binding a node in the query to a node in the process model. This binding occurs simply by assigning the node in the query the same identifier of the node in the process model.

To help describe the resolution of activities and data object nodes, we assume a function $label$. $label$ is used to find activity and data object nodes in a process that can be matched to those of a query. It is possible that an activity node in a query has more than one match activity node in a process, in case a process has multiple occurrences of the same activity. For each possible resolution node n_p of a query node n_q, the query processor generates a new copy of the query where n_q is assigned the same identifier as n_p. The new identifier of n_q is propagated to all its control flow, data flow, or paths relations. If for any node n_q in a query we cannot find a resolution n_p, the matching of a query to a process is dropped. The resolution of events is simple. Each event node in the query has to have a counterpart in the candidate process model, this is guaranteed in the case of well formed process models cf. Definition 4.2.

Resolving anonymous activities is achieved by finding an activity in the candidate process model for which all relations of the anonymous activity in the query are satisfied. For instance, if some anonymous activity '@A' has a data flow edge to some data object 'D' in the query, activity 'M' in the candidate process model is a resolution to '@A' *iff* there is a data flow edge to 'D' in the process. In case that an anonymous activity in a query has no data flow relationships, the query processor tries all available activities in the process model. Each time an anonymous activity is resolved, the query processor generates a new copy of the query where the anonymous activity is replaced with the resolvant activity. Also, this replacement is propagated to all relations of the anonymous activity. Gateways are resolved in the same way.

Note that our description about matching a query to a process is limited to the subset we use in this thesis. More details can be found in [7, 117]

As mentioned above, the query processor generates a new copy of the query when it binds a node. A query that has a subset of its nodes bound is called a *partially* refined query. On the other hand, when all resolution steps are completed, all queries are *refined*, i.e., all nodes are bound. We can notice that the resolution steps are of combinatorial nature, as the query processor has to find all possible bindings to every node. However, it is possible to start the query processing by a *partially* or completely *refined* query, we will use this feature heavily in Chapter 6 when we talk about compliance violation explanation.

Finally, for each refined query the query processor substitutes path edges in a query with the set of nodes and edges in the process model that constitute the process sub-graph in which the *target* node of the edge is reachable from its *source*.

To help explain how path edges are substituted, we define the transitive closure over the control flow relation.

Definition 4.7. *[Transitive closure over control flow]*
Let $\mathcal{P} = (\mathcal{A}, \mathcal{E}, \mathcal{D}, \mathcal{G}, \mathcal{S}, \mathcal{D}ata\mathcal{S}tate, \mathcal{CF}, \mathcal{DF}, \mathcal{CFC}, \mathcal{ID})$ *be a well-formed process model.* The transitive closure over the control flow relation is defined as $\mathcal{CF}^*$. *For each node* $n \in \mathcal{A} \cup \mathcal{G} \cup \mathcal{E}$*, we define the set of* reachable *nodes from* n *as* $[\![n]\!]_{\mathcal{CF}^*} = \{m : (n, m) \in \mathcal{CF}^*\}$.

Based on the above definition, we can define the reachable nodes from some node x under different conditions. For instance $[\![x]\!]_{(\mathcal{CF} \setminus \{(x,y)\})^*}$ defines the reachable node from node x where we removed the edge (x, y) from the control flow relation $\mathcal{CF}$.

With the help of Definition 4.7, we can explain how path edges are evaluated. A path edge $(source, target, EXC)$ evaluates to a sub-graph in which all nodes are 1) reachable from $source$ node 2) $target$ node is reachable from every node 3) every node $e \in EXC$ is not in the sub-graph and every node solely reachable by e is not in the sub-graph either 4) edges between nodes in the sub-graph are those between these nodes in the process model.

Definition 4.8. *[Subgraph] Let* $\mathcal{N} = \mathcal{A} \cup \mathcal{E} \cup \mathcal{G}$ *be the set of control flow nodes in a process model* $\mathcal{P} = (\mathcal{A}, \mathcal{E}, \mathcal{D}, \mathcal{G}, \mathcal{S}, \mathcal{D}ata\mathcal{S}tate, \mathcal{CF}, \mathcal{DF}, \mathcal{CFC}, \mathcal{ID})$*. Let* EXC *be the set of nodes to be excluded from a path evaluation. Let* $\mathcal{CF}_{EXC} = \mathcal{CF} \setminus \{(x, y) : y \in EXC \wedge (x, y) \in \mathcal{CF}\}$ *be the control flow relation without edges incoming to excluded nodes.* A function subgraph $(a, b, EXC) := (N', E')$*, where* $a, b \in \mathcal{N}$*, and* EXC *is the set of nodes to be excluded, constructs the process sub-graph where:*

- $N' = \{n : n \in [\![a]\!]_{\mathcal{CF}^*_{EXC}} \wedge b \in [\![n]\!]_{\mathcal{CF}^*_{EXC}}\}$,
- $E' = \{(x, y) : x, y \in N' \wedge (x, y) \in \mathcal{CF}\}$.

If $N' = \emptyset$, the subgraph is empty.

A candidate process model is said to match the refined query if it satisfies all path edges and data flow dependencies as in Definition 4.9.

Definition 4.9. *[Matching a query to a business process model]*
A candidate process model $\mathcal{P} = (\mathcal{A}, \mathcal{E}, \mathcal{D}, \mathcal{G}, \mathcal{S}, \mathcal{D}ata\mathcal{S}tate, \mathcal{CF}, \mathcal{DF}, \mathcal{CFC}, \mathcal{ID})$ matches *a query*
$\mathcal{Q} = (\mathcal{A}_\mathcal{Q}, \mathcal{E}_\mathcal{Q}, \mathcal{D}_\mathcal{Q}, \mathcal{G}_\mathcal{Q}, \mathcal{S}_\mathcal{Q}, \mathcal{D}ata\mathcal{S}tate_\mathcal{Q}, \mathcal{P}_\mathcal{Q}, \mathcal{DF}_\mathcal{Q}, isAnonymous, \mathcal{X}, \mathcal{ID}_\mathcal{Q})$ *if :*

- $\forall a_q \in \mathcal{A}_\mathcal{Q} \wedge isAnonymous(a) = false \; \exists a_p \in \mathcal{A} : label(a_q) = label(a_p)$. *Each* non-anonymous activity *must have at least one activity in the process model with the same label,*
- $\forall e \in \mathcal{E}_\mathcal{Q} \exists e' \in \mathcal{E}$ *where* e' *is a resolvant for* e*. Each* event *in the query must have a counterpart event in the process,*
- $\forall d_q \in \mathcal{D}_\mathcal{Q} \exists d_p \in \mathcal{D} : label(d_q) = label(d_p)$. *Each* data object *in the query must have a data object in the process with the same label,*

- $\mathcal{D}ataState_{\mathcal{Q}} \subseteq \mathcal{D}ataState$. *Data object* states *in a query must be in the process model,*

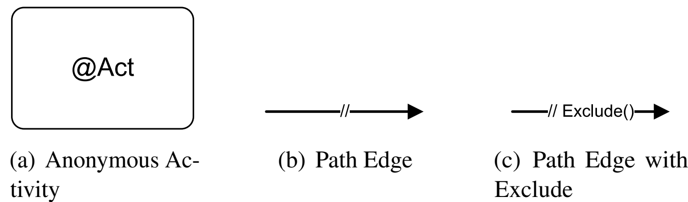
- $\{((a, m), (n, a) : (a, m), (n, a) \in \mathcal{DF}_{\mathcal{Q}}, a \in \mathcal{A}_{\mathcal{Q}} \wedge isAnonymous(a) = false\} \subseteq \mathcal{DF}$. *All* data flow *relations for a non-anonymous activity in a query must be included in the process model,*

- $\forall a_q \in \mathcal{A}_{\mathcal{Q}} \wedge isAnonymous(a) = true \exists a_p \in \mathcal{A} : ((a_q, m) \in \mathcal{DF}_{\mathcal{Q}} \rightarrow (a_p, m) \in \mathcal{DF}) \wedge ((n, a_q) \in \mathcal{DF}_{\mathcal{Q}} \rightarrow (n, a_p) \in \mathcal{DF}$. *All* data flow *edges must be satisfied for anonymous activities,*

- $\forall (n, m) \in \mathcal{P}_{\mathcal{Q}} \rightarrow subgraph(n, m, \mathcal{X}((n, m))) \neq \emptyset$. a process must have a non empty subgraph matching paths in the query.

4.3 BPMN-Q

BPM-Q by itself lacks a concrete syntax (visual notation). Thus, to be used, concepts have to be given a visual notation. Figure 4.5 summarizes the symbols used to represent the new concepts in the BPM-Q in the BPMN [1] syntax, introducing the visual query language BPMN-Q. However, other modeling languages can be used as well. The overhead will be to define visualization for the new concepts of BPM-Q in the selected language, e.g., EPC-Q, etc.



(a) Anonymous Activity (b) Path Edge (c) Path Edge with Exclude

Figure 4.5: BPM-Q concepts expressed in BPMN notation (BPMN-Q)

After describing the concepts of BPM-Q and its visual representation as BPMN-Q, we illustrate, through an example, the approach of query matching to a process. Figure 4.6 shows a process model where activity B occurs twice. The number attached to each node is its identifier.

The query in Figure 4.7 looks for activities B, D with a path in between. To match that query to the process model in Figure 4.6, the query processor starts with resolving nodes in the query. Based on the resolution approach described above, we end up with two *refined* queries as shown in Figure 4.8.

There are two refinements because activity B has two occurrences in the model while activity D has only one occurrence. In the first refinement of Figure 4.8(a) activity B in the original query was bound to activity B with ID 4 in the process model. In the second

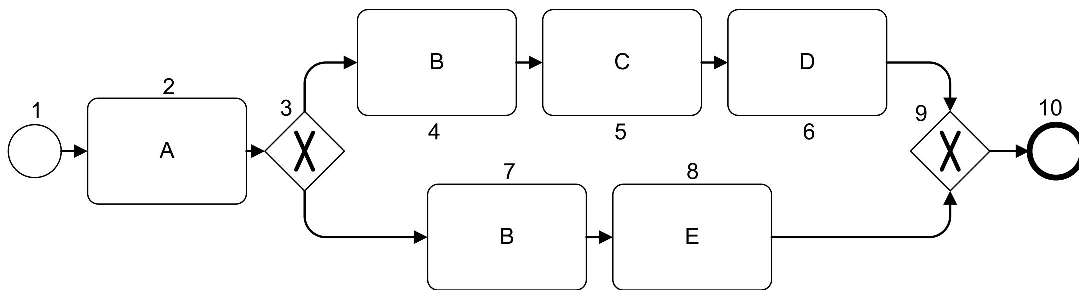


Figure 4.6: A process model with repeated activities

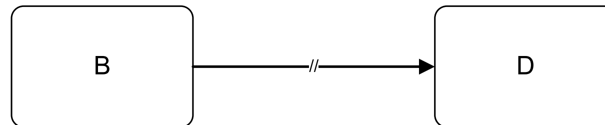
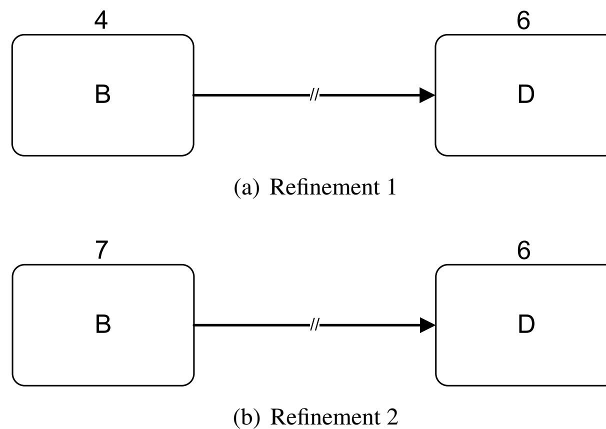


Figure 4.7: A BPMN-Q query

refinement of Figure 4.8(b) activity B of the original query was bound to activity B in the process model with ID 7. In both refinements D was bound to activity D with ID 6.



(a) Refinement 1

(b) Refinement 2

Figure 4.8: Refinements of BPMN-Q in Figure 4.7

The final step to complete the matching is to substitute the path edge in the query with its matching subgraph in the process. For the first refinement, the path edge is matched to activity C with ID 5, the two control flow edges from $B4$ to $C5$ and from $C5$ to $D6$. However, the path edge in the second refinement does not find a match. There is no sequence of nodes for which $B7$ and $D6$ belong. Thus, the query processor drops the second refinement. At the end, the query processor returns the match to the first refinement as the match to the original query.

4.4 The Business Knowledge

As stated earlier, compliance management is a knowledge-intensive problem. The more knowledge can be collected and encoded; the more automation we can achieve. Domain knowledge reflects the way a business provides its added value. Moreover, it could be seen as the unified view of the domain among the participants, e.g., stakeholders, practitioners, business analysts, compliance officers, etc. The knowledge is a global process-independent description of the business activities and objects. In this setting, each business process can be seen as a composition of business activities and business objects to achieve a set of business goals. Thus, it is rational to argue that each process model should be consistent with the domain knowledge.

Each business activity provides a value by its own [151]. For instance, an activity *Pay back claim to customer by bank transfer* provides the value of transferring the amount of money claimed by a customer as its *effect*. However, for such an activity to perform its job, there has to be a valid claim as an input. Moreover, having another activity *Pay back claim by a cheque*, we could derive, based on the domain knowledge, that it is not possible for the two payment activities to be *both* executed in the same process instance.

To make the concept more familiar, one might think of the notion of service registries [98] in service oriented architecture as a sub-concept for the domain knowledge. There, each service is described independently in terms of its inputs and outputs on a syntactical level. In a business domain knowledge, business activities are the counterpart of services in a registry.

Aspects are the means to describe the various relations within a business domain. Besides activities, *aspects* might consider data objects used in the business domain. Thus, the domain knowledge does not specify all valid execution scenarios globally, but defines constraints locally for each single activity. Therefore, it is independent of any concrete process model.

Definition 4.10. *[Domain knowledge]*
The domain knowledge $\mathbb{C}$ is a 7-tuple
$(N_{act}, A, T, asptype, con_{t\in T}, pre_{t\in T}, post_{t\in T})$*, where:*

- N_{act} is the set activities used within the business domain to accomplish atomic actions that add value within a business process,
- A is the set of objects, which define model aspects. It describes the artifacts used within the business domain,
- T is the set of aspect types,
- $asptype : A \to T$ is the function assigning aspect types to objects,
- $con_t : (N_{act}\times N_{act})\cup(\{a : \forall a \in A, asptype(a) = t\}\times\{a : \forall a \in A, asptype(a) = t\})$ is the contradiction relation between activities and objects of the same aspect type.

- $pre_{t \in T} \subseteq N_{act} \times 2^{\{a: \forall a \in A, asptype(a)=t\}}$ is the relation defining the prerequisites of activity execution in terms of objects.

- $post_{t \in T} \subseteq N_{act} \times 2^{\{a: \forall a \in A, asptype(a)=t\}}$ is the relation defining the result of activity execution in terms of objects. Postconditions of an activity can be divided into *positive and negative*, i.e.,$post_{t \in T} = post^{+}_{t \in T} \cup post^{-}_{t \in T}$

One *aspect* of a context is a tuple $(N_{act}, A_t, t, asptype, con_t, pre_t, post_t)$, where $t \in T \wedge A_t = \{a : a \in A, asptype(a) = t\}$.

Definition 4.10 captures aspects with three elements: sets A, T and function $asptype$. Set A is the set of objects, describing the business environment from a certain perspective, e.g., dependencies of activities on data objects or their semantic annotations. Set T consists of the object types; an example is $T = \{activity, data, semantic\ Annotation\}$. Function $asptype$ specifies a type (element of set T) for an element of A. Typification of objects allows distinguishing aspects, e.g., distinguishing data flow from semantic description of a process.

The three basic relations are pre, $post$, and con. They respectively describe preconditions, postconditions, and contradiction relations. The pre relation describes what objects with different (types) aspects that are required for a certain activity in order to execute it. Similarly, the $post$ relation specifies what are the effects of executing a certain activity. For each activity there might be different sets of pre/post conditions to resemble the notion of alternation. Taking data as an aspect to describe such relations, pre_{data} would describe the precondition of each activity in terms of data elements. The con relation describes contradictions between activities as well as other aspects in the context. If two activities are known to be contradicting, at most one is allowed to appear in any process instance. With these three relations, pre, $post$, and con we are able to express the dependency and mutual exclusion between business activities and business objects which are necessary for automating compliance checking [82], explanation and resolution.

We assume process models to be consistent with the domain knowledge: at least one precondition for any activity in the model must be satisfied. Also, activities are assumed to produce the effect, post condition(s), as described in the context. Moreover, for any two contradicting activities, there must be no chance to execute both of them in a single instance.

We assume the business knowledge to be present. We depend on this knowledge to correctly model compliance requirements, explain violations, and to give automated support, if possible, to resolve violations, if any.

4.5 Model Checking

Model checking [24] is a static verification technique where a behavioral system is investigated to satisfy a property by exhaustively searching its state space. In case that a

property is not satisfied, a model checker generates a counter-example. A counter-example is a finite sequence of system states that violates the property.

Despite the various syntax of input languages for model checkers to describe systems, they all adhere to a structure called the Kripke structure. A Kripke structure is a special type of finite state machines where nodes represent states of the system, edges represent possible transitions between states. Moreover, each state is associated with a set of propositions that are true in that state. Over a Kripke structure, model checking algorithms can construct the state space in order to check the property. Kripke structures are used to describe systems with finite states and infinite transitions over these states.

The second input to the model checker, the property, is usually described in a temporal logic language. The following subsections will provide more details about generating a Kripke structure from a process model and about temporal logic languages respectively.

Definition 4.11. *[Kripke Structure] Let AP be a set of non-empty atomic propositions. A* Kripke structure *is a tuple $\mathcal{M} = (S, s_i, R, L)$ where:*

- *S is a finite set of* states*,*
- *s_i is a* dedicated initial state*,*
- *$R \subseteq S \times S$ is a* transition relation*, for which it holds that $\forall s \in S \exists s' \in S : (s, s') \in R$*
- *$L : S \rightarrow 2^{AP}$ is a* labeling function *associating a set of propositions with each state.*

Temporal Logic (TL) is an extension of classical proposition calculus that allows checking the truth value of propositions according to time. It is a variation of modal logic where new operators are used to allow expressing properties based on time. Temporal logic was first introduced by Arthur Prior in the sixties of the last century. It was further developed by computer scientists. Currently, there are two major variants of temporal logic linear temporal logic LTL [109] and computational tree logic CTL [35]. The expressiveness of the two logics is incomparable. We will briefly introduce both LTL, with both future and past time operators, and CTL as they are the foundation for formal compliance checking approaches contributed within this thesis.

4.5.1 Linear Temporal Logic

Linear temporal logic was first introduced by Pnueli [109]. In the original setting of LTL, temporal operators were concerned only with the future direction time. A major development was by the introduction of past time operators [160]. With LTL, the execution of a system, a process in our case, is seen as sequences, of states. For an LTL property to hold, it has to be satisfied by all execution sequences of the system.

Assuming a set of atomic propositions AP that describe facts about behavior of a system, LTL defines a set of temporal operators in addition to classical logical operators $(\neg, \vee, \wedge, \rightarrow)$

Definition 4.12. *[LTL Syntax] Let AP be a set of atomic propositions.* LTL formulas can be defined according to the grammar $\theta ::= \top|\bot|p|\neg\theta|\theta \wedge \theta|\theta \vee \theta|\theta \rightarrow \theta|\mathbf{F}\theta|\mathbf{G}\theta|\theta\mathbf{U}\theta|\mathbf{O}\theta|\theta\mathbf{S}\theta$*, where* $p \in AP$.

These operators are used to describe the truth value of propositions describing the behavior of a system over time. So, an execution of the system $\mathcal{M} = (S, s_i, R, L)$ can be seen as a trace, a sequence of states, $\sigma = s_0, s_1, \ldots$ where each state $s_i,\ i > 0$, is associated with a set of propositions $L(s_i)$ that are $true$ in that state. We say that proposition $p \in P$ is true at state s_i by $s_i \models p$.

Informally, the semantics of the different temporal operators are defined as follows

- $\mathbf{F}(\theta)$: in some future state of the system θ will hold.
- $\mathbf{G}(\theta)$: θ hold in the current state and in every future state.
- $\omega\mathbf{U}\theta$: ω is true in all states until the point (state) θ becomes true. There has to be a state where θ becomes true.
- $\mathbf{O}(\theta)$: $\mathbf{O}$ is the history-looking counterpart of the $\mathbf{F}$ operator. That is, at the state evaluating $\mathbf{O}(\theta)$, there must have been some previous state where θ was true.
- $\omega\mathbf{S}\theta$: similarly, $\mathbf{S}$ is the history-looking counter part of the $\mathbf{U}$ operator. That is, at the state evaluating $\omega\mathbf{S}\theta$, ω was true since the state where θ was true.

Definition 4.13 formally describes the semantics of the LTL operators.

Definition 4.13. *[LTL Semantics] Let $\sigma = s_0, s_1, \ldots$ be a word in Σ^* with $\Sigma = 2^P$. Let θ be an LTL formula.* Given a trace σ and a position i The relation $\sigma.i \models \theta$ is defined as*:*

- $\sigma.i \models \top$ *for any* i*,*
- $\sigma.i \not\models \bot$ *for any* i*,*
- $\sigma.i \models p$ *if* $p \in L(s_i)$*,*
- $\sigma.i \models \neg\theta_1$ *if* $\sigma.i \not\models \theta_1$*,*
- $\sigma.i \models \theta_1 \wedge \theta_2$ *if* $\sigma.i \models \theta_1$ *and* $\sigma.i \models \theta_2$*,*
- $\sigma.i \models \theta_1 \vee \theta_2$ *if* $\sigma.i \models \theta_1$ *or* $\sigma.i \models \theta_2$*,*
- $\sigma.i \models \theta_1 \rightarrow \theta_2$ *if* $\sigma.i \not\models \theta_1$ *or* $\sigma.i \models \theta_2$*,*
- $\sigma.i \models \mathbf{F}\theta$ *if* $\exists k \geq i : \sigma.k \models \theta$*,*

- $\sigma.i \models \mathbf{G}\theta$ *if* $\forall k \geq i : \sigma.k \models \theta$,
- $\sigma.i \models \theta_1 \mathbf{U} \theta_2$ *if* $\exists k \geq i : \sigma.k \models \theta_2$ *and* $\forall j, i \leq j < k, : \sigma.j \models \theta_1$,
- $\sigma.i \models \mathbf{O}\theta$ *if* $i \geq 1, \exists k < i : \sigma.k \models \theta$,
- $\sigma.i \models \mathbf{H}\theta$ *if* $i \geq 1, \forall 0 \leq k \leq i : \sigma.k \models \theta$,
- $\sigma.i \models \theta_1 \mathbf{S} \theta_2$ *if* $\exists k \leq i, \sigma.k \models \theta_2$ *and* $\forall k < j \leq i, \sigma.j \models \theta_1$.

If the behavior of a system $\mathcal{M}$ is nondeterministic, a set of sequences Ω represents the different possibilities of execution. For a system $\mathcal{M}$ to satisfy an LTL formula θ, written as $\mathcal{M} \models \theta$, all sequences of $\mathcal{M}$ have to satisfy θ.

Definition 4.14. *[LTL property satisfaction] Let Ω be the set of traces of a system $\mathcal{M}$.* $\mathcal{M}$ is said to satisfy an LTL formula θ, $\mathcal{M} \models \theta$ *if* $\forall \sigma \in \Omega \exists i : \sigma.i \models \theta$.

4.5.2 Computational Tree Logic

Computational tree logic (CTL) is another language to express properties for model checking. Unlike LTL, CTL is designed to handle nondeterminism in the behavior of a system $\mathcal{M}$. Thus, in addition to introducing temporal operators; it also introduces global and existential quantifiers.

Definition 4.15. *[CTL Syntax] Let AP be a set of atomic propositions.* CTL formulas can be defined according to the grammar $\theta ::= \top | \bot | p | \neg\theta | \theta \wedge \theta | \theta \vee \theta | \theta \rightarrow \theta | \mathbf{AF}\theta | \mathbf{EF}\theta | \mathbf{AG}\theta$ $| \mathbf{EG}\theta | \mathbf{A}[\theta \mathbf{U} \theta] | \mathbf{E}[\theta \mathbf{U} \theta]$, *where* $p \in AP$.

With respect to a system $\mathcal{M} = (S, s_i, R, L)$, for each state $s \in S$, there could be more than one subsequent state. Recursively, for each subsequent state there could be more than one subsequent state, etc. Thus, starting at any state $s \in S$ we could have multiple paths to go. CTL gives the possibility to express temporal formulas over *all* of these paths or *some* of these paths.

Definition 4.16. *[CTL Semantics] Let $\mathcal{M} = (S, s_i, R, L)$ be a transitions system. Let θ be a CTL formula and $s \in S$.* $\mathcal{M}, s \models \theta$ is defined as:

- $\mathcal{M}, s \models \top$,
- $\mathcal{M}, s \not\models \bot$,
- $\mathcal{M}, s \models p$ *if* $p \in L(s)$,
- $\mathcal{M}, s \models \neg\theta$ *if* $\mathcal{M}, s \not\models \theta$,
- $\mathcal{M}, s \models \theta_1 \wedge \theta_2$ *if* $\mathcal{M}, s \models \theta_1$ *and* $\mathcal{M}, s \models \theta_2$,
- $\mathcal{M}, s \models \theta_1 \vee \theta_2$ *if* $\mathcal{M}, s \models \theta_1$ *or* $\mathcal{M}, s \models \theta_2$,

- $\mathcal{M}, s \models \theta_1 \rightarrow \theta_2$ *if* $\mathcal{M}, s \not\models \theta_1$ *or* $\mathcal{M}, s \models \theta_2$,
- $\mathcal{M}, s \models \mathbf{AF}\theta$ *if* $\forall$ *path* $p = s, s_2, s_3, \ldots$ *there is a state* s_j *such that* $\mathcal{M}, s_j \models \theta$,
- $\mathcal{M}, s \models \mathbf{EF}\theta$ *if* $\exists$ *path* $p = s, s_2, s_3, \ldots$ *there is a state* s_j *such that* $\mathcal{M}, s_j \models \theta$,
- $\mathcal{M}, s \models \mathbf{AG}\theta$ *if* $\forall$ *path* $p = s, s_2, s_3, \ldots$ *for each state* s_j *on each* p $\mathcal{M}, s_j \models \theta$,
- $\mathcal{M}, s \models \mathbf{EG}\theta$ *if* $\exists$ *path* $p = s, s_2, s_3, \ldots$ *and for each state* s_j *on that path* $\mathcal{M}, s_j \models \theta$,
- $\mathcal{M}, s \models \mathbf{A}[\theta_1 \mathbf{U} \theta_2]$ *if* $\forall$ *path* $p = s, s_2, s_3, \ldots$ *there is a state* s_j *such that* $\mathcal{M}, s_j \models \theta_2$ *and* $\forall i < j, \mathcal{M}, s_i \models \theta_1$,
- $\mathcal{M}, s \models \mathbf{E}[\theta_1 \mathbf{U} \theta_2]$ *if* $\exists$ *path* $p = s, s_2, s_3, \ldots$ *there is a state* s_j *such that* $\mathcal{M}, s_j \models \theta_2$ *and* $\forall i < j, \mathcal{M}, s_i \models \theta_1$

CTL and LTL expressiveness is incomparable [146]. However, there is a common set of specifications that can be expressed in both LTL and CTL [85]. For this common space, CTL and LTL model checkers practically perform similarly [18].

While both LTL and CTL are capable of formalizing compliance requirements, as will be discussed in Chapter 5, CTL can formalize violations scenarios, as will be shown in Chapter 6, due to the ability to express *existential* formulas.

There are well-known logical equivalences in CTL[§]. We list some of them useful for the work presented in this thesis.

$$\mathbf{AG}\theta \equiv \neg\mathbf{EF}(\neg\theta) \tag{4.1}$$

$$\mathbf{AF}\theta \equiv \neg\mathbf{EG}(\neg\theta) \tag{4.2}$$

$$\mathbf{A}[\theta\ \mathbf{U}\Psi] \equiv \neg(\mathbf{F}[(\neg\Psi)\mathbf{U}\ \neg(\theta \vee \Psi)] \vee \mathbf{EG}(\neg\Psi)) \tag{4.3}$$

4.5.3 Temporal Logic Querying

Usually, model checking is an iterative process where the modeler gains more understanding of the system each time a property is violated. To enhance the understanding of system behavior and to gain more feedback, Chan [21] proposed to develop temporal logic queries (TLQ). In temporal logic queries, user puts a placeholder ?; the query solver has to find the strongest propositional formula p, formula composed of atomic propositions and classical logical connectors, that replaces the placeholder and make the resulting formula true. For instance, the query $\mathbf{AG}(?)$ simply asks about the model invariants. This mechanizes and speeds up the trial and error effort of analyzing a design [20]. To help the user focus more on specific properties, temporal logic queries

[§]These equivalences can be proved by contradiction in a straightforward manner

can be restricted(projected) on a specific set of propositions. For instance $\mathbf{AG}(?_{x,y})$ is restricted version of the previous query where result is a propositional formula based on x, y only. Model checking is a subproblem of temporal logic querying. In model checking, we ask only Boolean queries.

Temporal logic queries are also of great value to explain violations. Consider a CTL formula $\mathbf{AG}(x \wedge y \rightarrow \mathbf{AF}(z))$ that is checked against some system $\mathcal{M} = (S, s_i, R, L)$. In case of violation, model checkers generate counter examples. However, counter examples are not always useful. It simply finds a sequence of states that causes the violation. In our case, there is some state $s_j \in S$ where $x, y \in L(s_j)$ and there is no further state s_k reachable from s_j where $z \in L(s_k)$. Suppose that the user issues the following temporal logic query $\mathbf{AG}(? \rightarrow \mathbf{AF}(z))$. The solver returns $x \wedge y \wedge r$ as a value for $?$. In this case, the user got more useful feedback and learned more about the model.

Definition 4.17. *[Temporal Logic Query Checking] Let AP be a set of non-empty atomic propositions and $AP' \subset AP$. Let PF be the set of propositional formulas the can be made from atomic propositions in AP. Let $\mathcal{M}$ be a Kripke structure and let ϕ be a temporal logic query, both are defined over AP.* The query checking problem is to compute the set $\{\epsilon \in PF(AP') | \mathcal{M} \models \phi[\epsilon]\}$.

Definition 4.17 formally describes the problem of temporal logic query checking. The set AP' represents the atomic propositions relevant to the query. Thus, the query solver (checkers) finds the set of strongest propositional formulas ϵ that would make the Kripke structure $\mathcal{M}$ satisfy the temporal logic formula $\phi[\epsilon]$ obtained by substituting ϵ for the place holder in the query ϕ.

Generally, the temporal logic query solving can be reduced to several model checking problems [21, 20]. The worst case complexity for a TLQ solver is to model check $2^{2^{|AP'|}}$ formulas with length at most $|\phi| + O(2^{|AP'|})$. However, there were research on developing more efficient solvers [22].

In Chapter 6, we employ temporal logic querying techniques to help explain and provide useful feedback to the user. As will be discussed, when data related compliance rules are violated; temporal logic queries are issued to extract specific data conditions that caused the violation. Thus, the user focuses on parts of the process model that needs consideration. Also, we show how domain-specific knowledge can be used to simplify solving temporal queries to linear time model checking rather than its doubly exponential nature.

Chapter 5

Modeling and Checking Compliance Rules

The core objective of this thesis is to support automated checking of compliance rules against business process models at design-time. In this situation, control flow and data flow aspects are feasible to check. In many cases, it is necessary to ensure that certain activities *exist* in the process model. Also, execution ordering between activities must be verified. Moreover, execution ordering might be required to hold only under certain data conditions. Thus, the control flow, the data flow and the interplay between them need to be expressed in compliance rules.

Model checking [24] provides a formal approach for static verification with acceptable performance [154, 34]. However, an obstacle for directly employing the model checking approach is that the specification language for rules, i.e., temporal logic, is too technical to be used by business people. Moreover, it gives a very wide range of rules to be specified. However, certain patterns of rules might be sufficient to express compliance requirements [33, 39, 94].

To overcome this obstacle, we provide the users with a simple visual language to express compliance rules, BPMN-Q. As was discussed in Section 4.3, BPMN-Q is a visual language to query the structure of process models. It uses notations very similar to that used for designing process models. Also, BPMN-Q provides means for abstraction over process models details, e.g., path edges and anonymous activities. This matches the abstract and declarative nature of compliance rules, cf. $Req.$ 1.

We follow a pattern-based approach to express compliance rules [42, 33]. Each pattern is expressed visually in BPMN-Q. Thus, it helps the user specify rules without dealing with complexity of formal languages [9, 16, 14]. On the other hand, each pattern is mapped into a formula in temporal logic, enabling automated verification. Patterns are mapped into computation tree logic CTL [35] and past time linear temporal logic PLTL [160]. The purpose of providing these various mappings is to ensure independence

from a specific model checking software (cf. Spin [59] vs. Nusmv [23]). Meanwhile, we will benefit from certain mappings when discussing issues about consistency checking among rules.

The rest of the chapter is organized as follows. We discuss the different categories of compliance rules we address in Section 5.1. Section 5.2 explains how rules are modeled as BPMN-Q queries and what are their counterpart temporal logic formulas. In Section 5.3 we address the issue of consistency checking among related rules. Checking rules against processes is discussed in Section 5.4. In Section 5.5 we give an example case study for modeling and checking compliance that we extracted from guidelines for anti money laundering [25]. Section 5.6 summarizes the chapter and links to the next chapter.

5.1 Categories of Compliance Rules

Compliance rules can be divided into three categories a) Control flow rules b) Data flow rules c) Conditional control flow rules.

5.1.1 Control Flow Rules

Control flow rules focus on activities and their execution relationships. For instance, certain activities must be present or absent in process models. An example would be "Complaints must always be archived". Regarding to relationships, activities might be required to execute in a certain order or must be exclusive to each other.

Based on work in [33], we can further categorize rules according to the concept of a scope. A scope has a temporal duration that might be lower- or upper-bounded. A scope is either *global*, *before*, *after*, or *between*. Activities, and later on data conditions, determine the boundary of a scope. A *global*-scope is bounded by the start and end of the process respectively, the whole process. An *after*-scope is lower-bounded by some activity. A *before*-scope is upper-bounded with an activity. *Between*-scope is both lower- and upper-bounded with activities. Within a scope, an activity may be required to *exist/absent*. Depending on the scope execution ordering between two activities maybe either *precedence* (execution history) or *response* (execution future). Figure 5.1 describes the different scopes.

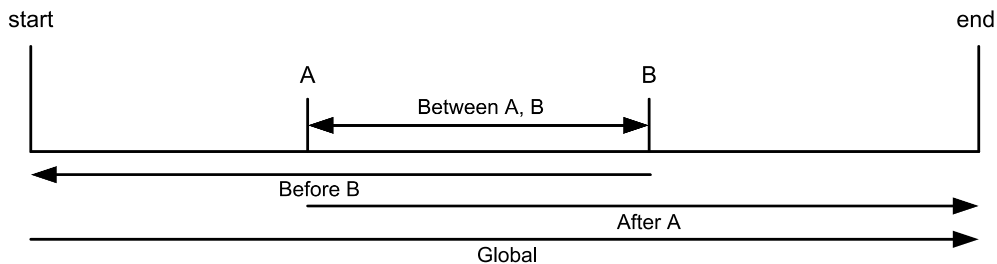


Figure 5.1: Scopes for compliance rules

5.1.2 Data Flow Rules

It might be necessary to check that certain data conditions must hold at a specific point in a process model, e.g., the point an activity is about to execute. An example would be at shipping time of goods payment must have been received. Also, to evaluate whether the process model always achieves the specified goals, such rules can be employed.

5.1.3 Conditional Control Flow Rules

Conditional control flow rules are variants of control flow rules, cf. 5.1.1 where the ordering is only required to hold under certain data conditions. Several data conditions might be required to hold in order to trigger the checking of such rules. Also, data conditions can play the role of a scope for the checking of the compliance rule.

5.2 Modeling Compliance Rules

We use BPMN-Q, the concrete syntax of BPM-Q using BPMN notation, to visually express compliance requirements as queries. As stated earlier, this helps the modeler abstract from the technical details of temporal logics. However, in order to express the different semantics of compliance requirements, we had to extend BPMN-Q elements.

5.2.1 Extensions of BPMN-Q

Originally, BPMN-Q was developed to express queries on the structure of process models, see Section 4.2. However, for compliance checking, we need to express queries not only on the structure of business processes but also on their behavior. Thus, we extended BPMN-Q by stereotyping some of its elements.

The path edge of BPMN-Q is a means to abstract over process details between two nodes. However, for reasoning about the behavior of the process between two nodes, path edge is not sufficient. For instance, to distinguish between the precedence and response patterns, discussed later, we stereotype the path edge with *«Precedes»* and *«Leads To»* to express the difference between precedence and response patterns respectively.

Data flow rules require to check whether specific data conditions hold at the point an activity is about to execute. Nevertheless, it is not necessary that these conditions are explicitly mentioned in the process model. On the other hand, some conditional rules, for instance, require that in case an activity A executes and produces a specific $data$ result, then activity B is required to occur. In this situation, an explicit data flow is targeted. To distinguish between explicit and implicit data flow, we extend the data flow edge with the *behavioral* stereotype.

We reflect these extensions on the formal definition of BPMN-Q queries by adding a new function $\mathcal{TYPE} : \mathcal{P}_Q \cup \mathcal{DF}_Q \rightarrow \{none, leadsto, precedes, behavioral\}$ where path and data flow edges can be annotated with either $none$, reflecting the structural paths and data flow edges, $leadsto$ to reflect the path edge looks for response between

its source and target nodes respectively, $precedes$ to reflect that the path edge looks for precedence between its source and target nodes respectively, or $behavioral$ to indicate that the data flow edge is for behavioral preconditions as will be discussed later. Thus, a behavioral BPMN-Q query is
$\mathcal{Q} = (\mathcal{A}_\mathcal{Q}, \mathcal{E}_\mathcal{Q}, \mathcal{D}_\mathcal{Q}, \mathcal{S}_\mathcal{Q}, \mathcal{D}ata\mathcal{S}tate_\mathcal{Q}, \mathcal{P}_\mathcal{Q}, \mathcal{DF}_\mathcal{Q}, isAnonymous, \mathcal{X}, \mathcal{TYPE})$.

5.2.2 Predicates Used in the Formalization

We use the temporal operators discussed in Section 4.5. In addition, to reason about the states of execution of activities and data states for data objects we use the following predicates:

- The predicate **state**$(dataObject, stateValue)$ describes the fact that a data object assumes a certain state.
- The predicates **ready**$(activity)$ and **executed**$(activity)$ state that a certain activity is ready to be executed or has already been executed, respectively.
- The predicates **start** and **end** to indicate the start and end of execution of the process respectively.

Usually an activity life cycle consists of several states, e.g., $initial$, $ready$, $running$, $executed$, with transitions indicating the possible changes among states. This detailed activity life cycle is adequate for a process execution engine. We follow a simple activity life cycle for the verification purpose. That is, an activity has two possible states $ready$ and $executed$, as described above, with one transition from $ready$ to $executed$. Also, we assume that process models are compliant with the data objects' life cycle [74]. That is, the data object state changes by the execution of activities respect the object life cycle.

5.2.3 Modeling Control Flow Rules

We introduce seven patterns describing the presence/absence of activities within a scope.

5.2.3.1 Global-scope Presence

A single activity might be required to execute in all process instances, e.g., in a shipment process the received packets must be inspected in every case. Thus, we call such pattern a global presence as shown in Figure 5.2.

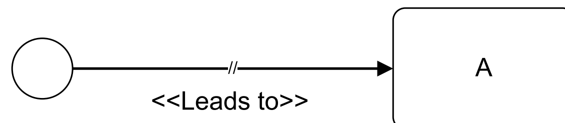


Figure 5.2: Global-scope presence pattern

The PLTL formula for global-scope presence is

$$\mathbf{G}(\textbf{start} \rightarrow \mathbf{F}(\textbf{executed}(A))) \tag{5.1}$$

The above formula is not the only way to model that pattern. Simply the sub formula $\mathbf{F}(\textbf{executed}(A))$ is equivalent. This is true since **start** holds only in one state through the execution of the process, process initiation, and there is no chance for it not to hold, as we consider well formed process models. However, the reason to model the pattern that way is to establish a generic template in which the change of pattern scope is achieved by changing formula's condition part, see after-scope presence pattern below.
The CTL formula for global-scope presence is

$$\mathbf{AG}(\textbf{start} \rightarrow \mathbf{AF}(\textbf{executed}(A))) \tag{5.2}$$

The two formulas above formally reflect that requirement of executing a certain activity in all instances. Since **start** is guaranteed to occur in any process, the process has to execute activity A in order to be compliant.

5.2.3.2 Global-scope Absence

It might be the case that certain activity must never execute, i.e., such an activity is absent from the process model. This is called the global absence as shown in Figure 5.3.

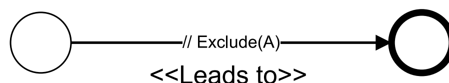


Figure 5.3: Global-scope absence pattern

The PLTL formula for global-scope absence is

$$\mathbf{G}(\textbf{start} \rightarrow \neg\textbf{executed}(A)\ \mathbf{U}\ \textbf{end}) \tag{5.3}$$

The CTL formula for global-scope absence is

$$\mathbf{AG}(\textbf{start} \rightarrow \mathbf{A}[\neg\textbf{executed}(A)\ \mathbf{U}\ \textbf{end}]) \tag{5.4}$$

Both formulas express the requirement of not executing an activity by assuring that there is no state of the process execution, from start to end, where A was executed.

5.2.3.3 Before-scope Presence

Before-scope presence, also known as precedence [33], rules require that the execution of an activity, B, is preceded by the execution of another activity, A. This requirement does not mean that B must be immediately preceded by A. There might be other activities in between. However, each time B is ready to execute, A must have had executed before. Figure 5.4 shows the BPMN-Q query to represent this pattern.

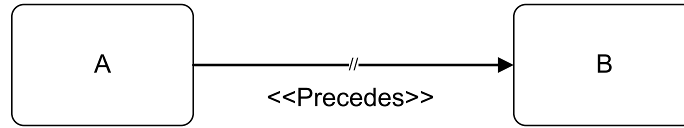


Figure 5.4: Precedence pattern

The PLTL formula for before-scope presence is

$$\mathbf{G}(\mathbf{ready}(B) \rightarrow \mathbf{O}(\mathbf{executed}(A))) \tag{5.5}$$

Formula 5.5 expresses the precedence relation by checking that each time activity B is about to execute $\mathbf{ready}(B)$, activity A must have been executed at least once before, the temporal operator $\mathbf{O}$. It is possible to represent this pattern in LTL without past operators. But with the use of past operators we can produce shorter specifications that are easier to understand.
The CTL formula for before-scope presence is

$$\neg\mathbf{E}[\neg\mathbf{executed}(A)\ \mathbf{U}\ \mathbf{ready}(B)] \tag{5.6}$$

CTL has no past operators. However, we can still describe the precedence relation by ensuring that there is no execution path where B can be ready to execute without having A executed before.

5.2.3.4 Before-scope Absence

An activity A might be required to be absent before the execution of another activity B. An example rule is “if you pay claim then there must not be any fraud investigation before", this type of rules is addressed in the BPMN-Q query in Figure 5.5.

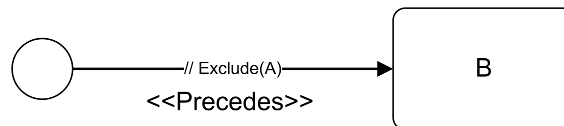


Figure 5.5: Before-scope absence pattern

The PLTL formula for before-scope absence is

$$\mathbf{G}(\mathbf{ready}(B) \rightarrow \neg\mathbf{executed}(A)\ \mathbf{S}\ \mathbf{start}) \tag{5.7}$$

The formula states that each time B is ready to execute it must be true that A was never executed since the start of the process.
The CTL formula for before-scope absence is

$$\neg\mathbf{EF}(\mathbf{start} \wedge \mathbf{EF}(\mathbf{executed}(A) \wedge \mathbf{EF}(\mathbf{ready}(B)))) \tag{5.8}$$

Here we express the fact that A must never be executed before B by stating that: there must be no execution path that goes from the start of the process to activity A and eventually to the activity B.

5.2.3.5 After-scope Presence

The after-scope presence, also known as response, pattern states that after the execution of an activity, A, activity B has to eventually be executed. For instance, for each claim received; a reply has to be sent to the customer. Within the scope lower bounded with A, B must be executed at some point. The representation of such a pattern in BPMN-Q is shown in Figure 5.6.

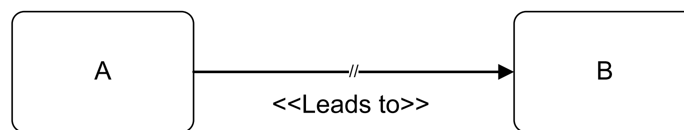


Figure 5.6: Response pattern

Formalization

The PLTL formula for this pattern is

$$\mathbf{G}(\mathbf{executed}(A) \rightarrow \mathbf{F}(\mathbf{executed}(B))) \tag{5.9}$$

The CTL formula is

$$\mathbf{AG}(\mathbf{executed}(A) \rightarrow \mathbf{AF}(\mathbf{executed}(B))) \tag{5.10}$$

Both formulas simply state that after activity A is executed; activity B must be eventually executed.

5.2.3.6 After-scope Absence

Similar to the before-scope absence pattern, it might be required to express that certain activities are forbidden to be executed after execution of another activity. For example, after confirming a purchase order; it cannot be cancelled. The query in Figure 5.7 shows a BPMN-Q query to capture this requirement.

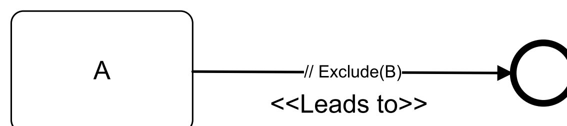


Figure 5.7: After-scope absence pattern

The PLTL formula for after-scope absence pattern is

$$\mathbf{G}(\mathbf{executed}(A) \rightarrow \neg \mathbf{executed}(B) \ \mathbf{U} \ \mathbf{end}) \tag{5.11}$$

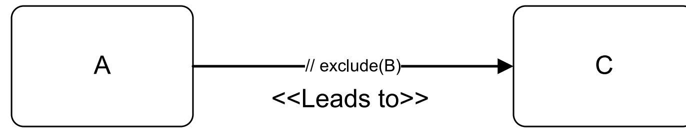
The CTL formula for that pattern is

$$\mathbf{AG}(\mathbf{executed}(A) \rightarrow \mathbf{A}[\neg \mathbf{executed}(B) \ \mathbf{U} \ \mathbf{end}]) \tag{5.12}$$

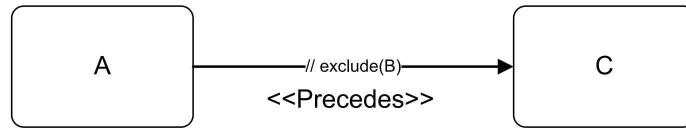
Both formulas state that after execution of activity A there is no chance to execute B in the remaining part of the process. Note the analogy between the above two formulas on the one hand and Formulas 5.3, 5.4, respectively, on the other hand.

5.2.3.7 Between-scope Absence

An example of this rule is “When a new order is received; it is not allowed to forward it to finance department until its bill of material is calculated". In this situation, between the execution of two activities, it is forbidden to execute certain other activities. However, the emphasis maybe put on one of the two boundary activities. This is reflected in Figure 5.8. In Figure 5.8(a), activity C is required to execute after A while in between there is no chance to execute B. We call this pattern response with absence. On the other hand, we might want to express precedence with absence. That is, we need to check that whenever C is executed; A must have been executed before without executing B in between. The BPMN-Q query for the latter case is shown in Figure 5.8(b).



(a) Response Pattern with Absence = Between-scope Absence Type I



(b) Precedence Pattern with Absence = Between-scope Absence Type II

Figure 5.8: Between-scope absence pattern

The PLTL formula for response with absence (between-scope absence type I) pattern is

$$\mathbf{G}(\mathbf{executed}(A) \rightarrow \neg \mathbf{executed}(B) \ \mathbf{U} \ \mathbf{executed}(C)) \tag{5.13}$$

The formula 5.13 is a variant of formula 5.11 where **end** is replaced with $\mathbf{executed}(C)$ to set the upper bound of a scope lower bounded with $\mathbf{executed}(A)$.
The PLTL formula for precedence with absence (between-scope absence type II) pattern is

$$\mathbf{G}(\mathbf{ready}(C) \rightarrow \neg \mathbf{executed}(B) \ \mathbf{S} \ \mathbf{executed}(A))) \tag{5.14}$$

Similarly, this formula is a variant of the formula 5.7.
The CTL formula for response with absence (between-scope absence type I) pattern is

$$\mathbf{AG}(\mathbf{executed}(A) \rightarrow \mathbf{A}[\neg\mathbf{executed}(B)\ \mathbf{U}\ \mathbf{executed}(C)]) \quad (5.15)$$

The CTL formula for precedence with absence (between-scope absence type II) pattern is

$$\begin{aligned}&\neg\mathbf{E}[\neg\mathbf{executed}(A)\ \mathbf{U}\ \mathbf{ready}(C)]\wedge\\&\neg\mathbf{EF}(\mathbf{executed}(A) \wedge \mathbf{EF}(\mathbf{executed}(B) \wedge \mathbf{EF}(\mathbf{ready}(C))))\end{aligned} \quad (5.16)$$

The above CTL formula captures the history-looking nature of the rule by stating that it is not possible to find any execution path in which A is not executed before C is reached. And, it also not possible to find an execution path where A executes and B afterwards and finally C is reached.

5.2.4 Modeling Data Flow Rules

To ensure compliance, it might be interesting to be sure that data manipulated by a certain business process will always have certain values at some point of the process execution, e.g., when an activity is about to execute. This, for instance, could be a way to measure the compliance of a process with an object life cycle [118].

To express data conditions, the data object notation is used in BPMN-Q to express condition of data values (states). In process models there must be explicit data flow edges from/to data elements/activities to express preconditions/effects respectively. However, it is possible in a compliance rule to check a data condition at the start of execution of an activity that is not explicitly mentioned in the process. To differentiate between explicit and implicit data conditions, we extended BPMN-Q with a new type of edge called behavioral data flow edges, cf. Section 5.2.1. This type of edges is distinguished by the *double* arrow head on the edge. Figure 5.9 shows data flow rules as BPMN-Q queries.

(a) Activity (b) End Event

Figure 5.9: Data flow pattern

If it is required to model a complex data condition involving several data objects, a data object node is added for each data object state, cf. Figure 4.3. Also, a behavioral data flow edge is added to connect the data object to the activity. Moreover, activity symbol can be replaced with an event, Figure 5.9(b), to indicate the point in the process where the condition has to hold.

Formalization

Recalling the discussion about data objects and their access semantics in Section 4.1.3.2, a data condition on a single data element d can generally be expressed as

$$dataCondition_d = \bigvee_{(d,s) \in \mathcal{D}ataState_{\mathcal{Q}}} \textbf{state}(d, s) \tag{5.17}$$

This means, we can express that the data object d is required to be in any of the sates relevant to the compliance rule. Having multiple behavioral data flow edges from different data objects is interpreted as the *conjunction* of conditions on the different data objects states. Thus, the general data condition form is shown in Formula 5.18, based on the single data object condition from Formula 5.17.

$$dataCondition = \bigwedge_{d \in \mathcal{D}_{\mathcal{Q}}} dataCondition_d \tag{5.18}$$

The PLTL formula for the data flow pattern is

$$\textbf{G}((\textbf{ready}(A)/\textbf{end}) \rightarrow dataCondition) \tag{5.19}$$

The CTL formula can be derived by adding $CTLA$ quantifiers

$$\textbf{AG}((\textbf{ready}(A)/\textbf{end}) \rightarrow dataCondition) \tag{5.20}$$

Both formulas check that in each state the activity is ready to execute, or the process terminates, the data condition has to hold. In the above and upcoming formulas, we use the notation (p/q) to indicate that either p or q can appear in the formula. We use this notation to reduce the number of formulas.

5.2.5 Modeling Conditional Control Flow Rules

By attaching data conditions to activities appearing in control flow rules, we can get several variants that refine the situations in which presence or absence of activities within scopes is required to hold. Also, in some cases, the compliance officer might be interested in the occurrence of the data condition without having to explicitly state which activity caused it. Figure 5.10 shows the possible data refinements for future-looking («Leads to »paths) control flow rules.

As shown in Figure 5.10, it is possible to attach data conditions to the source (antecedent) of the rule, or to the target (consequent) of the rule. Data refinement is represented as output data resulting from activities. Activities with "@" symbol are called

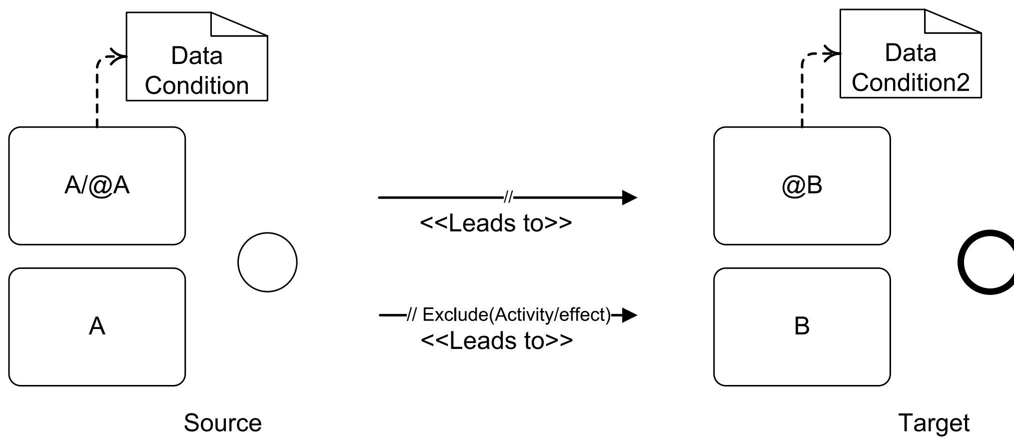


Figure 5.10: Possible data refinements for future-looking control flow rules

anonymous activities. With an anonymous activity associated with a data condition, the modeler abstracts from specific activities and focuses on the data condition of interest. Note also that for data conditions at the consequent (target) part of the rule, it is possible only to use anonymous rather than specific activities. However, not every combination of source, path edge, and target in Figure 5.10 makes sense. For instance, having a conditional activity at the source with the upper path edge and an end event as the target carries no useful property for compliance modeling. We discuss the combinations of interest in the rest of this section.

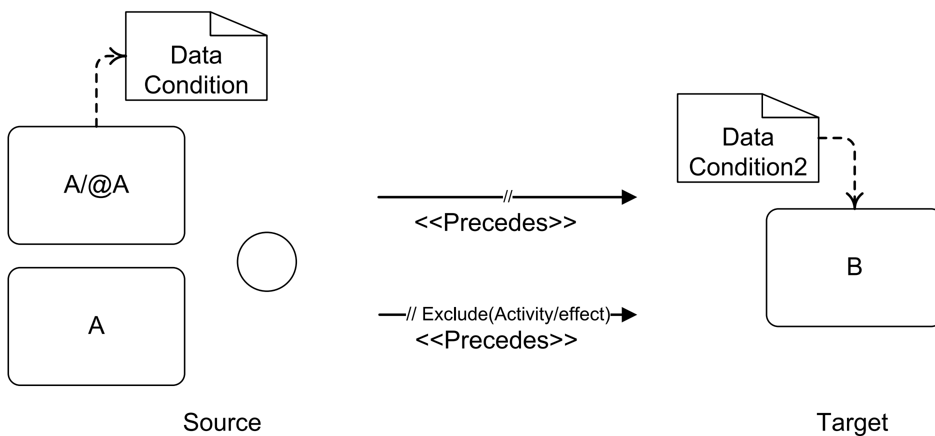


Figure 5.11: Possible data refinements for history-looking control flow rules

On the other hand, Figure 5.11 shows the possible data conditions refinement for history-looking («Precedes »paths). For the source (consequent) of the rule, it is possible to attach data conditions to both activities and anonymous activities. For the target (antecedent) of the rule, only input data conditions are allowed. Also meaningful combinations of sources and targets are discussed in the rest of this section.

In either cases future or history looking, absence patterns, where the exclude property refers to an activity, could as well refer to data conditions.

5.2.5.1 Effect Rules

In Section 5.2.3.1 we discussed the global-scope presence pattern where an activity is required to be executed in any process instance. Utilizing data conditions, we can derive the effect presence pattern as shown in Figure 5.12. By effect, we mean a data condition (result) that is required.

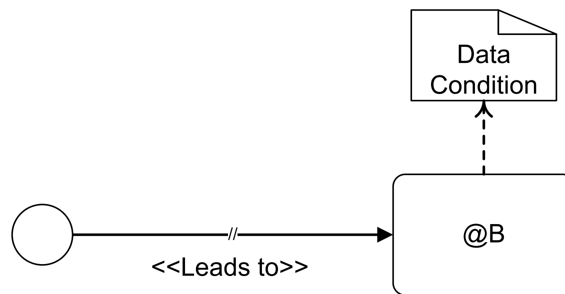


Figure 5.12: Global-scope effect presence pattern

To show the use for that new pattern, imagine a claim handling situation where a compliance rule requires that "for each claim there has to be a reply to the customer". One possibility is to model that rule as a "Reply Customer" activity is response to the "Start" event. However, checking this rule against the process model in Figure 5.13 fails. However, semantically, the model satisfies the requirement. On the other hand, if the rule is modeled using the *effect presence*, the requirement is better captured and the process is now compliant.

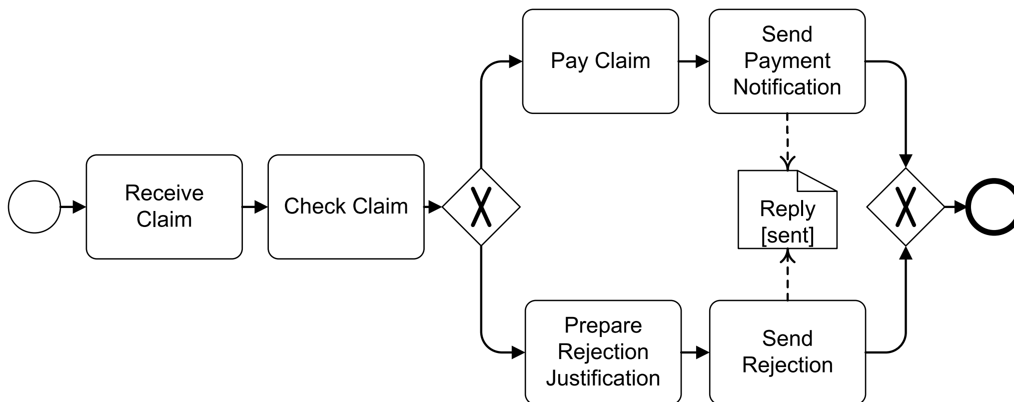


Figure 5.13: A claim handling process

The global-scope effect absence case could be modeled similar to the one in Figure 5.3. The difference is that the to-be-absent activity is replaced with the data condition (effect)

to be absent.

After-scope variants of this pattern can be obtained by replacing the start event in Figure 5.12 with an activity. Figure 5.14 shows the after-scope effect presence pattern. The after-scope effect absence could be derived in the same way as global-scope effect absence.

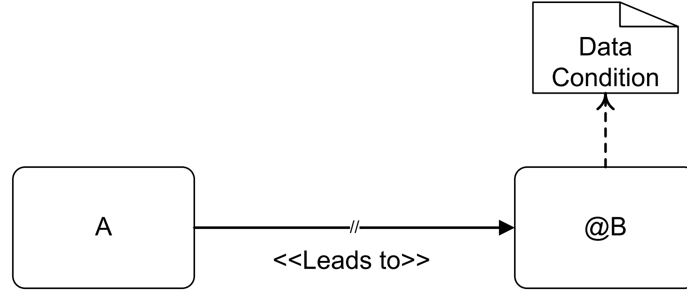


Figure 5.14: After-scope effect presence pattern

Formalization

As we discussed, the $effect$ is merely a data condition that is expected to occur. Formula 5.18 describes how the $effect$ is represented.

The PLTL formula for the global/after -scope effect presence is

$$\mathbf{G}((\textbf{start}/\textbf{executed}(A)) \rightarrow \mathbf{F}(effect)) \tag{5.21}$$

We can notice that the only difference between the above the formula and Formulas 5.1, 5.9 is that the target activity is replaced with the data effect.
The CTL formula for global/after -scope effect presence is

$$\mathbf{AG}((\textbf{start}/\textbf{executed}(A)) \rightarrow \mathbf{AF}(effect)) \tag{5.22}$$

Similarly, the PLTL formula for global/after -scope effect absence is

$$\mathbf{G}((\textbf{start}/\textbf{executed}(A)) \rightarrow \neg effect\ \mathbf{U}\ \textbf{end}) \tag{5.23}$$

The CTL formula for global/after -scope effect absence is

$$\mathbf{AG}((\textbf{start}/\textbf{executed}(A)) \rightarrow \mathbf{A}[\neg effect\ \mathbf{U}\ \textbf{end}]) \tag{5.24}$$

5.2.5.2 Conditional After-scope Presence

It might be required to execute a certain activity as a response to occurrence of a certain data condition. For instance, in a banking process, a rule could be “In case that a due diligence evaluation fails, the respondent back must be added to a black list”. Figure 5.15

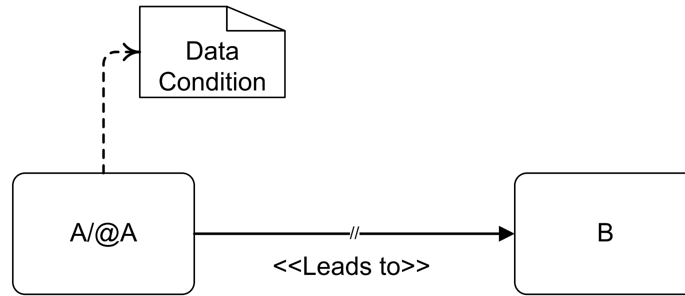


Figure 5.15: conditional after-scope presence pattern

shows a BPMN-Q query capturing conditional presence pattern. This pattern could also be referred to as *response to condition*.

To model the occurrence of the data condition; an (anonymous) activity $@A/A$ is associated to the data condition. The use of anonymous activity gives the modeler the means to abstract from the specific activity responsible for generating this condition. Activity B is required as a response for that condition.

This pattern can be considered as a refinement for the global-scope and after-scope presence patterns discussed in Sections 5.2.3.1, 5.2.3.5 respectively.

Similar to the discussion in Section 5.2.5.1 a data effect, i.e., an anonymous activity attached to an effect, could be modeled as a response to a data condition.

Formalization

The formalization of this pattern is not straightforward. The problem is that inappropriate modeling of data conditions might lead to false alarms during checking. For instance, consider the process excerpt in Figure 5.16. A rule could be that whenever activity A is executed and $\textbf{state}(D, bad)$ results as the effect of A, C must be executed thereafter. Formally, $\mathbf{G}(\textbf{executed}(A) \wedge \textbf{state}(D, bad) \rightarrow \mathbf{F}(\textbf{executed}(C)))$.

Model checking this formula with the process excerpt in Figure 5.16 results in a false alarm. That is, the model checker decides that the formula is not satisfied. To explain this, consider the case where A is executed for the first time and results in $\textbf{state}(D, bad)$. At this execution state, the model checker records that the condition for the rule is $true$. Afterwards, the process executes B and decides to take the yes branch of the XOR decision. At this point the process executes A again with the decision to set D to $good$. From that point, the process continues without executing C at all making the model checker to signal the violation.

The problem with the false alarm is that the formula is under-specified. The data condition is not well specified. In other words, we require that C executes only when D takes the value bad and never takes the value $good$ afterwards. We can say that bad and $good$ are contradicting data states. We assume the knowledge about contradicting data states to be available in the domain knowledge, cf. Section 4.4. For each data state s,

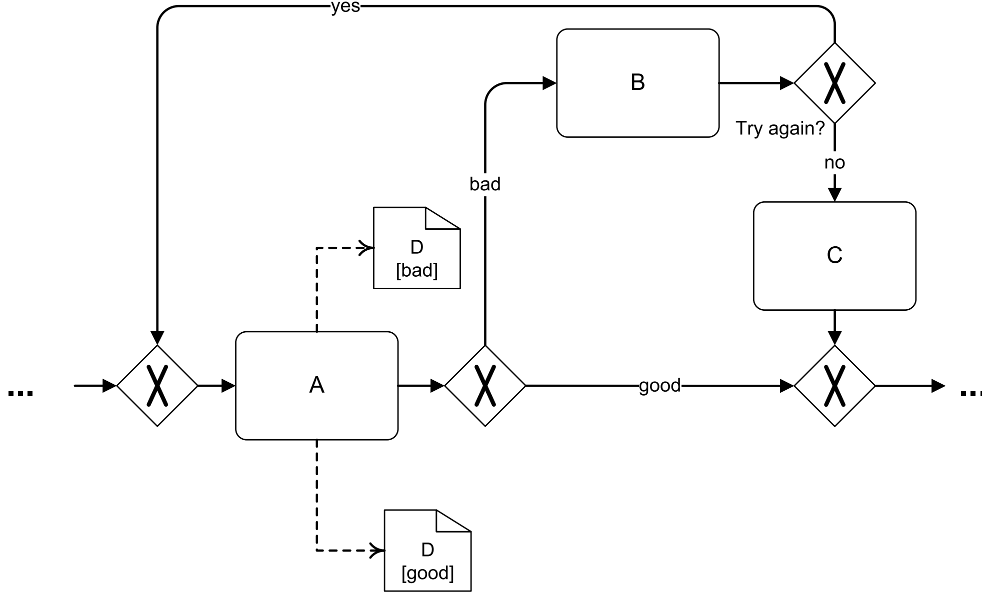


Figure 5.16: A process excerpt with a false alarm

we can find the set of contradicting data states CON_s. Thus, we need to explicitly add contradicting conditions to the formula.

$$contraCondition_{(d,s)} = \mathbf{G}(\bigwedge_{s' \in CON_s} \neg\mathbf{state}(d, s')) \quad (5.25)$$

The correct data condition for this case would be

$$stableCondition_{LTL} = \bigwedge_{d \in \mathcal{D}_Q} (\bigvee_{(d,s) \in \mathcal{D}ata\mathcal{S}tate_Q} (\mathbf{state}(d, s) \wedge \mathbf{G}(\bigwedge_{s' \in CON_s} \neg\mathbf{state}(d, s')))) \quad (5.26)$$

Now, model checking the formula $\mathbf{G}(\mathbf{executed}(A) \wedge \mathbf{state}(D, bad) \wedge \mathbf{G}\neg\mathbf{state}(D, good) \rightarrow \mathbf{F}(\mathbf{executed}(C)))$ gives a positive result. However, if we abstract from specifying the execution of activity A and focus only on the data condition, i.e., we model check the formula $\mathbf{G}(\mathbf{state}(D, bad) \wedge \mathbf{G}\neg\mathbf{state}(D, good) \rightarrow \mathbf{F}(\mathbf{executed}(C)))$ we get a negative result. The reason for this negative answer is the fact that whenever $\mathbf{state}(D, bad) \wedge \mathbf{G}\neg\mathbf{state}(D, good)$ holds true for some state it will remain true in all subsequent states until the data object D changes its value to some other value or will remain ever so. Referring to Figure 5.16, for the execution state that occurs immediately after the XOR join, call it s_j, sub-formula $\mathbf{state}(D, bad) \wedge \mathbf{G}\neg\mathbf{state}(D, good)$ is true, since no activity changes the value of D. Thus, at that state s_j the model checker records that the condition is $true$. However, assuming that no further execution of C, model checker fails to find any future execution state where $\mathbf{executed}(C)$ will hold. Thus, signaling violation to the formula.

Conceptually, the model is correct since C is executed in response to the *first* occurrence of the data condition $\textbf{state}(D, bad) \wedge \textbf{G}\neg\textbf{state}(D, good)$. The fact that the data condition holds true afterwards must not indicate failure. There are two possibilities to get rid of this false alarm. The first way is to be sure of having the predicate $\textbf{executed}(act)$ always as part of the rule's condition. Even if the user uses an anonymous activity at rule specification time, it is always possible at checking time against a specific process model to resolve the activity(ies) that generate that data condition. Assuming a function $update_{(d,s)}$ that returns all activities in a process model that update the data object d to the state (value) s, we can have the following representation of conditional presence pattern.
The PLTL formula for conditional presence pattern is

$$\textbf{G}((\bigvee_{A \in \bigcup_{(d,s) \in \mathcal{D}ataState_{Q}} update_{(d,s)}} \textbf{executed}(A)) \wedge stableCondition_{LTL} \rightarrow \textbf{F}(\textbf{executed}(B)/effect)) \quad (5.27)$$

The $stableCondition_{LTL}$ in the above formula refers to Formula 5.26. In all subsequent formulas where $stableDataCondition_{LTL}$ appears, it refers to Formula 5.26.

The same correct data condition above can be used within CTL formulas. However, a small syntactical change has to be done by adding the for all quantifier before the global operator.

$$stableCondition_{CTL} = \bigwedge_{d \in \mathcal{D}_{Q}} (\bigvee_{(d,s) \in \mathcal{D}ataState_{Q}} (\textbf{state}(d, s) \wedge \textbf{AG}(\bigwedge_{s' \in CON_s} \neg\textbf{state}(d, s'))))) \quad (5.28)$$

The CTL formula for conditional presence pattern is

$$\textbf{AG}((\bigvee_{A \in \bigcup_{(d,s) \in \mathcal{D}ataState_{Q}} update_{(d,s)}} \textbf{executed}(A)) \wedge stableCondition_{CTL} \rightarrow \textbf{AF}(\textbf{executed}(B)/effect)) \quad (5.29)$$

The $stableCondition_{CTL}$ in the above formula refers to Formula 5.28. In all subsequent formulas where $stableDataCondition_{CTL}$ appears, it refers to Formula 5.28.

The other possibility to reflect that response is needed only for the *first* occurrence of the data condition, in case that the user abstracts from a specific activity, is to explicitly model that in the temporal logic formula. Referring to the process excerpt of Figure 5.16 and the LTL formula we had so far $\textbf{G}(\textbf{state}(D, bad) \wedge \textbf{G}\neg\textbf{state}(D, good) \rightarrow \textbf{F}(\textbf{executed}(C)))$, we can modify that LTL formula to look like $\textbf{G}(\textbf{state}(D, bad) \wedge \textbf{G}\neg\textbf{state}(D, good) \wedge \neg\textbf{O}(\textbf{state}(D, bad) \wedge \textbf{G}\neg\textbf{state}(D, good)) \rightarrow \textbf{F}(\textbf{executed}(C)))$. We modified the condition of the formula by looking for the very first state in which the data condition holds. That is the state which is not preceded by any other state where the data condition held. To represent the checking for the *first* state in which the data condition

holds in CTL, we need to think it differently in the absence of past time operators in CTL. That is, we require the $negation$ of the data $condition$ to hold until a state where the $condition$ holds and C executes thereafter, or the condition never holds. In other words, we must not find any sequence of states where the $condition$ ceases from holding and in some states it holds without C executing afterwards. Formally, $\neg\mathbf{E}[\neg(((\textbf{state}(D, bad) \wedge \mathbf{G}(\neg\textbf{state}(D, good))))\wedge\mathbf{AF}(\textbf{executed}(C)))\mathbf{U}\,(((\textbf{state}(D, bad)\wedge\mathbf{G}(\neg\textbf{state}(D, good))))\wedge \neg(((\textbf{state}(D, bad) \wedge \mathbf{G}(\neg\textbf{state}(D, good)))) \wedge \mathbf{AF}(\textbf{executed}(C))))]$.

5.2.5.3 Conditional Before-scope Presence

There are many ways where the before-scope presence (precedence) pattern , see Section 5.2.3.3, can be refined with data conditions. A data condition could be attached to the source, target activity, or both.

Precedence to conditional activity execution is one case where we need to be sure that an activity A has been executed before another activity B executes under a data condition. An example, “before archiving confirmed order; a copy must have been sent to the marketing department”. This pattern is captured in Figure 5.17.

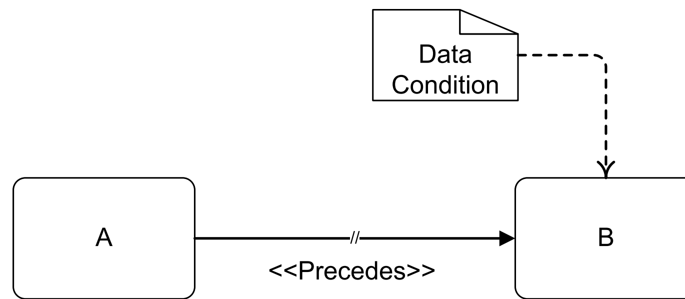


Figure 5.17: Precedence to conditional activity execution pattern

Conditional Precedence is another case where the execution of activity B must have been preceded by execution of an activity A that resulted in a data condition. In some cases, we can abstract from the specific activity A using anonymous activity. This is captured in the BPMN-Q query shown in Figure 5.18.

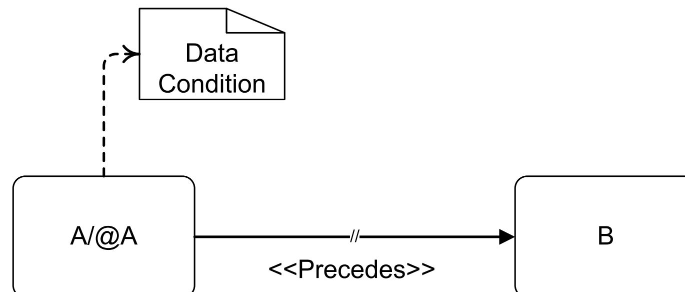


Figure 5.18: Conditional precedence pattern

Finally, both cases discussed above can be merged by attaching data conditions on both activities.

The PLTL formula for precedence to conditional activity execution is

$$\mathbf{G}(\mathbf{ready}(B) \wedge dataCondition \rightarrow \mathbf{O}(\mathbf{executed}(A))) \tag{5.30}$$

The CTL formula for precedence to conditional activity execution is

$$\neg\mathbf{E}[\neg\mathbf{executed}(A)\ \mathbf{U}\ (\mathbf{ready}(B) \wedge dataCondition)] \tag{5.31}$$

In the above two formulas $dataCondition$ refers to Formula 5.18.

The PLTL formula for conditional precedence pattern is

$$\mathbf{G}(\mathbf{ready}(B) \rightarrow \mathbf{O}((\mathbf{executed}(A)/true) \wedge stableCondition_{LTL})) \tag{5.32}$$

The CTL formula for conditional precedence pattern is

$$\neg\mathbf{E}[\neg((\mathbf{executed}(A)/true) \wedge stableCondition_{CTL})\ \mathbf{U}\ \mathbf{ready}(B)] \tag{5.33}$$

The $stableCondition_{CTL}$ refers to the general form of data condition of Formula 5.28. In all subsequent formulas where the term $stableDataCondition_{CTL}$ appears, it refers to Formula 5.28.

5.2.5.4 Conditional Before-scope Absence

Before-scope absence rules as introduced in Section 5.2.3.4 can be refined with data dependencies as follows: In case an activity is about to execute and a certain data condition holds; it is required that some other activity must have not been executed before. For instance, in an order processing process, it is required that at the time of executing the "archive order" activity where the order is cancelled; the "ship order" activity must have never been executed. Figure 5.19 shows the BPMN-Q query capturing this pattern.

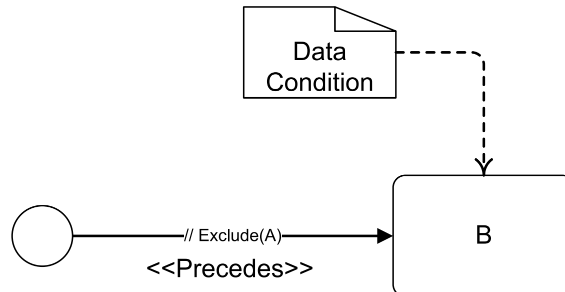


Figure 5.19: Conditional before-scope absence pattern

The PLTL formula for the conditional before-scope absence pattern is

$$\mathbf{G}(\mathbf{ready}(B) \wedge dataCondition \rightarrow \neg(\mathbf{executed}(A)/effect)\ \mathbf{S}\ \mathbf{start}) \tag{5.34}$$

Notice that in the above formula, we might require a certain effect (data condition) to be absent rather than a specific activity. Both $dataCondition$ and $effect$ refer to Formula 5.18.
The CTL formula for conditional before-scope absence is

$$\neg\mathbf{EF}(\mathbf{start} \wedge \mathbf{EF}((\mathbf{executed}(A)/effect) \wedge \mathbf{EF}(\mathbf{ready}(B) \wedge dataCondition))) \tag{5.35}$$

The CTL formula reflects the history-looking requirement by stating that it is not possible to reach an activity A, or a data $effect$, from which activity B and the $dataCondition$ are reachable. Both $dataCondition$ and $effect$ refer Formula 5.18.

5.2.5.5 Conditional After-scope Absence

In the same fashion, process models might be prohibited from executing certain activities when some data condition becomes true. Figure 5.20 shows the BPMN-Q query to capture this pattern.

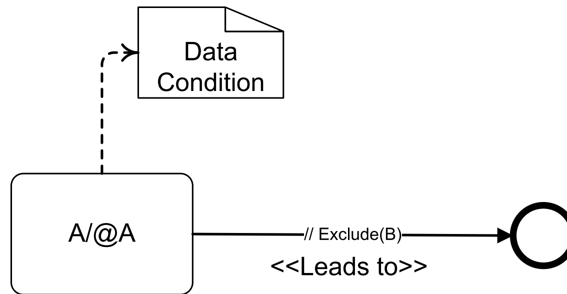


Figure 5.20: Conditional absence pattern

Also, this pattern is considered as a refinement on the global-scope absence and after-scope absence patterns , discussed in Sections 5.2.3.2, 5.2.3.6 respectively, where an activity must not be executed based on a condition. It is also possible to model the absence by the data effect to be absent rather than a specific activity.

Following the same discussion about the contradicting data values and representation of data conditions, see Section 5.2.5.2 , the formalization of conditional absence rules is as follows.

The PLTL formula for conditional after-scope absence is

$$\mathbf{G}((\bigvee_{A \in \bigcup_{(d,s) \in \mathcal{D}ataState_Q} update_{(d,s)}} \mathbf{executed}(A)) \wedge stableCondition_{LTL} \rightarrow \neg(\mathbf{executed}(B)/effect)\ \mathbf{U}\ \mathbf{end}) \tag{5.36}$$

The CTL formula for that pattern is

$$\mathbf{AG}((\bigvee_{A \in \bigcup_{(d,s) \in \mathcal{D}ataState_Q} update_{(d,s)}} \mathbf{executed}(A)) \land stableCondition_{CTL} \rightarrow \mathbf{A}[\neg(\mathbf{executed}(B)/effect) \; \mathbf{U} \; \mathbf{end}]) \quad (5.37)$$

5.2.5.6 Conditional Between-scope Absence

There are several ways to refine the between-scope absence discussed in Section 5.2.3.7 with data conditions. As discussed for conditional presence and precedence patterns, data conditions could be attached to source, target, or both activities.

The case of Figure 5.21(a) represents a situation where a conditional execution of an activity C must be preceded with an execution of another activity A; in between activity B, or a data effect, never occurs. The case of Figure 5.21(b) is a variant of the first case where the unconditional execution of an activity C must be preceded with another activity A with a specific output data condition. In between activity B, or a data effect, is not allowed to occur. It is possible to attach data conditions to both source and target activities.

The pattern in Figure 5.21(c) shows another case where the concern is to check that after the execution of an activity A resulting in a specific data condition, it must be followed with execution of activity C, or an effect, in between activity B or another data effect is not allowed to execute.

The PLTL formula for conditional between-scope absence type I is

$$\mathbf{G}(\mathbf{ready}(C) \land dataCondition \rightarrow \neg(\mathbf{executed}(B)/effect) \; \mathbf{S} \; \mathbf{executed}(A)) \quad (5.38)$$

The CTL formula for conditional between-scope absence type I is

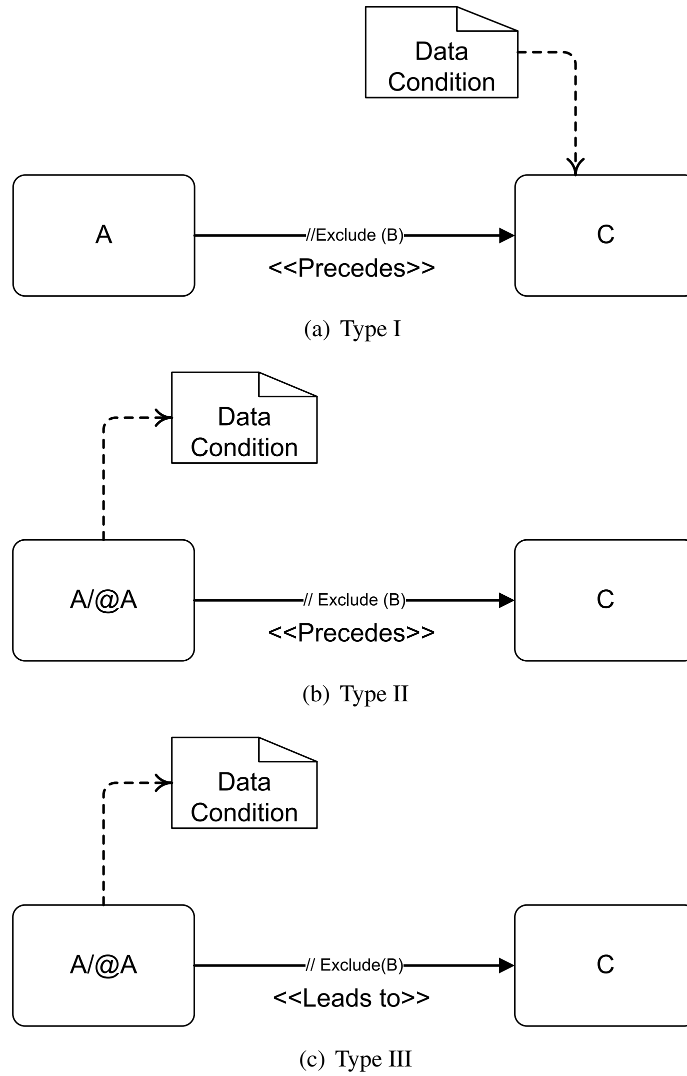
$$\neg\mathbf{E}[\neg\mathbf{executed}(A) \mathbf{U} \; \mathbf{ready}(C) \land dataCondition]) \land \neg\mathbf{EF}(\mathbf{executed}(A) \land \mathbf{EF}((\mathbf{executed}(B)/effect) \land \mathbf{EF}(\mathbf{ready}(C) \land dataCondition))) \quad (5.39)$$

The PLTL formula for conditional between-scope absence type II is

$$\mathbf{G}(\mathbf{ready}(C) \rightarrow \neg(\mathbf{executed}(B)/effect) \; \mathbf{S} \; ((\mathbf{executed}(A)/true) \land stableDataConidition_{LTL})) \quad (5.40)$$

The CTL formula for conditional between-scope absence type II is

$$\neg\mathbf{E}[\neg((\mathbf{executed}(A)/true) \land stableCondition_{CTL}) \; \mathbf{U} \; \mathbf{ready}(C)] \land \neg\mathbf{EF}((\mathbf{executed}(A)/true) \land stableCondition_{CTL}) \land \mathbf{EF}((\mathbf{executed}(B)/effect) \land \mathbf{EF}(\mathbf{ready}(C))) \quad (5.41)$$



(a) Type I

(b) Type II

(c) Type III

Figure 5.21: Conditional between-scope absence patterns

The PLTL formula for conditional between-scope absence type III is

$$\mathbf{G}((\bigvee_{A \in \bigcup_{(d,s)\in \mathcal{D}ataState_{\mathcal{Q}}} update_{(d,s)}} \mathbf{executed}(A)) \wedge stableCondition_{LTL} \rightarrow \neg(\mathbf{executed}(B)/effect)\ \mathbf{U}\ (\mathbf{executed}(C)/effect2)) \quad (5.42)$$

The CTL formula for conditional between-scope absence type III is

$$\mathbf{AG}((\bigvee_{A \in \bigcup_{(d,s)\in \mathcal{D}ataState_{\mathcal{Q}}} update_{(d,s)}} \mathbf{executed}(A)) \wedge stableCondition_{CTL} \rightarrow \mathbf{A}[\neg(\mathbf{executed}(B)/effect)\ \mathbf{U}\ (\mathbf{executed}(C)/effect2)]) \quad (5.43)$$

5.2.6 Modeling Complex Rules

So far, we have shown basic patterns to express the various types of rules concerning control flow, data flow, and conditional flow. However, it is possible to compose these patterns together to make complex rules. As an example, we might want to state that activities A and B are mutually exclusive. We can model this rule as shown in Figure 5.22 by using the before-scope absence pattern type I and the after-scope absence pattern. A

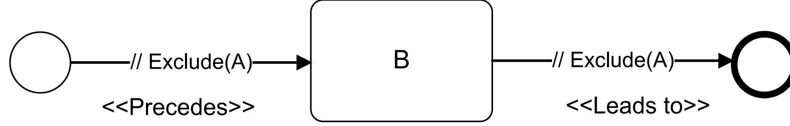


Figure 5.22: Mutual exclusion between A and B

complex rule will be decomposed into the basic rules and its TL formula will be the conjunction of the TL of its basic rules. To prevent arbitrarily composed complex rules, we require a complex rule to be a well formed one.

Definition 5.1. *[Well formed behavioral queries]* A behavioral BPMN-Q query $\mathcal{Q} = (\mathcal{A}_\mathcal{Q}, \mathcal{E}_\mathcal{Q}, \mathcal{D}_\mathcal{Q}, \mathcal{S}_\mathcal{Q}, \mathcal{D}ata\mathcal{S}tate_\mathcal{Q}, \mathcal{P}_\mathcal{Q}, \mathcal{DF}_\mathcal{Q}, isAnonymous, \mathcal{X}, \mathcal{TYPE})$ is well formed query if:

- $\forall n \in (\mathcal{A}_\mathcal{Q} \cup \mathcal{E}_\mathcal{Q}) : (n, _) \in \mathcal{P}_\mathcal{Q} \vee (_, n) \in \mathcal{P}_\mathcal{Q} \vee (((d,s),n) \in \mathcal{DF}_\mathcal{Q} : \mathcal{TYPE}(((d,s),n)) = behavioral)$. Each node must be a source or a target of a path edge or receiving a behavioral data flow edge,

- $\forall p \in \mathcal{P}_\mathcal{Q}\ \mathcal{TYPE}(p) \in \{precedes, leadsto\}$. All path edges must be stereotyped with either precedes or leads to,

- $\forall (a,(d,s)) \in \mathcal{DF}_\mathcal{Q} : \mathcal{TYPE}((a,(d,s))) = structural$. No behavioral data flow edges allowed to be outgoing from activities,

- $\exists a \mathcal{A}_\mathcal{Q} : isAnonyomous(a) = true \rightarrow \nexists((d,s),a) \in \mathcal{DF}_\mathcal{Q} : \mathcal{TYPE}(((d,s),a)) = behavioral$. No behavioral data flow edges allowed to be incoming into anonymous activities,

- $\exists((d,s),a) \in \mathcal{DF}_\mathcal{Q} : \mathcal{TYPE}(((d,s),a)) = structural \rightarrow \exists(o,a) \in \mathcal{P}_\mathcal{Q} : \mathcal{TYPE}((o,a)) = precedes$. Incoming data flow to an activity must have a precedes path associated to the activity where it is the target,

- $\exists((d,s),a) \in \mathcal{DF}_\mathcal{Q} : \mathcal{TYPE}(((d,s),a)) = structural \rightarrow \not\exists(o,a) \in \mathcal{P}_\mathcal{Q} : \mathcal{TYPE}((o,a)) = leadsto$. Incoming data flow to an activity must have no leads to path where the activity is the target,

- $\exists(a,(d,s)) \in \mathcal{DF_Q} \wedge \mathcal{TYPE}((a,(d,s))) = structural \wedge isAnonymous(a) = true \rightarrow \nexists(o,a) \in \mathcal{P_Q} \wedge \mathcal{TYPE}((o,a)) = precedes$. Outgoing data flow from an anonymous activity must have no precedes path where the activity is the target.

- $\forall s \in \mathcal{E}_{\mathcal{Q}_s} : \nexists(o,s) \in \mathcal{P_Q}$. *Start events cannot be targets to any path edge.*

- $\forall e \in \mathcal{E}_{\mathcal{Q}_e} : \nexists(e,o) \in \mathcal{P_Q}$. *End events cannot be sources to any path edge.*

- $\forall(o,e) \in \mathcal{P_Q} : e \in \mathcal{E}_{\mathcal{Q}_e} \rightarrow \mathcal{TYPE}((o,e)) = leadsto \wedge \mathcal{X}(o,e) \neq \emptyset$. *An end event in a query can be only a target of a leads to path and the exclude property of the path edge must not be empty.*

5.3 Consistency Checking Among Rules

With the divergent sources of compliance requirements, it is likely to have *redundant* or *conflicting* compliance requirements [19]. Redundant requirements would consume more effort verifying the same property. On the other hand, conflicting requirements would consume a lot of effort before figuring out that it is not possible to satisfy them collectively.

While it is hard to derive a consistent subset of rules, it is inevitable to check for inconsistency [49]. In Section 5.3.1 we discuss various types of redundancy and how to check them. In Section 5.3.2 we address the issue of conflict detection and provide means to decide about conflicts.

Recall that $\mathbb{Q}$ is the set of all BPMN-Q queries, $\mathbb{P}$ is the set of all business process models and $\mathbb{TAG}$ is the set of all tags that can be used to annotate queries and processes.

Definition 5.2. *[Related compliance rules]* for a specific set of tags $\mathbb{T} \subseteq \mathbb{TAG}$ the set $q_{\mathbb{T}} \subset \mathbb{Q}$ contains all queries that are assigned this set of tags, $\forall q \in q_{\mathbb{T}}(\mathbb{T} \subseteq annotate(q))$

Definition 5.2 states that $q_{\mathbb{T}}$ contains compliance rules (queries) that have the same set of tags. Therefore, the requirements imposed by these rules have to be checked for consistency before proceeding to check them against process models.

5.3.1 Redundancy Checking

Redundancy occurs when two or more rules are logically equivalent [49]. Deciding about redundant rules helps remove them from checking and save effort without losing any compliance requirement.

Definition 5.3. *[Redundancy]* A rule r_i is redundant within rule set $\mathbb{R}$ if and only if for any arbitrary process model p: $p \models \bigwedge_{r_k \in \mathbb{R} \setminus \{r_i\}} r_k \leftrightarrow p \models \bigwedge_{r_j \in \mathbb{R}} r_j$

According to Definition 5.3 the redundancy is a property for a collection of compliance rules and is independent from the set of process models under investigation.

Considering the compliance patterns discussed in Section 5.2, redundancy can occur in any of the following situations:

1. (Duplication) Two rules $R1$ and $R2$ state the same requirements. This might occur due to two different developments of compliance rules.

2. (Subsumption) An example is a response pattern $R1$ on the form $\mathbf{G}(\textbf{executed}(a) \rightarrow \mathbf{F}(\textbf{executed}(b)))$ that is subsumed by a between-scope absence rule $R2$ on the form $\mathbf{G}(\textbf{executed}(a) \rightarrow \neg\textbf{executed}(c)\ \mathbf{U}\ \textbf{executed}(b))$.

3. (Mutual Exclusion) A rule $R1$: $\mathbf{G}(\textbf{executed}(a) \rightarrow (\neg\textbf{executed}(b)\mathbf{S}\ \textbf{start}) \wedge (\neg\textbf{executed}(b)\ \mathbf{U}\ \textbf{end}))$ is redundant to $R2 : \mathbf{G}(\textbf{executed}(b) \rightarrow (\neg\textbf{executed}(a)\mathbf{S}\ \textbf{start}) \wedge (\neg\textbf{executed}(a)\ \mathbf{U}\ \textbf{end}))$ since $R1$ states that $\textbf{executed}(a)$ and $\textbf{executed}(b)$ are mutually exclusive. $R2$ also tells that $\textbf{executed}(b)$ and $\textbf{executed}(a)$ are mutually exclusive. The redundancy is because of the commutative nature of mutual exclusion relation.

4. (Contrapositive) Two rules state the same requirements differently. For instance $R1$ states the "open account if the risk is low"; while R2 states "if risk is high never open an account". This could be considered as a special type of mutual exclusion that could be expressed in different ways.

In the first case, there is a duplication of rules and rules are equivalent. It is trivial to detect this case and only one copy of the rule is retained and all other copies are removed.

The subsumption case occurs when some rule states a stronger conclusion than the other. For instance a response rule $R1$:$\mathbf{G}(\textbf{executed}(a) \rightarrow \mathbf{F}(\textbf{executed}(b)))$ and a between-scope absence $R2$:$\mathbf{G}(\textbf{executed}(a) \rightarrow \neg\textbf{executed}(c)\mathbf{U}\ \textbf{executed}(b))$. It is obvious that the satisfaction of $R1 \wedge R2$ is determined by $R2$. That is, there is no chance that $R1$ is false and $R2$ is true. Thus, $R1$ is unnecessary. For the case of mutual exclusion, it is sufficient to ensure that $\textbf{executed}(a)$ is mutually exclusive to $\textbf{executed}(b)$ without the other way around. The last case is another sort of equivalence similar to logical equivalence between an implication $p \rightarrow q$ and its contrapositive $\neg q \rightarrow \neg p$.

Duplicate and mutual exclusion redundancy can be detected easily. However, subsumption and contrapositive redundancy need further investigation about different patterns causing them in addition to domain knowledge. It is necessary to get rid of redundant rules prior to checking them against process models in order to reduce checking effort.

In general, a rule $R1$ subsumes another rule $R2$ when the condition of $R2$ logically implies the condition of $R1$, i.e., $R1$ has a weaker or equivalent condition than $R2$ and the consequent of $R1$ logically implies the consequent of $R2$, i.e., $R1$ states a stronger or equivalent conclusion than $R2$. Table 5.1 summarizes the possible occurrences of subsumption redundancies among the compliance patterns discussed in Section 5.2. Notice that only patterns with possibility of subsumption redundancies are mentioned.

Obviously, every control flow pattern subsumes its conditional counterpart. For instance, an after-scope presence rule $\mathbf{G}(\textbf{executed}(a) \rightarrow \mathbf{F}(\textbf{executed}(b)))$ subsumes a con-

Pattern	After Pres.	Before Pres.	Cond. Pres.	Prec. Cond. Exec.	Cond. Prec.	Cond. Before Abs.	Cond. After Abs.	Cond. Bet. Abs. Type 1	Cond. Bet. Abs. Type 3
After Pres.			•						
After Abs.							•		
Before Pres.				•					
Before Abs.						•			
Response with Abs.	•		•						•
Prec. with Abs.		•		•				•	
Cond. Prec.		•							
Cond. Bet. Abs. Type 1				•					
Cond. Bet. Abs. Type 2					•				
Cond. Bet. Abs. Type 3			•						

Table 5.1: Possible subsumption redundancies

ditional presence rule $\mathbf{G}(\textbf{executed}(a) \wedge \ cond \rightarrow \mathbf{F}(\textbf{executed}(b)))$ since $\textbf{executed}(a) \wedge$ $cond \rightarrow \textbf{executed}(a)$ and the consequent of both patterns is the same. Also, a response with absence rule $R1$ $\mathbf{G}(\textbf{executed}(a) \ \rightarrow \ \neg\textbf{executed}(c)\mathbf{U}\ \textbf{executed}(b))$ subsumes an after-scope presence rule $R2$ $\mathbf{G}(\textbf{executed}(a) \rightarrow \mathbf{F}(\textbf{executed}(b)))$ because the condition part of both rules is the same and the consequent of $R1$ is stronger than that of $R2$. The previous response with absence rule $R1$ also subsumes a conditional presence rule $R3$ $\mathbf{G}(\textbf{executed}(a) \wedge \ cond \ \rightarrow \ \mathbf{F}(\textbf{executed}(b))$. In this case both the condition of $R1$ is weaker than that of $R3$ and the consequent of $R1$ is stronger than that of $R3$. In the same way, other subsumption redundancy cases can be described.

Contrapositive redundancy is seen as some sort of logical equivalence between compliance rules in case one rule is on the form $p \rightarrow q$ and the other is on the form $\neg q \rightarrow \neg p$. It is not possible to have contrapositive redundancy for control flow rules as it is not possible to state negations in rule conditions. Contrapositive redundancy occurs among certain conditional rules and under certain domain knowledge. Table 5.2 summarizes the situations where contrapositive redundancy is possible.

The case of conditional presence redundancy to conditional before absence could be exemplified with a rule $R1$: $\mathbf{G}(\textbf{state}(Claim, fraudulent) \rightarrow$ $\mathbf{F}(\textbf{state}(Investigation, started)))$ and $R2$: $\mathbf{G}(\neg\textbf{state}(Investigation, started) \rightarrow$ $\neg\textbf{state}(Claim, fraudulent)\mathbf{S}\ \textbf{start})$. $R1$ states that in any case where an insurance claim is found to be fraudulent, a investigation must follow. On the other hand, $R2$ states that *not* initiating an investigation means that the claim is *not* fraudulent.

Redundancy between conditional precedence and after absence in a situation where there is a rule $R3$: $\mathbf{G}(\textbf{state}(Account, open) \rightarrow \mathbf{O}(\textbf{state}(Risk, low)$

Pattern	Cond. Presence	Prec. Cond. Execution	Cond. Prec.	Cond. Before Abs.	Cond. After Abs.
Cond. Presence				•	
Prec. Cond. Execution					•
Cond. Prec.					•
Cond. Before Abs.	•	•			•
Cond. After Abs.		•	•	•	

Table 5.2: Possible contrapositive redundancies

$\wedge$ $\mathbf{G}(\neg\textbf{state}(Risk, high))))$ and $R4$: $\mathbf{G}(\textbf{state}(Risk, high) \wedge \mathbf{G}(\neg\textbf{state}(Risk, low)) \rightarrow \neg\textbf{state}(Account, open)\mathbf{U}$ **end**). $R3$ requires that whenever the account is opened, its risk assessment must have been low, and never high. $R4$ states the same requirement but in a different way. That is, whenever the risk assessment is found to be high, there is no chance to open the bank account.

However, it is not always explicitly stated in rules the condition and its negation. For example, the last two rules stated above $R3$, $R4$. One rule states that if the account to be open then the risk must have been low and never high. The other states the same fact but the other way around. That is, if the risk is high and never low then there is no chance to open the account during the remainder of the process.

Next, we will prove the logical equivalence between rules like $R3$ and $R4$ under the domain knowledge of contradicting data values. Other cases, could be proved using the same argument.

Theorem 5.1. *[Redundant Conditional Rules]* A conditional after-scope absence rule on the form $\mathbf{G}(contraCond \wedge \mathbf{G}(\neg cond) \rightarrow (\neg action)\mathbf{U}$ **end**) is redundant to a conditional precedence on the form $\mathbf{G}(action \rightarrow \mathbf{O}(cond \wedge \mathbf{G}(\neg contrCond)))$ if and only if $cond$ and $contraCond$ are contradicting.

Proof. Before we prove Theorem 5.1 we need to do some inference and rewriting of the rules above. If $cond$ and $contraCond$ are known to be contradicting, we can infer that $cond \equiv \neg contraCond$. Similarly, we can infer that $contraCond \equiv \neg cond$.

Also, to ease the proof without changing the semantics we can rewrite the conditional after-scope absence as $\mathbf{G}(contraCond \wedge \mathbf{G}(\neg cond) \rightarrow \mathbf{G}(\neg action))$. We replaced the $\mathbf{U}$ operator with a $\mathbf{G}$ where they give the same meaning, as **end** logically means the termination of process execution.

Based on that inference, we can rewrite the modified conditional after-scope absence as $\mathbf{G}(\mathbf{G}(\neg cond) \rightarrow \mathbf{G}(\neg action))$, let it be $R1$. We can rewrite the conditional precedence rule using future-only temporal operators as $\mathbf{F}(action) \rightarrow \neg action\mathbf{U}\mathbf{G}(cond)$, let it be $R2$. we prove Theorem 5.1 by contradiction.

Let a system $\mathcal{M} \models R1$. Thus $s_0 \models R1$. I.e., $\exists s_m : s_0, \ldots, s_m \wedge cond \notin L(s_m) \wedge action \notin L(s_m)$ and $\forall s_k$ that come after s_m also $cond \notin L(s_k) \wedge action \notin L(s_k)$.

Assume that $\mathcal{M} \not\models R2$. Thus, $\exists s_v : s_0, \ldots, s_v$ where s_v is the very first state where $action \in L(s_v)$. Then, for $\mathcal{M} \not\models R2$ to hold, there must be some state s_t where $s_0 \ldots, s_t, \ldots, s_v \wedge s_t \not\models \mathbf{G}(cond)$. I.e., $cond \notin L(s_t)$. But, since $\mathcal{M} \models R1$. We can derive that $action \notin L(s_t)$. and for all states that come after s_t, including s_v, $action \notin L(s_v)$. Thus we reach a contradiction where we require $action \in L(s_v)$ and $action \notin L(s_v)$ $\square$

5.3.2 Conflict Checking

While redundancy checking is needed to reduce effort, conflict checking is needed to decide the satisfiability of these related compliance requirements. For a conflict free set of compliance rules, it is possible to find a process model that satisfies all of these requirements.

To decide about conflict freedom, we utilize existing approaches for LTL model checking. We depend on the fact that for each LTL formula, it is possible to find a computation that satisfies this formula, in case there is one. Thus, we can construct the conjunction of these individual LTL formula and look for a computation that satisfies them. However, we show that to reach a correct decision about conflict-freedom, we need to add extra information to the satisfiability check [15].

5.3.2.1 LTL to Büchi Automaton

It is possible for each LTL formula to generate a Büchi automaton that represents a computation to satisfy the formula, if there is one. The Büchi automaton describes systems with infinite behavior. An LTL formula is satisfiable if its Büchi automaton has accepting runs and thus a non-empty set of accepting states [24]. Approaches like [44, 43] are capable of generating a Büchi automaton for a given LTL formula. The difference is in the efficiency of the algorithms behind each of them.

LTL to Büchi automaton is an intermediate step in LTL model checking. LTL model checking is done by first negating the LTL formula. A Büchi automaton is generated for the negated formula. In the mean time a Büchi automaton is generated for the system under investigation. If there are common words that can be accepted by both automata, this means that the system can behave in a way that satisfies the negated formula. Thus, it violates the original formula under investigation.

Related to compliance rules, one can check conflicts among a set of rules by formulating a conjunction of their LTL counterparts, generating a Büchi automaton for it and finally checking for accepting states. However, we will show that in order to correctly utilize that approach, we have to add domain specific knowledge, expressed as LTL formulas to the conjunction.

5.3.2.2 Detecting Conflicts between Compliance Rules

The approach to detect *conflicts* among compliance rules in $q_{\mathbb{T}}$ depends on the conjunction of the LTL formulas of such rules in addition to any related knowledge that is specific to the domain. Also, knowledge about execution environment is relevant. The conjunction of such formulas is then input to an algorithm that generates a corresponding Büchi automaton. In case that the resulting automaton has no accepting states, we can conclude that the compliance rules are conflicting, the compliance officer is informed about this to take correcting actions. The notion of conflict comes from the fact that there is no possible computation (process model) that is capable of generating such behavior stated in the rules and in the mean time consistent with the domain-specific knowledge.

The purpose of this section is to show the knowledge to be included about the domain to help reach correct decisions about conflict freedom. It is of crucial importance to include correct assumptions about the system/domain under investigation in order to correctly decide about the conflict-freedom of a specification [28, 37]. We will describe how such assumptions are derived from the domain knowledge and from the behavior of process models. This knowledge is encoded as LTL formulas that are added to the conjunction to be checked for conflicts.

We start with a set of pairs of exemplary compliance rules. We illustrate the resulting Büchi automaton for the conjunction of compliance rules for each pair. Of course, one assumes the compliance rules to be conflict free, if the Büchi automaton has accepting runs, whereas inconsistent compliance rules are indicated by an automaton that has no accepting runs. However, our examples show that detection of conflict is not as straight-forward as expected.

Example 1. (Cyclic Dependency) Imagine two rules that are related to registration of foreigners at Police. Let the first rule $R1$ state that obtaining a health insurance precedes registration, whereas the second rule $R2$ states the opposite, i.e., registration precedes obtaining health insurance. If we represent the registration activity with a and the obtaining health insurance activity with b, the corresponding LTL formulas would be $\mathbf{G}(\mathbf{executed}(b) \rightarrow \mathbf{O}(\mathbf{executed}(a)))$ and $\mathbf{G}(\mathbf{executed}(a) \rightarrow \mathbf{O}(\mathbf{executed}(b)))^*$.

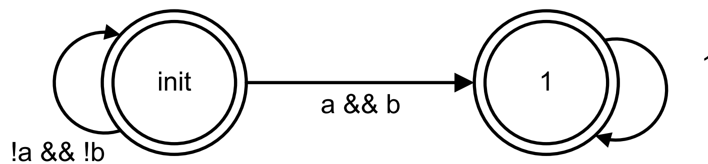


Figure 5.23: Automaton for the conjunction of rule $R1$ and $R2$

*We deviate a bit from the original template of before-scope presence to simplify the discussion

The Büchi automaton generated for the conjunction of both formulas is illustrated in Figure 5.23†. Although both rules, $R1$ and $R2$, are conflicting due to cyclic dependency, the respective Büchi automaton has two accepting runs. In particular, the following runs are possible.

1. Neither a nor b are executed at all, which is represented by the transition from state $init$ to itself.

2. Both activities, a and b are executed in a single step, which is represented by the transition from $init$ to state 1.

The phenomenon that led to the first accepting run is vacuous satisfiability [28]. The rules $R1$ and $R2$ are satisfied by all process models that do not contain both activities a and b. We have discussed vacuous satisfiability before and we pointed out that it is not the type of satisfiability we expect. That is, we expect process models to exhibit a behavior where both the condition and the consequent of a rule are satisfied.

The second accepting run in Figure 5.23 is also invalid in our context. The execution semantics we adopt (cf. Section 4.1.3) assumes interleaving semantics. That is, two activities cannot complete their execution at the very same time. That, in turn, is reflected in any behavioral model of a process model, which is derived by transformation to some intermediary representation, for instance, Petri nets. These models do not contain a single state transition, in which two activities finish their execution.

Example 2. (Contradictions) Imagine two rules describe the relation between receiving payment and shipment of goods. Rule $R3$ specifies that the reception of payment leads to the activity of sending goods by DHL. A second rule $R4$, in turn, states that after the reception of payment by a premium customer, the goods have to be sent by express delivery. Let a be the activity of receiving the payment, while the send activities are represented by b in case of DHL, and c in case of express delivery, respectively. Then, the resulting LTL formula would be $R3$ $\mathbf{G}(\mathbf{executed}(a) \rightarrow \mathbf{F}(\mathbf{executed}(b))) \wedge \mathbf{G}(\mathbf{executed}(a) \wedge premium \rightarrow \mathbf{F}(\mathbf{executed}(c)))$. Of course, the Büchi automaton, which was omitted due to its size, has accepting runs, owing to the phenomena discussed for the first example, i.e., vacuous satisfiability and state transitions with more than one activity being executed at a time. However, there are further accepting runs that might not be traced back to these phenomena. These runs correspond to scenarios, in which activity a is executed and thereafter activity b is executed and c is conditionally executed, as shown in Figure 5.24. It is obvious that the process fragment does not make sense as it requires to send the goods twice in case of premium customer. In this situation we have to explicitly add the knowledge about contradicting activities as they are defined as part of the domain knowledge, see Section 4.4.

†The symbols !, & are used to represent negation and conjunctions respectively.

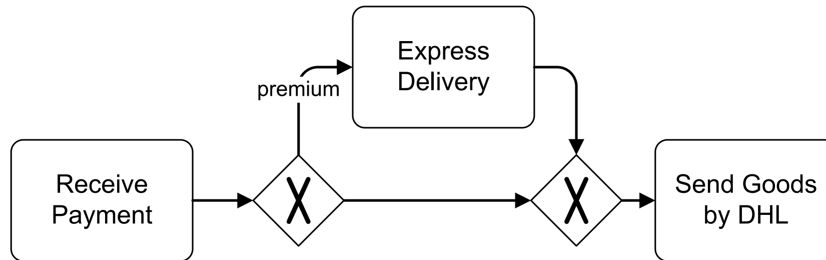


Figure 5.24: An inconsistent process fragment

Example 3. (Data Issues) The aforementioned aspects of process execution and contradictions are also of relevance for data-related rules. Consider a compliance rule $R5$ that requires a purchase request to be in state $archived$ when the case is closed, i.e., $\mathbf{G}(\mathbf{ready}(close) \rightarrow \mathbf{state}(purchase\ request, archived))$. Further on, a second rule $R6$ requires a purchase request to be either in state $accepted$, or $rejected$, i.e., $\mathbf{G}(\mathbf{ready}(close) \rightarrow \mathbf{state}(purchase\ request, accepted) \vee state(purchase\ request, rejected))$, when the case is about to be closed.

Not surprisingly, the Büchi automaton for the conjunction of these two rules, the automaton was omitted due to its size, has a lot of accepting runs, even though both rules are inconsistent. That results from the phenomena that have already been discussed above for activities. That is, accepting runs violate the requirement of exclusive data states, require data objects to be in no state, vacuous satisfiability as explained in Example 1 for activities, or are invalid from a business perspective, e.g, the state of the purchase request might be set to both, $accepted$ and $rejected$.

The above examples showed that domain knowledge have to be explicitly added to the conjunction of LTL formulas representing compliance rules in order to receive a correct decision about conflicts. Moreover, we need to enforce the Büchi automaton to behave in way that it executes the conditions of the compliance rules. This extra knowledge can be classified into:

- Separate execution states of activities;
- Separate data states of data objects;
- Enforcing non-vacuous satisfaction;
- Including knowledge about contradicting activities and data objects

The first three classes are generic and must be applied for any check for conflicts. On the other hand, the fourth class applicability depends on the rules to be checked. We formalize the knowledge in the above four classes as LTL formulas.

Definition 5.4. *[Separate execution states]* Any two activities a, b cannot complete their execution simultaneously, $\mathbf{G}(\neg(\mathbf{executed}(a) \wedge \mathbf{executed}(b)))$.

Definition 5.5 formalizes the nature of interleaving execution by not allowing a state where any two activities a and b can complete their execution. This can be applied also to the case of data object states, where a data object can assume only value at a time.

Definition 5.5. *[Separate data object states]* For any data object d having a finite set of states S_d, d can assume exactly one state at a time $\forall s \in S_d : G(\mathbf{s}tate(d, s) \rightarrow \bigwedge_{s' \in S_d \setminus \{s\}} \neg \mathbf{s}tate(d, s'))$

To enforce non-vacuous satisfaction, for each compliance rule on the form $p \rightarrow q$ we enforce the execution of p by adding $\mathbf{F}(p)$ to the conjunction.

Definition 5.6. *[Non Vacuous Satisfiability] A compliance rule r has to be non-vacuously satisfied by adding $\mathbf{F}(a)$ to the conjunction to be checked if and only if a appears in the condition of r.*

To reflect contradictions between activities, as can be derived from the domain knowledge, we add an LTL formula that insures that the execution of one activity a can never be followed by the execution of any of its contradicting activities.

Definition 5.7. *[Contradicting Activities] If two activities a and b are known to be contradicting, exclusive, from the domain knowledge, this is reflected as $\mathbf{G}(\textbf{\textit{executed}}(a) \rightarrow \mathbf{G}(\neg \textbf{\textit{executed}}(b)))$.*

The LTL formula from Definition 5.7 is added to the conjunction for each activity a that appears in the compliance rules and all of its contradicting activities as defined in the domain knowledge.

Theorem 5.2. *A set of activity execution ordering compliance rules R are consistent under the domain knowledge rules D, where elements of both sets are LTL formulas, if and only if the Büchi automaton generated for their conjunction has at least one accepting run.*

Proof. If the generated Büchi automaton has an accepting run then there is a computation, sequence of execution, in which the formulas are satisfied. On the other hand, lack of any accepting runs implies the impossibility of finding a computation that satisfies the conjunction of formulas. □

Theorem 5.2 implies that the decision about conflict freedom is bound to the availability of sufficient domain knowledge. That is, for instance, if we lack the knowledge about contradiction between two activities, we can decide conflict freedom of two rules that require them to execute non-exclusively. Still, the approach of utilizing Büchi automata may help construct the domain knowledge. This could be achieved by synthesizing process fragments out of the Büchi automaton and negotiating whether it is acceptable with domain experts and business analysts. At that point, domain experts are able to identify conflicts and the domain knowledge must be updated. The synthesis of process fragments is out of scope of the thesis.

5.4 Checking Rules Against Processes

In this section we discuss in detail the checking of a compliance rule, a BPMN-Q behavioral query, against a process model. According to our approach, see Section 2.3, the first step to checking is to identify a set of process models that are related to the rule. This is done by means of tagging, see Section 4.2. In general, we can decide on compliance of a process model to a rule, query, by reducing the compliance checking to a model checking problem. As was discussed in Section 4.5, a model checker receives two inputs; the Kripke structure, which is a special type of annotated automaton and the temporal logic formula to be checked against the Kripke structure. However, model checking is known to suffer from state space explosion [24]. To work around this problem, model checkers apply advanced techniques to reduce the state space [60], use less memory to do the reachability analysis [132], or use symbolic model checking approaches [88]. These are considered as domain-independent approaches to reduce the complexity of model checking. However, it is also possible to use the domain-specific knowledge to simplify the problem more, or even avoid the model checking totally. Figure 5.25 summarizes our approach for compliance checking.

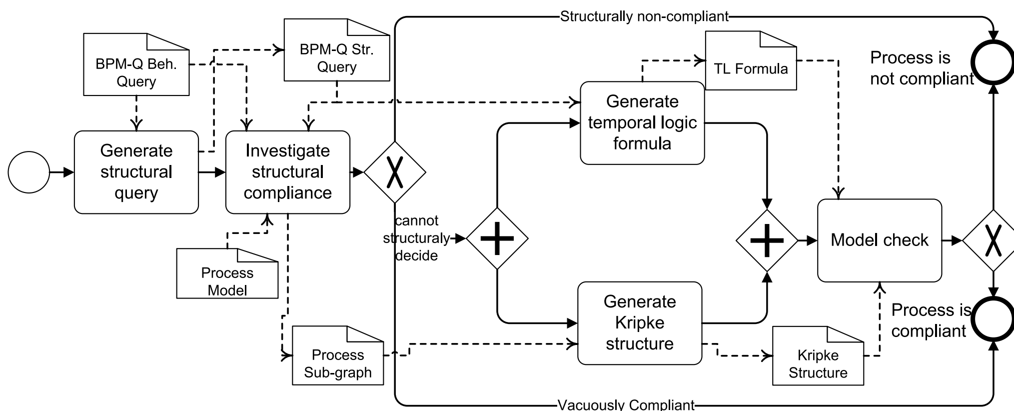


Figure 5.25: The checking process of compliance rules against process models

Our compliance checking process starts with the compliance query and the process model as input. In the first step, we generate so-called implicit structural query from the behavioral query. Next, we try to early decide about compliance by investigating some structural aspects of the process based on both behavioral and structural queries. As a result of this step we might decide structural non-compliance, vacuous compliance, or we are unable to decide. In the latter case, we have to do behavioral analysis, i.e., model checking. To do model checking, we generate the Kripke structure from the process (cf. Section 5.4.3). Meanwhile, the temporal logic formula is derived from the compliance query as discussed in Section 5.2.

5.4.1 Implicit Structural Queries

A behavioral BPMN-Q query contains two pieces of information. Firstly, it contains a structural query, the nodes and edges. This structural query will be used to check whether we can structurally decide about compliance, non-compliance. Secondly, it contains the temporal logic formula that would be model checked against the process. We showed how behavioral queries map to temporal logic formulas in Section 5.2. Also, we discussed in Section 4.2 how a structural query is matched to a process model. Here, we need to show that obtaining a structural query from a behavioral query is not straightforward.

All compliance rules discussed earlier describe ordering between nodes in a process. For instance, global-scope presence requires that certain activity has to occur somewhere after the start of the process. Thus, finding a path , sequence of nodes, between the source and target nodes provides the first step to check for compliance, by considering execution semantics of those nodes in between. This means, if we cannot find any paths between the source and target nodes, we can structurally decide about non compliance, since no execution ordering is guaranteed.

To obtain a structural query from a well formed behavioral one, we use the same set of nodes as in the behavioral query. All paths specified in the behavioral query are mapped to similar structural paths, i.e., without the behavioral stereotypes. If a path in the behavioral query has an exclude property, this is neglected in the structural counterpart. Similarly, behavioral data flow edges are dropped since they cannot be decided structurally.

5.4.2 Structural Investigation about Compliance

Up to this point, we have three pieces of information in hand, the process model, the behavioral query, and the implicit structural queries. In this step we want to run structural investigations to see whether we can early decide about compliance.

Depending on the rule r, we can decide on the compliance of a process model p to r. For instance, the case of global-presence, see Section 5.2.3.1, a process p can be judged as structurally non-compliant if it lacks any occurrence of the activity a specified in r. Conversely, is the case of global-absence.

Another example is the after-scope presence pattern, cf. Section 5.2.3.5. Consider r to be $\mathbf{G}(\textbf{executed}(a) \rightarrow \mathbf{F}(\textbf{executed}(b))$. In this case a process model p is structurally non-compliant if it has an occurrence of a and:

- It has no occurrence of b, or,
- It has an occurrence of b but with no execution path from a to b

Such structural investigations can be done efficiently. Checking of activity occurrences can be done in linear time to the number of activities in a process. Checking execution paths can be achieved by matching the structural query to the process. This is achieved also in a low polynomial time. We can decide about non-compliance due to lack of execution paths if the matching returns an empty sub-graph. If the matching sub-graph

is not empty, we cannot structurally decide about compliance and model checking has to take place.

Definition 5.8. *[Structural non-compliance] A process model* $\mathcal{P} = (\mathcal{A}, \mathcal{E}, \mathcal{D}, \mathcal{G}, \mathcal{S}, \mathcal{D}ata\mathcal{S}tate, \mathcal{CF}, \mathcal{DF}, \mathcal{CFC}, \mathcal{ID})$ *is* structurally non compliant *to a BPMN-Q behavioral query* $\mathcal{Q} = (\mathcal{A}_\mathcal{Q}, \mathcal{E}_\mathcal{Q}, \mathcal{D}_\mathcal{Q}, \mathcal{S}_\mathcal{Q}, \mathcal{D}ata\mathcal{S}tate_\mathcal{Q}, \mathcal{P}_\mathcal{Q}, \mathcal{DF}_\mathcal{Q}, isAnonymous, \mathcal{X}, \mathcal{TYPE})$ *if:*

- $\exists (x, y) \in \mathcal{P}_\mathcal{Q} : \mathcal{TYPE}((x, y)) = leadsto \ \wedge isAnonymous(y) = false \nexists z \in (\mathcal{A} \cup \mathcal{E}) : label(y) = label(z)$. No occurrences of target nodes for the *leads to* paths in the investigated process,
- $\exists (x, y) \in \mathcal{P}_\mathcal{Q} \wedge \mathcal{TYPE}((x, y)) = precedes \ \wedge isAnonymous(x) = false \ \nexists z \in (\mathcal{A} \cup \mathcal{E}) : label(x) = label(z)$. No occurrences of source nodes for the *precedes* paths in the investigated process,
- The implied structural query has no matches within the process.

Another interesting issue is the ability to decide about *vacuous* compliance. An example situation for vacuous compliance is the rule r above. A process model p vacuously satisfies this rule if it lacks any occurrence of activity a. Reporting vacuous compliance may be important for experts so that they update their rules or have further investigations about the process [28]. Of course, vacuous compliance can be detected via model checkers. However, this calls for generating specific vacuity compliance testing formulas, which will be discussed later.

Definition 5.9. *[Structural vacuous compliance] A process model* $\mathcal{P} = (\mathcal{A}, \mathcal{E}, \mathcal{D}, \mathcal{G}, \mathcal{S}, \mathcal{D}ata\mathcal{S}tate, \mathcal{CF}, \mathcal{DF}, \mathcal{CFC}, \mathcal{ID})$ is structurally vacuous compliant *to a BPMN-Q behavioral query* $\mathcal{Q} = (\mathcal{A}_\mathcal{Q}, \mathcal{E}_\mathcal{Q}, \mathcal{D}_\mathcal{Q}, \mathcal{S}_\mathcal{Q}, \mathcal{D}ata\mathcal{S}tate_\mathcal{Q}, \mathcal{P}_\mathcal{Q}, \mathcal{DF}_\mathcal{Q}, isAnonymous, \mathcal{X}, \mathcal{TYPE})$ *if:*

- $\forall (x, y) \in \mathcal{P}_\mathcal{Q} : \mathcal{TYPE}((x, y)) = leadsto \ \wedge isAnonymous(y) = false \nexists z \in (\mathcal{A} \cup \mathcal{E}) : label(x) = label(z)$. No occurrences of source nodes for the *leads to* paths in the investigated process model,
- $\forall (x, y) \in \mathcal{P}_\mathcal{Q} : \mathcal{TYPE}((x, y)) = precedes \ \wedge isAnonymous(y) = false \nexists z \in (\mathcal{A} \cup \mathcal{E}) : label(y) = label(z)$. No occurrences of target nodes for the *precedes* paths in the investigated process model.

5.4.3 Model Checking Rules

When it is not possible to structurally decide about compliance/non-compliance, we have to do an exhaustive state space analysis, i.e., model checking. As was discussed in Section 4.5, we need two inputs; the temporal logic formula to be checked, the compliance rule in our case, and the system to check, the process model.

In Section 5.2 we showed how to obtain for each compliance rule, pattern, an equivalent temporal logic formula. In this section, we describe how to derive the Kripke structure, cf. Definition 4.11, from process models.

In Section 4.1.3.3, we discussed how to derive a Petri net from a well-formed process model. In this section we build on that and show how to derive a Kripke structure $\mathcal{M} = (S, s_i, R, L)$ from the reachability graph of a Petri net.

Definition 5.10. *[Reachability Graph]* The reachability graph *of a Petri net $PN = [P, T, F, m_0]$ is a tuple $RG = (V, TR, v_0)$ where V is the set of states, markings. $TR \subseteq V \times V$ is the transition relation between states, caused by firing transitions in T. v_0 is the initial state of RG which is equal to m_0.*

The Petri net $PN = [P, T, F, m_0]$ obtained from a process model $\mathcal{P} = (\mathcal{A}, \mathcal{E}, \mathcal{D}, \mathcal{G}, \mathcal{S}, \mathcal{D}ata\mathcal{S}tate, \mathcal{CF}, \mathcal{DF}, \mathcal{CFC}, \mathcal{ID})$ according to Definition 4.4 already determines the initial marking of the net. Thus, we can compute the reachability graph RG of the net using the firing of transitions [113]. Each time a transition fires, the state of the net changes, the marking of places. For our case we are interested in the marking of three different types of places. Firstly, data places which correspond to the data value(state) assumed by a data object. In Section 4.1.3.2 we showed that a data object is mapped to a set of places corresponding to its set of states. Thus, each time a data place, say $dob1_state3$, contains a token; the predicate **state**$(dob1, state3)$ holds. The other two types of places correspond to control flow. Since any activity is assigned exactly one input control flow place and one output control flow place, we can determine whether an activity a is ready,**ready**(a), or executed, **executed**(a), in any Petri net marking m_j if $p_i \in {}^{\bullet}t_a \wedge m_j(p_i) > 0$ or $p_o \in {}^{\bullet}t_a \wedge m_j(p_o) > 0$ respectively.

The proposition **start** holds in the initial marking where the input place for the transition t_s, corresponding to the start event of the process, has a token. Similarly, the proposition **end** holds when the output place of transition t_e, corresponding to the end event of the process, has a token, indicating the termination of execution.

The set of atomic propositions AP is constructed as follows

Definition 5.11. *[Atomic propositions for compliance checking]* The set of atomic propositions AP is constructed from a process model $\mathcal{P} = (\mathcal{A}, \mathcal{E}, \mathcal{D}, \mathcal{G}, \mathcal{S}, \mathcal{D}ata\mathcal{S}tate, \mathcal{CF}, \mathcal{DF}, \mathcal{CFC}, \mathcal{ID})$ as follows*:* $AP = \{ready_a : a \in \mathcal{A}\} \cup \{executed_a : a \in \mathcal{A}\} \cup \{state_d_s : (d, s) \in \mathcal{D}ata\mathcal{S}tate\} \cup \{start, end\}$

Definition 5.12. *[Generating Kripke structure from process models]* A Kripke structure $\mathcal{M} = (S, s_i, R, L)$ can be obtained from the reachability graph $RG = (V, TR, v_0)$ of a Petri net $PN = [P, T, F, m_0]$ generated from a process $\mathcal{P} = (\mathcal{A}, \mathcal{E}, \mathcal{D}, \mathcal{G}, \mathcal{S}, \mathcal{D}ata\mathcal{S}tate, \mathcal{CF}, \mathcal{DF}, \mathcal{CFC}, \mathcal{ID})$ with respect to a set of atomic propositions AP as follows*:*

- $S = V$*,*

- $s_i = v_0$,
- $R = TR \cup \{(v, v) : v \in V \wedge \nexists r \in V : (v, r) \in TR\}$,
- $\forall s \in S$:
  - $start \in L(s)$ *if and only if* $s = s_i$. *The* start proposition holds in the initial state of the execution,
  - $end \in L(s)$ *if and only if* $\nexists (s, _) \in TR$. *The* end proposition holds for states in the reachability graph with no outgoing edges to other states,
  - $ready_a \in L(s)$ *if and only if* $m_s(p_{(_,a)}) = 1 \wedge p_{(_,a)} \in {}^{\bullet}t_a$ *where* $a \in \mathcal{A}$. The ready proposition holds in state s for each Petri net transition, representing a process activity, having its input control flow place marked,
  - $executed_a \in L(s)$ *if and only if* $m_s(p_{(a,_)}) = 1 \wedge p_{(_,a)} \in t_a{}^{\bullet}$ *where* $a \in \mathcal{A}$. The executed proposition holds in state s for each Petri net transition, representing a process activity, having its output control flow place marked,
  - $state(data, state) \in L(s)$ *if and only if* $m_s(p_{(data,state)}) = 1$ *where* $(data, state) \in \mathcal{D}ata\mathcal{S}tate$. The proposition $state(data, value)$ holds in each execution state s where its corresponding place is marked.

After obtaining the Kripke structure, the compliance rule can be model checked against it. Also, we can run a vacuous compliance test. In Section 5.4.2, we discussed how vacuous compliance can be checked on the structure of the business process, cf Definition 5.9. However, when it comes to data and conditional compliance rules, it might not be possible to decide vacuous compliance on a process structure. Thus, we need to run this check against the Kripke structure obtained from the process model.

A vacuous compliance check simply tests whether a compliance rule's condition will ever be enabled during the execution of a process. That is, we need to find at least one execution state in which the condition part of the rule will be true. For instance, if we take a conditional after-scope presence rule on the form $\mathbf{AG}(\textbf{state}(d1, s1) \wedge \textbf{state}(d2, s3) \rightarrow \mathbf{AF}(\textbf{executed}(act1)))$, a vacuous compliance check would be $\mathbf{EF}(\textbf{state}(d1, s1) \wedge \textbf{state}(d2, s3))$. We can notice that CTL not LTL is capable of expressing vacuous compliance checking due to its support to existential quantifiers over execution paths.

From a processing cost point of view, the checking for vacuous compliance is, theoretically, much cheaper than checking the rule itself, due to its existential nature. Thus, once a vacuous compliance checking fails, we do not need to check the original rule itself as we know it is vacuously satisfied.

5.5 Example

This section introduces a process model along with a set of compliance requirements related to this process model in order to illustrate the applicability of our approach. We

focus on a process model from the financial domain, Figure 5.26 shows the process of opening a correspondent bank account, expressed in BPMN.

This correspondent account is a type of bank accounts that is opened by a bank (respondent bank), in some country, in another bank, in another country, to ease and speed the operation of money transfer. The process starts with “Receive correspondent Account open request” to open an account. Bank Identity looked up in activity “Identify Respondent Bank” in order to go on with the procedure of opening the account . If this is the first time such respondent bank requests to open an account, a new record for that bank is *created* and some checks must take place. The bank to open the account needs to conduct a study about the respondent bank due diligence “Conduct due diligence study” where the Respondent bank may *pass* of *fail* this study. In case the respondent bank fails the evaluation of due diligence, the bank inquires one of its partner banks about the respondent bank then there is a decision made whether to make an extra study, activity “Lookup Partner Banks”. If the decision is that it deserves an extra evaluation, the process loops; otherwise the process proceeds. It is also needed to assess the risk of opening an account for that Respondent Bank “Assess Respondent Bank risk” with the resulting risk assessment categorized as either *high* or *low*. In the mean time, respondent bank certificate is checked for *validity* in order to proceed with opening the account. If the due diligence study evaluation fails, the Respondent Bank is added to a black list. On the other hand, if such respondent bank has a record with the bank, these checks are skipped. In any of the cases, the bank has to obtain a report about the performance of the Respondent Bank “Obtain Respondent Bank Annual Report”. This report is analyzed by the Bank “Analyze Respondent Bank annual report”, and the Respondent Bank rate is reviewed “Review Respondent Bank rating”. If the respondent bank passes the checks, i.e., it passes the due diligence evaluation and its rating is accepted or there is already a record for the respondent bank an account is opened “Open Correspondent Account”.

The process is subject to the following compliance requirements [25]:

- $R1$: We have to obtain and analyse the respondent bank report.
- $R2$: If the respondent bank evaluation fails, it must be added to a black list.
- $R3$: Opening an account must be of low risk.
- $R4$: Before opening an account, the respondent bank rating must have been accepted.
- $R5$: If it is the first time to deal with the respondent bank, advanced due diligence study must be conducted.
- $R6$: If the respondent bank rating is rejected, an account must never be opened.

In the following we utilize the patterns described in Section 5.2 to express these compliance requirements. Afterwards, we check their consistency and report the compliance status of the process in Figure 5.26 to them.

Figure 5.26: A process model to open a bank account

$R1$: We have to obtain and analyse the respondent bank report

$R1$ requires both activities "Obtain Respondent Bank Annual Report", "Analyze Respondent Bank Annual Report" to occur in the process model. Moreover, it might require order between them.[‡] Thus, we represent these requirements as shown in Figure 5.27

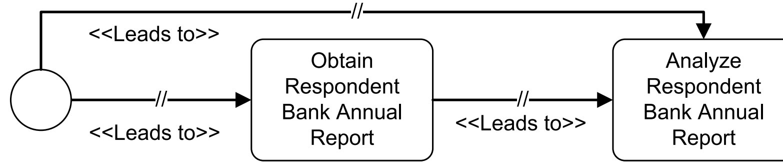


Figure 5.27: Representation of $R1$ in BPMN-Q

$R1$ was represented by using global-scope presence and after-scope presence patterns. The corresponding PLTL formula is

$$\begin{aligned}&\mathbf{G}(\mathbf{start} \rightarrow \mathbf{F}(\mathbf{executed}(Obtain\ Respondent\ Bank\ Annual\ Report)))\wedge\\&\mathbf{G}(\mathbf{start} \rightarrow \mathbf{F}(\mathbf{executed}(Analyze\ Respondent\ Bank\ Annual\ Report)))\wedge\\&\mathbf{G}(\mathbf{executed}(Obtain\ Respondent\ Bank\ Annual\ Report) \rightarrow\\&\mathbf{F}(\mathbf{executed}(Analyze\ Respondent\ Bank\ Annual\ Report)))\end{aligned}$$

$R2$: If the respondent bank evaluation fails, it must be added to a black list

$R2$ can be represented by a conditional response pattern as shown in Figure 5.28 where activity "Add Respondent Bank to Black List" is the response to the condition that "Evaluation" *fails*.

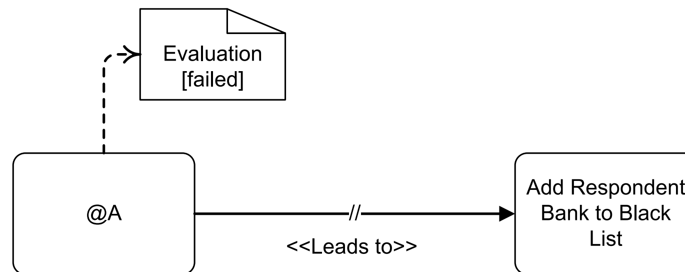


Figure 5.28: Representation of $R2$ in BPMN-Q

The PLTL formula for $R2$ is

$$\begin{aligned}&\mathbf{G}(\mathbf{executed}(Conduct\ due\ diligence\ study) \wedge \mathbf{state}(Evaluation, failed)\wedge\\&\mathbf{G}(\neg\mathbf{state}(Evaluation, passed)) \rightarrow \mathbf{F}(\mathbf{executed}(Add\ Respondent\ Bank\ to\ Black\ list)))\end{aligned}$$

[‡]The interpretation of the informal requirements, e.g., legislation text, into formal requirements is out of scope of this thesis.

$R3$: **Opening an account must be of low risk**

This rule can be represented as a data flow rule, see Section 5.2.4. The rule above is expressed as a BPMN-Q query as shown in Figure 5.29 where the requirement that “Risk” data object must be in state low when the “Open Correspondent Account” activity is about to execute, is represented by a behavioral association between the data object and the activity.

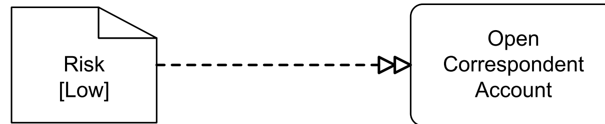


Figure 5.29: Representation of $R3$ in BPMN-Q

The PLTL formula for $R3$ is

$$\mathbf{G}(\mathbf{ready}(Open\ Correspondent\ Account) \rightarrow \mathbf{state}(Risk, low))$$

$R4$: **Before opening an account, the respondent bank rating must have been accepted**

We can express this rule using the conditional precedence pattern discussed in Section 5.2.5.3. The BPMN-Q compliance query for this rule is shown in Figure 5.30.

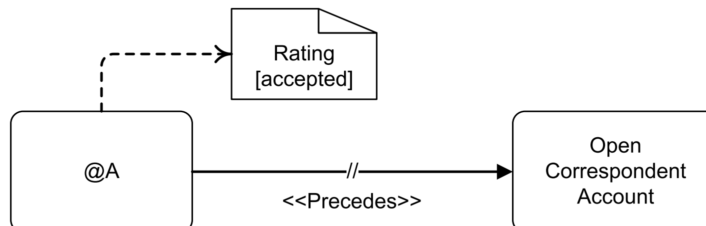


Figure 5.30: Representation of $R4$ in BPMN-Q

The PLTL formula for that rule is

$$\mathbf{G}(\mathbf{ready}(Open\ Correspondent\ Account) \rightarrow \\ \mathbf{O}(\mathbf{state}(Rating, accepted) \wedge \mathbf{G}(\neg\mathbf{state}(Rating, rejected))))$$

$R5$: **If it is the first time to deal with the respondent bank, advanced due diligence study must be conducted**

This rule can be also represented as a conditional response pattern. We can reflect the requirement of being the first time to deal with the bank using the state *created* of the “R Bank Record” data object.

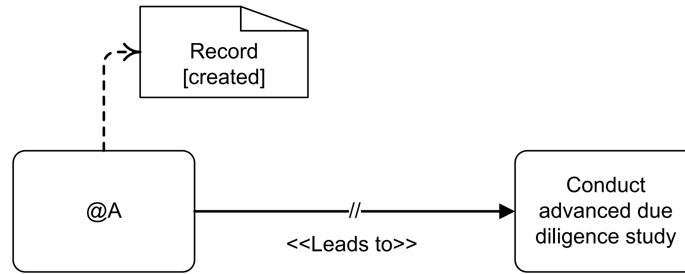


Figure 5.31: Representation of $R5$ in BPMN-Q

The PLTL formula for $R5$ is

$$\mathbf{G}(\mathbf{executed}(Identify\ respondent\ bank) \wedge \mathbf{state}(Record, created) \rightarrow \mathbf{F}(\mathbf{executed}(Conduct\ advanced\ due\ diligence\ study)))$$

$R6$: If the respondent bank rating is rejected, an account must never be opened

This rule can be modeled using the conditional after-scope absence pattern, see Section 5.2.5.5. Figure 5.32 realizes this rule as a BPMN-Q query.

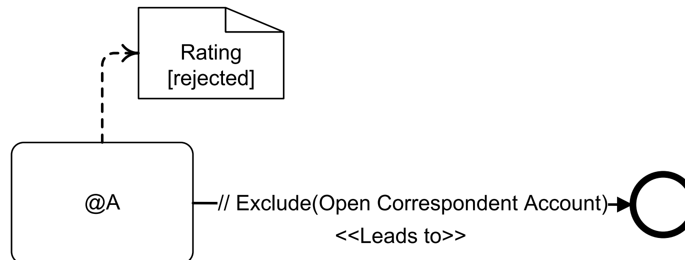


Figure 5.32: Representation of $R6$ in BPMN-Q

$R6$ PLTL formula is

$$\mathbf{G}(\mathbf{executed}(Review\ respondent\ bank\ rating) \wedge \mathbf{state}(Rating, rejected) \wedge \mathbf{G}(\neg\mathbf{state}(Rating, accepted)) \rightarrow \neg\mathbf{executed}(Open\ Correspondent\ Account)\ \mathbf{U}\ \mathbf{end})$$

Checking Rules Consistency

It is obvious that rule $R1$ by its own contains a transitive redundancy. However, as we discussed earlier, this type of redundancy can only be determined at checking time. Moreover, rules $R4$ and $R6$ contain a contrapositive redundancy, see Section 5.3.1. Thus, we can arbitrarily drop any of them from the checking, we drop rule $R4$.

To check for conflicts, we add the extra domain knowledge, as was discussed in Section 5.3.2, to the conjunction of rules $R1$-$R3$,$R5$-$R6$. Because of space, we give symbols for the different activities as follows:

- ObRe : Obtain Respondent Bank Annual Report
- AnRe : Analyze Respondent Bank Annual Report
- ADBL : Add Respondent Bank to Black List
- OCA : Open Correspondent Account
- CdD : Conduct due diligence study
- CAdD : Conduct Advanced due diligence study
- RRBR : Review Respondent Bank Rating
- Eval_failed: (Evaluation,failed)
- Eval_passed: (Evaluation,passed)
- R_Bank_Rec_created: (R Bank Record,created)
- Rate_rejected: (Rating,rejected)
- Rate_accepted: (Rating,accepted)

To check for consistency, the extra formulas to be added to the conjunction are

$\mathbf{G}(\neg(\textbf{executed}(ObRe) \wedge \textbf{executed}(AnRe)))$
$Separate\ execution\ states$

$\mathbf{G}(\neg(\textbf{executed}(ObRe) \wedge \textbf{executed}(ADBL)))$
$Separate\ execution\ states$

$\vdots$

$\mathbf{G}(\neg(\textbf{executed}(OCA) \wedge \textbf{executed}(CAdD)))$
$Separate\ execution\ states$

$\mathbf{G}(\neg(\textbf{state}(Eval_failed) \wedge \textbf{state}(Eval_passed)))$
$Separate\ execution\ states$

$\mathbf{G}(\neg(\textbf{state}(Rate_rejected) \wedge \textbf{state}(Rate_accepted)))$
$Separate\ execution\ states$

$\mathbf{F}(\textbf{executed}(CdD)\wedge$
$\textbf{state}(Eval_failed) \wedge \mathbf{G}(\neg\textbf{state}(Eval_passed)))$
$Non\ vacuous\ statisfiability$

$\mathbf{F}(ready(OCA))$ $Non\ vacuous\ statisfiability$

$\mathbf{F}(\textbf{state}(R_Bank_Rec_created))$
$Non\ vacuous\ statisfiability$

$\mathbf{F}(\textbf{executed}(RRBR) \wedge$
$\textbf{state}(Rate_rejected) \wedge \mathbf{G}(\neg\textbf{state}(Rate_accepted)))$
$Non\ vacuous\ statisfiability$

$\mathbf{G}(\textbf{state}(Eval_failed) \rightarrow \mathbf{G}(\neg\textbf{state}(Eval_passed)))$
$Contradicting\ states$

$\mathbf{G}(\textbf{state}(Eval_passed) \rightarrow \mathbf{G}(\neg\textbf{state}(Eval_failed)))$
$Contradicting\ states$

$\mathbf{G}(\textbf{state}(Rate_rejected) \rightarrow \mathbf{G}(\neg\textbf{state}(Rate_accepted)))$
$Contradicting\ states$

$\mathbf{G}(\textbf{state}(Rate_accepted) \rightarrow \mathbf{G}(\neg\textbf{state}(Rate_rejected)))$
$Contradicting\ states$

Checking consistency of the whole conjunction yielded a Büchi automaton with accepting states. Thus, there are no conflicts among the rules.

Checking Rules Against Process

Once we are sure that the rule set is consistent, we can proceed with checking the rules against the process. We can decide that $R5$ is structurally non-compliant (cf. Definition 5.8) since the process model in Figure 5.26 has no occurrence of activity "Conduct advanced due diligence study" at all. For the other rules $R1$-$R3$ and $R6$ we cannot structurally decide about either non-compliance or vacuous compliance. Thus, we have to model check them against the process.

The result of model checking is

- $R1$ is satisfied.
- $R2$ is satisfied.
- $R3$ is not satisfied.
- $R6$ is satisfied.

5.6 Summary & Outlook

In this chapter we discussed how compliance rules are modeled using a pattern-based approach. Each pattern is visualized as a BPMN-Q behavioral query; each pattern has a counterpart temporal logic formula. Patterns were categorized as control flow patterns, data flow patterns and conditional patterns. For (conditional) control flow patterns, we showed how can presence, absence, and execution ordering relations could be expressed. Conditional control flow patterns provided more expressiveness and capability to address cases that were not addressable using pure control flow patterns, see Section 5.2. To formalize patterns, we provided mappings to LTL and CTL. One reason for these two mappings is to allow the choice among a wider range of model checker, depending on the temporal logic they support. Nevertheless, there is another pragmatic justification. We benefit from LTL model checking techniques to check conflicts. The use of CTL will be explained more in the next chapter.

We addressed consistency checking among rules, see Section 5.3. A consistent set of rules should have no redundancies and must have no conflicts among its elements. We were interested in discovering types of redundancy, duplicate, mutual exclusion or contrapositive redundancy, that could be discovered before checking rules against processes. For the case of conflict checking, we showed that execution environment assumptions and domain-specific knowledge inclusion was necessary to correctly decide. Finally, we showed how rules can be checked against process models, see Section 5.4. We showed that for some rules, we can structurally decide about compliance. If it is not possible to structurally decide; we go for model checking. Also, we pointed the importance of reporting about vacuous compliance. This is the situation where a compliance rule is satisfied by a process model when the rule's condition never holds.

In Section 5.5, we had a case study where we modeled and checked a set of anti money laundering compliance rules.

In the next chapter, we will describe an approach to provide useful feedback, explanation, to the user in case a process model violates rule(s).

Chapter 6

Explaining Compliance Rules Violations

So far, compliance rules addressing control and data flow aspects of process models have been modeled and verified. In general, when a property is not satisfied by the system under investigation, the model checker generates a counterexample. Such a counterexample is a sequence of states which shows *how* the property was violated.

This is another interesting question; how can we explain violations to the user? A starting point could be using the counterexample generated from the model checker (cf. [38]). The drawbacks of such an approach are manifold. Firstly, the generated counterexamples are given in terms of transitions, i.e., the sequence of states in the investigated finite state machine. This endures a cost of re-translating such traces backwards into the structure of process models, it might not always be possible to generate meaningful representation that the user can easily understand, e.g., the handling of parallel threads. Secondly, the generated counterexample is not exhaustive, i.e., not every possible violation is detected by the model checker. Rather, only the first met violation is reported. Thirdly, the translation will be dependent on both the output of the model checker as well as the visual notation the user understands. Each time the model checker software is changed; the translation component has to be adapted. Finally, in the previous chapter, we showed that we can structurally decide about non-compliance, in this case, there is no counterexample generated at all.

Our objective in this chapter is to provide the user with a useful feedback in the form of parts of the process model whose execution causes violation to a certain rule. By locating parts of the process whose execution causes the violation, we save users effort to locate such causes and also avoid the threat of missing an unanticipated violation scenario.

We will show in the rest of this chapter that violating scenarios can be declaratively described as BPMN-Q structural queries. We call queries describing violation scenarios,

anti-pattern queries. For each compliance pattern we derive anti-pattern query(ies) describing the violation. Based on the pattern, derivation of the anti pattern queries might examine process-specific conditions, via temporal logic querying.

We start by describing our approach of deriving anti patterns in Section 6.1. Explaining control flow violations comes in Section 6.2. Explaining violations to data flow rules is covered in Section 6.3. Section 6.4 covers the explanation of conditional rules violation. Section 6.5 shows how we can simplify the complexity of evaluating the temporal logic queries using domain-specific knowledge. Limitations of our approach to explain violations is discussed in Section 6.6. Section 6.7 builds on the example we introduced in Section 5.5. Finally, Section 6.8 summarizes this chapter and links to the next chapter.

6.1 Deriving Anti Patterns

The question of *how* violation occurred needs to be explained to the user in a *comprehensible* way. Our approach to answer such a question depends on analyzing the temporal logic formulas corresponding to the various compliance patterns discussed in Chapter 5. For each pattern, the possible causes of violations are derived by negating the CTL formula of the pattern. For each possible violation scenario a structural BPMN-Q query is developed. Such queries are called anti pattern queries. The reason of this naming is that these queries represent the *unwanted* behavior, i.e., behavior that if exposed by the process the violation occurs . The use of BPMN-Q queries visualizes the violation scenario to the user on a process model level, rather than the underlying transition system, by highlighting parts in the model whose execution causes the violation. Depending on the type of compliance rules being either control flow, data flow, or conditional rules, the generation of such anti pattern queries might investigate the behavioral model of a process, via temporal logic querying.

The generation of anti pattern queries starts only after the process model is known to be violating the rule. For each anti pattern we provide a CTL formula that formalizes the anti pattern. This anti pattern formula has two roles. First, it guides the design of the anti pattern query as it declaratively describes the violation. Second, in some cases, as will be discussed later, there might be more than one possibility to violate a compliance rule. In such cases, we have to model check the separate causes of violation in order to match the correct anti pattern query to the process to locate the error.

6.2 Control Flow Rules Violations

6.2.1 Global-scope Violation

The violation to global-scope presence occurs whenever there is a chance not to execute the to-be-present activity. Formally, the violation can be described by negating Formula 5.2. The negated CTL formula which describes the violation for global-scope

presence is

$$\neg\mathbf{AG}(\textbf{start} \rightarrow \mathbf{AF}(\textbf{executed}(a)))$$

Using the temporal logic equivalences discussed in Section 4.5.2, the formula looks like

$$\mathbf{EF}(\textbf{start} \wedge \mathbf{EG}(\neg\textbf{executed}(a))) \tag{6.1}$$

Formula 6.1 declaratively describes the violation scenario by finding at least one execution path where the process starts and in *some* possible following path there is no chance to execute A,$\wedge\mathbf{EG}(\neg\textbf{executed}(a))$, at all. We can notice the *existential* nature of violation scenarios.

We can express this violation via a structural BPMN-Q query. The query in Figure 6.1 describes the violation scenario by finding an execution path from the start of the process to its end without visiting activity A. Here, the *exclude* property of the path edge is set to A. All process models that are related to the compliance rule and lack any occurrences of A would be matched to that anti pattern, recall structural non-compliance (cf. Definition 5.8). Also, any related process model that has a chance to skip the execution of any of the occurrences of activity A is matched.

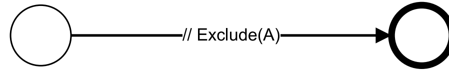


Figure 6.1: Global-scope presence anti pattern

The case of global-scope absence violation occurs when there is a chance to execute the to-be-absent activity. That is, the investigated process model has at least one occurrence of the to-be-absent activity. Formally, the anti pattern CTL formula is

$$\mathbf{EF}(\textbf{start} \wedge (\mathbf{E}[\neg\textbf{end}\ \mathbf{U}\ (\textbf{executed}(a) \wedge \neg\textbf{end})] \vee \mathbf{EG}(\neg\textbf{end}))) \tag{6.2}$$

Based on Formula 6.2, the violation could occur in one of two cases. The first case is the chance to execute A before the end of the process, $\mathbf{E}[\neg\textbf{end}\ \mathbf{U}\ (\textbf{executed}(a) \wedge \neg\textbf{end})]$. The second case is not to terminate the process, $\mathbf{EG}(\neg\textbf{end})$, at all. For the process models we consider, well formed processes, we know that the latter case cannot occur, processes have to terminate. Thus, the only possible violation is that there is a chance to execute the to-be-absent activity. That is, there is an execution path form the start of the process to an occurrence of the to-be-absent activity A. This is captured by the query in Figure 6.2.

6.2.2 Before-scope Violation

The before-scope presence pattern requires that an activity A is always executed before another activity B. So, the violation occurs when there is a chance to reach activity B without executing A at all before. Formally, the violation is defined as

$$\mathbf{E}[\neg\textbf{executed}(a)\ \mathbf{U}\ \textbf{ready}(b)] \tag{6.3}$$

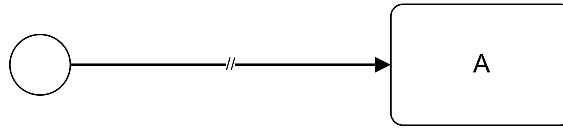


Figure 6.2: Global-scope absence anti pattern

To capture this violation on the process structure, we define the anti pattern query shown in Figure 6.3. The query declaratively describes the violation scenario where B is reachable from the start of the process, path edge, without visiting A, the exclude property.

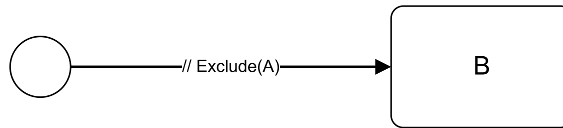


Figure 6.3: Before-scope presence anti pattern

On the other hand, violation to the before-scope absence pattern occurs when the to-be-absent activity A has the chance to execute before activity B. Formally, the violation is defined as.

$$\mathbf{EF}(\mathbf{start} \wedge \mathbf{EF}(\mathbf{executed}(a) \wedge \mathbf{EFready}(b))) \tag{6.4}$$

In Formula 6.4, the violation occurs when there is a chance to start the process, execute A and eventually reach activity B. However, the actual violation occurs whenever after execution of A we can reach B. This is the part of the anti pattern interesting to the violation scenario. The first interpretation for this formula is to find an execution path where A occurs and B occurs thereafter. This is correct on the behavioral level where the CTL formula fits. However, on the process structural level, there are other possibilities. The anti-pattern in Figure 6.4 is the first glance representation of the negated formula.

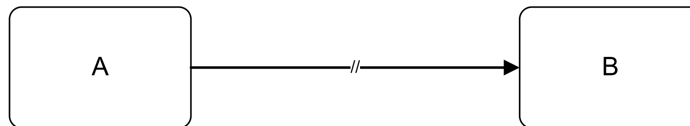


Figure 6.4: Incomplete before-scope absence anti pattern

However, this anti pattern does not find matches in the process of Figure 6.5. That is because the anti pattern assumes that there is a structural execution path between activity A and activity B. In that process, the concurrent execution of the two activities is also a violation, since they can be executed in an arbitrary order.

The anti-pattern of Figure 6.6 detects violations due to parallel execution of activities. With a path from an AND split to activity A, we declaratively describe that activity A

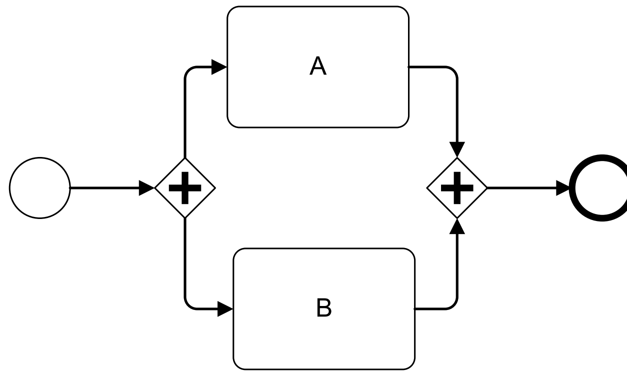


Figure 6.5: A sample non-compliant process

is not necessarily the very first activity to be executed on the parallel thread. Moreover, A might be nested in some arbitrary structure within the thread. With respect to the well-formed processes we address, these are the only two possibilities to violate the before-scope absence pattern on the model structure.

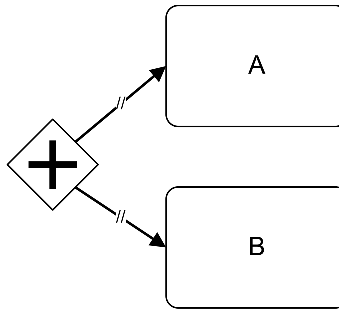


Figure 6.6: Another before-scope absence anti pattern

6.2.3 After-scope Violation

The violation for the after-scope presence takes place in a process instance when activity A executes but never B afterwards. The CTL formula for the after-scope presence anti pattern is:

$$\mathbf{EF}(\mathbf{executed}(a) \wedge \ \mathbf{EG}(\neg \mathbf{executed}(b))) \tag{6.5}$$

This violation is captured by the query in Figure 6.7 by looking an occurrence of activity A from which the end of the process is reached, the path edge, without executing B, the exclude property of the path edge.

On the other hand, the after-scope absence pattern is violated if there is a chance to execute activity B after activity A has been executed. Formally, the after-scope absence anti pattern is:

$$\mathbf{EF}(\mathbf{executed}(a) \wedge (\mathbf{E}[\neg \mathbf{end}\ \mathbf{U}\ (\mathbf{executed}(b) \wedge \neg \mathbf{end})] \vee \mathbf{EG}(\neg \mathbf{end}))) \tag{6.6}$$

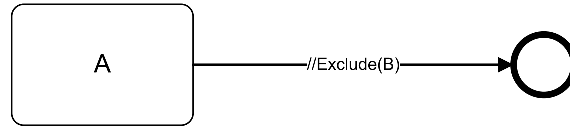


Figure 6.7: After-scope presence anti pattern

Following the same discussion for global-scope absence anti pattern, the formula above is similar to formula 6.2 where **start** is replaced with **executed**(a). However, to locate the violation on the process structure, we can reuse the anti pattern queries for before-scope absence in Figures 6.4, 6.6. In case that A and B are executed in parallel and/or B executes after A violation occurs.

6.2.4 Between-scope Violation

The violations of the between-scope absence patterns are variants of the anti patterns discussed above. The difference is in the upper and lower bounds of the scopes.

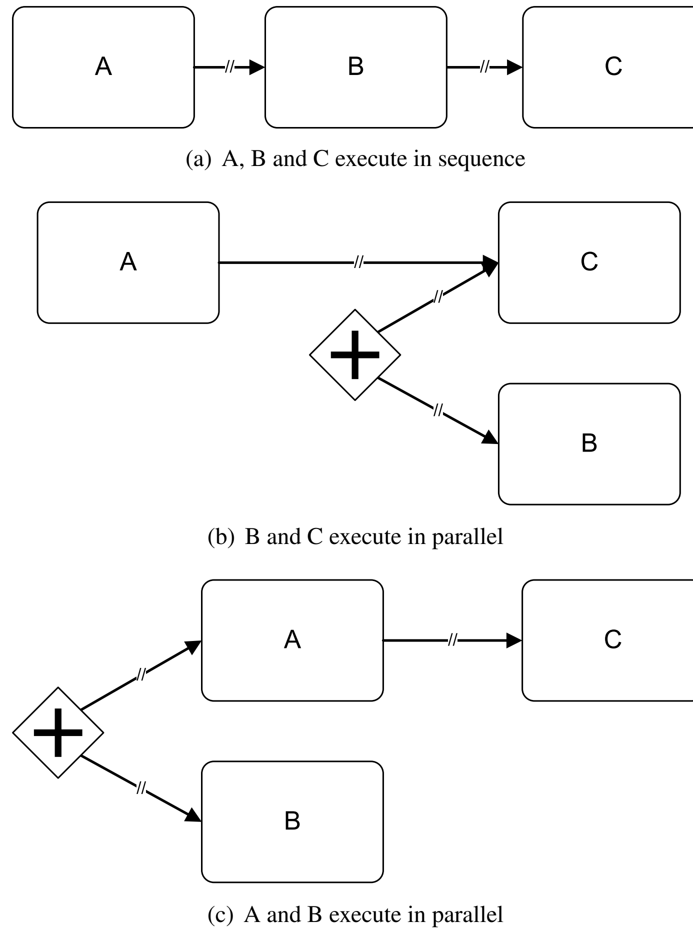
The between-scope absence type I violation occurs in one of two cases. Firstly, after executing activity A; activity B executes and activity C afterwards. The second possibility of violation is that activity C may never be executed after A. The anti pattern CTL formula for between-scope absence type I is:

$$\begin{aligned}&\mathbf{EF}(\mathbf{executed}(a) \wedge \mathbf{E}[\neg\mathbf{executed}(c) \ \mathbf{U} \ (\mathbf{executed}(b) \wedge \neg\mathbf{executed}(c))]) \vee \\ &\mathbf{EF}(\mathbf{executed}(a) \wedge \mathbf{EG}(\neg\mathbf{executed}(c)))\end{aligned} \tag{6.7}$$

The formula above is a variant of after-scope absence anti pattern, see Formula 6.6. However, the difference is that in the formula above, both cases of violations, the two disjuncts, are possible to occur.

The first possibility of violation is declaratively described as the case that activity A executes; afterwards B executes and finally C executes. This is clear on the execution traces of the process. However, there are many structural combinations of A, B, and C that are capable of generating that execution trace. At the first glance, activities A, B, and C execute in a sequence, not necessarily one after the other. However, having A and B executing in parallel, where C executes afterwards, also satisfies the first disjunct above. Also, having B and C executing in parallel, where A executes before, also satisfies the first disjunct above. Finally, having all of them execute in parallel also satisfies the first disjunct. However, having A and C executing in parallel satisfies the second disjunct above. Thus, we can exclude it from the causes of the first possibility at that point.

Anti pattern queries in Figure 6.8 detect the different cases of violation causing the behavior in the first disjunct. The anti pattern query in Figure 6.8(a) detects the case where activities A, B and C execute in sequence. Figure 6.8(b) depicts an anti pattern query that looks for a parallel execution between B and C. Meanwhile, activity A executes before C. Notice that this anti pattern query matches process models where activity A



(a) A, B and C execute in sequence

(b) B and C execute in parallel

(c) A and B execute in parallel

Figure 6.8: Between-scope absence type I anti patterns

executes either before the parallel threads or it belongs to the same thread as activity C. Finally, the anti pattern query in Figure 6.8(c) detects the violation where A and B run in parallel and C executes after A. The matches to the last two anti pattern queries are not disjoint. Moreover, the last anti pattern matches any process where C executes after joining all parallel threads or it executes within the same parallel thread of A.

The second possibility of violation is similar to the violation to after-scope presence. So, we can reuse the anti pattern query in Figure 6.7. Note that this anti pattern also matches processes where activities A and C execute in parallel.

In order to decide about which of the anti pattern queries to match to the process. We have to identify the exact cause of a violation. That is, whether the process has a violation due to executing B between A and C or because C is never executed after an activity A is executed. To determine this, we model check each disjunct in Formula 6.7 separately against the process. For each disjunct satisfied by the process, its corresponding anti pattern queries are matched to the process. This approach will be followed for all anti pattern formulas that describe several possibilities of violation.

The case of between-scope absence type II violation is similar. However, it looks at the history of execution. Thus, the violation of such rule occurs in two cases. The first case is similar to the one above, activity A and B execute before activity C executes. The second case of violation occurs when activity C might be reached without executing A before it. Formally, the between-scope absence type II anti pattern is expressed in CTL as

$$\begin{aligned}&\mathbf{EF}(\mathbf{executed}(a) \wedge \mathbf{EF}(\mathbf{executed}(b) \wedge \mathbf{EFready}(c)))\vee \\ &\mathbf{E}[\neg\mathbf{executed}(a)\ \mathbf{U}\ \mathbf{ready}(c)]\end{aligned} \tag{6.8}$$

The anti pattern queries in Figure 6.8 still fit to detect the first possibility of violation. For the second possibility of violation, we can use the anti pattern query for before-scope presence of Figure 6.3 replacing the activity B with C. Again, we have to model check each disjunct in Formula 6.8 to determine which anti pattern queries to be matched to the process.

6.3 Data Flow Rules Violations

One common feature of control flow anti patterns is that their derivation depends only on the compliance rule (pattern). For data flow rules and conditional rules, the derivation of anti pattern is more complicated. In many cases, we have to investigate the process model-specific data conditions that caused the violation. Thus, the derivation of anti patterns depends on both the rule and the investigated process model. To help discover violating data conditions, we depend on temporal logic queries, see Section 4.5.3. Afterwards, based on violating data conditions, we formulate BPMN-Q anti pattern queries that would capture execution paths of the process that cause the violation.

In the rest of this section we focus on deriving anti patterns for data flow rules. In the next section we discuss deriving anti patterns for conditional rules.

A data flow compliance rule, cf. Formula 5.20, is violated if there is an execution state in which the respective activity A is ready to execute, **ready**(a) holds, but the data condition is not fulfilled. Formally the data flow anti pattern is

$$\mathbf{EF}(\mathbf{ready}(a) \wedge \neg dataCondition) \tag{6.9}$$

The violation occurs because of $\neg dataCondition$. That is, the data object(s) relevant to the compliance rule assume other states (values) rather than specified in the rule. To discover these *unwanted* data values, for each data object d mentioned in the rule, we issue the following temporal logic query against the investigated process

$$\mathbf{AG}(\mathbf{ready}(a) \rightarrow \mathbf{state}(d, ?_s)) \tag{6.10}$$

With Formula 6.10, we are asking about the data states (values) that d assumes at the point **ready**(a) holds. Here, the symbol d is a placeholder for the data object mentioned in the compliance rule while $?_s$ is the placeholder for its states. In general, the answer to

such a query delivers the different data object states that make the statement hold. The general form of the query result complies with Formula 5.18. We can exclude the data object values mentioned in the compliance rule from those returned as the temporal logic query answer. For each remaining value s_r, we seek to build a BPMN-Q anti pattern query that highlights the execution path from the point d was assigned that value s_r until the activity A is reached.

Recall that each data object d assumes a finite set of states. Also, for each data object we assume a known $initial$ state. To help explain how anti patterns are derived, we use an example. Consider the banking business process in Figure 5.26 that handles the "Risk" data object. the set of possible states for the "Risk" object is $\{initial,\ high,\ low\}$. Now, consider the data flow compliance rule

$$\mathbf{AG}(ready(Open\ Correspondent\ Account) \rightarrow state(Risk, low))$$

which we discussed in Section 5.5. We learned that the above rule is violated by the banking process model. Issuing the temporal logic query

$$\mathbf{AG}(\mathbf{ready}(Open\ Correspondent\ Account) \rightarrow \mathbf{state}(Risk, ?_s))$$

we receive the following answer to the query

$$\mathbf{state}(Risk, initial) \vee\ \mathbf{state}(Risk, high) \vee\ \mathbf{state}(Risk, low)$$

We can discard **state**$(Risk, low)$ from the answer since it is the value required by the rule. Having this state in the query answer means that there are *some* cases where the "Open Correspondent Account" activity is ready to execute while the "Risk" object takes the *low* value.

The other two data states, namely *initial* and *high*, are the causes of the violations. The value *initial* in the query answer tells us that there are execution path(s) where the "Risk" data object was not updated at all. That is, it keeps its initial state until the "Open Correspondent Account" activity is reached. To capture this situation on the process structure, we need to formulate an anti pattern query where there is an execution path from the start of the process to the "Open Correspondent Account" activity that excludes all activities updating the "Risk" object. It is easy to investigate the process model for those activities that update the "Risk" object. We call this set of activities $update_{Risk}$. With this information in hand, we can formulate an anti pattern BPMN-Q query as shown in Figure 6.9 where d is replaced with "Risk" and A is replaced with "Open Correspondent Account".

The other cases of violations occur due to setting the data object to some *unwanted* state that is kept until the activity in the compliance rule is reached. For the case in hand, The "Risk" object could be set to *high* and it keeps that value until "Open Correspondent Account" is reached. Thus, on the process structure, we need to find an execution path from some activity that sets the "Risk" to *high* to the open account activity. However, to highlight correct parts of the process, we have to be sure to exclude any activity on the

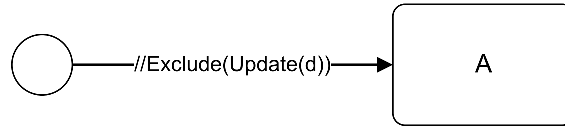


Figure 6.9: Anti pattern for data flow violation, initial value

path that set the "Risk" to low. We refer to the set of activities updating the "Risk" to state low as $update_{(Risk,low)}$. The anti pattern for such case is shown in Figure 6.10, we will return to this example with more details in Section 6.7.

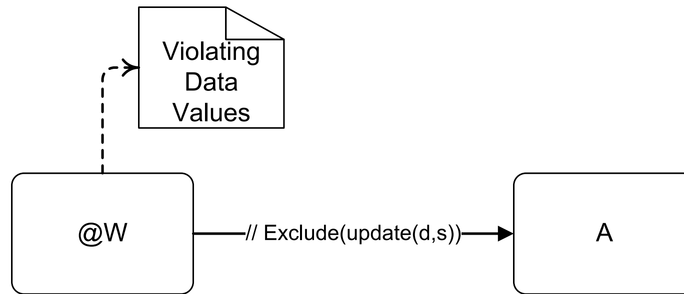


Figure 6.10: Anti pattern for data flow violation, unwanted value

6.4 Conditional Control Flow Rules Violations

Depending on the conditional rule, the generation of the anti pattern might investigate process-specific violation conditions. This is true for conditional rules that have the data condition as part of its consequent.

6.4.1 Effect Rules Violation

A violation for effect presence pattern, see Section 5.2.5.1, occurs whenever there is no activity that executes after the condition of the rule whose execution produces the required effect. Thus, the violation detection is similar to global-scope and after-scope presence violations. Formally, the violation is described as follows

$$\mathbf{EF}((\mathbf{start}/\mathbf{executed}(c)) \wedge \mathbf{EG}(\neg effect)) \tag{6.11}$$

The $effect$ propositional formula refers to Formula 5.18. However, to build the BPMN-Q anti pattern query, we have to investigate the process model for activities that are responsible for producing that effect. For each data object d and its set of states S_d mentioned in $effect$, we can lookup activities that produce these effects $\bigcup_{s \in S_d} update_{(d,s)}$. At this point, we can use anti patterns in Figures 6.1, 6.7. But, we set the *exclude* property to the set of activities $\bigcup_{s \in S_d} update_{(d,s)}$. We repeat the generation of these anti pattern queries for each data object mentioned in $effect$.

On the other hand, the effect absence violation occurs when the to-be-absent effect has a chance to occur. This situation is formalized as follows

$$\mathbf{EF}((\mathbf{start}/\mathbf{executed}(a)) \wedge (\mathbf{E}[\neg\mathbf{end}\ \mathbf{U}\ (effect \wedge \neg\mathbf{end})] \vee \mathbf{EG}(\neg\mathbf{end}))) \quad (6.12)$$

Therefore, the anti pattern is similar to those of global-scope absence, after-scope absence respectively. The anti pattern queries will be similar to the ones shown in Figures 6.2, 6.4, 6.6 respectively. However, the to-be-absent activity is replaced with the to-be-absent effect. To represent the effect in the anti pattern query, for each data state of a data object, we introduce an anonymous activity that is attached to that data value.

6.4.2 Conditional Presence Violation

The violation of conditional (after-scope) presence is similar to the violation of global-scope (effect) presence and after-scope (effect) presence, the violation occurs whenever there is a chance to skip the execution of the response activity or the activity(ies) responsible for the response effect. Formally the anti pattern can be expressed as

$$\mathbf{EF}((\mathbf{executed}(a)/true) \wedge stableDataCondition_{CTL} \wedge \mathbf{EG}(\neg(\mathbf{executed}(b)/effect))) \quad (6.13)$$

We can notice that the anti pattern formula above detects a similar violation like Formula 6.1. Thus, the anti pattern query looks for a path from the point where the condition occurred, i.e., $\mathbf{executed}(a) \wedge stableDataCondition_{CTL}$ with no chance to execute B, $effect$, afterwards to the end of the process, as shown in Figure 6.11. Following the same discussion for effect presence violation, we can deduce from the process models the set of activities that are responsible for producing the $effect$.

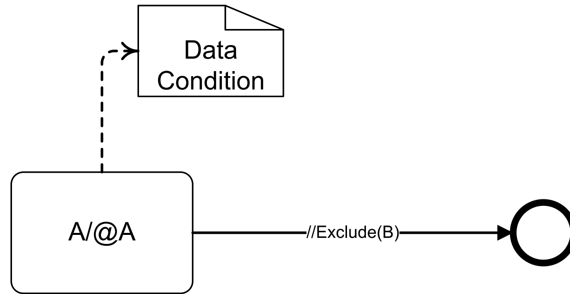


Figure 6.11: Conditional presence anti pattern

6.4.3 Conditional Precedence Violation

As discussed in Section 5.2.5.3, there are two cases for conditional precedence. The first case is for precedence to conditional activity execution, cf. Formula 5.31. The violation to this case is similar to the violation to the before-scope presence discussed above. That

is, the violation occurs whenever there is a chance to skip the execution of the precedent activity before reaching the conditionally executed activity. Formally, the anti pattern for precedence to conditional activity execution is defined as

$$\mathbf{E}[\neg\mathbf{executed}(a)\ \mathbf{U}\ \mathbf{ready}(b) \wedge dataCondition] \tag{6.14}$$

The $dataCondition$ above refers to Formula 5.18.

The anti pattern query to detect this violation is shown in Figure 6.12. The path edge, excluding A, between the start event and activity B highlights the part of the process causing the violation. The other path connecting the anonymous activity with the data condition to activity B highlights the part of the process where the conditional activity execution occurs. In case the data condition refers to more than one data object, an anonymous activity is added to the anti pattern query for each data object referred to in the data condition. Each of these anonymous activities has an outgoing data flow edge to the respective data object.

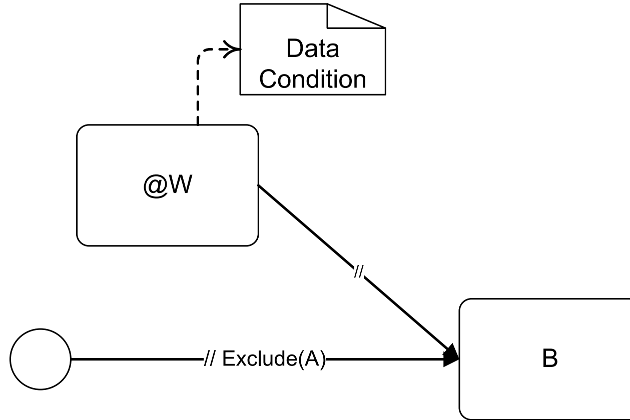


Figure 6.12: Conditional activity Execution anti pattern

Unlike precedence to conditional activity execution, detection of conditional precedence (cf. Formula 5.33) violation requires querying the behavioral model of the investigated process model. Formally, the violation occurs as follows

$$\mathbf{E}[\neg((\mathbf{executed}(a)/true) \wedge stableDataCondition_{CTL})\ \mathbf{U}\ \mathbf{ready}(b)] \tag{6.15}$$

According to Formula 6.15, the violation occurs because $(\mathbf{executed}(a)/true) \wedge stableDataCondition_{CTL}$ did not occur before activity B is reached. Recalling the definition of $stableDataCondition_{CTL}$ in Formula 5.28, the violation might be traced back to any of the following reasons:

1. Either activity A was not executed at all, or
2. The data condition $\bigvee_{(d,s)\in \mathcal{D}ataState_Q} \mathbf{state}(d,s)$ was not fulfilled for any of the data objects mentioned in the compliance rule, or

3. $\mathbf{AG}(\bigwedge_{s' \in CON_s} \neg \mathbf{state}(d, s'))$ was not fulfilled. That is, the state of the data object had been altered to a contradicting value.

Note that in case an anonymous activity was used in the rule, we can drop the case of $\mathbf{executed}(a)$ from violation explanation.

In order to identify the exact reason for the violation, we have to issue a sequence of TL queries. Depending on their results we can derive the BPMN-Q query that shows how the violation occurred.

We start by the case of an anonymous activity is mentioned in the rule, focus is on data conditions. For each data object $d \in \mathcal{D}_Q$, we investigate the process model for those activities that update d to any of the required data states, mentioned in the compliance rule. This set is obtained from $\bigcup_{(d,s) \in \mathcal{D}ataState_Q} update_{(d,s)}$. Afterwards, we check whether any of these activities always executes before B. That is, we check the less strict form of before-scope presence, see Section 5.2.3.3. Formula 6.16 describes the TL query at this stage.

$$\neg \mathbf{E}[\neg(\bigwedge_{a \in \bigcup_{(d,s) \in \mathcal{D}ataState_Q} update_{(d,s)}} \mathbf{executed}(a)) \ \mathbf{U} \ \mathbf{ready}(b)] \tag{6.16}$$

If Formula 6.16 is satisfied, we know that data conditions are the causes of the violation. On the other hand, if Formula 6.16 is not satisfied, we can use the anti pattern for before-scope presence of Figure 6.3 to highlight the violation on the process structure.

Second, we investigate for data conditions violations, cases 2, 3 listed above. Using the set of updating activities $\bigcup_{(d,s) \in \mathcal{D}ataState_Q} update_{(d,s)}$ described above, we can issue the following TL query for each data object d mentioned in the data condition.

$$\mathbf{EF}(\mathbf{executed}(a) \wedge \mathbf{state}(d, ?_s) \wedge \mathbf{EF}(\mathbf{ready}(b))) \tag{6.17}$$

where $a \in \bigcup_{(d,s) \in \mathcal{D}ataState_Q} update_{(d,s)}$.

The temporal logic query in Formula 6.17 simply asks for whatever data values the data object d can take at the moment an updating activity completes execution and afterwards activity B is reached. The answer for this query would be on the form shown in Formula 5.17. However, it is possible for some updating activities that the answer is empty. This means, for that specific updating activity, it is not the case that it executes before B.

For each reported data state that is not included in the compliance rule, we formulate an anti pattern query as shown in Figure 6.13.

Finally, the violation could be due to the fact that the data object changes its value to any of the contradicting values. To discover those contradicting values specific to the process model we issue a TL query in Formula 6.18.

$$\mathbf{EF}(\mathbf{executed}(a) \wedge \bigvee_{(d,s) \in \mathcal{D}ataState_Q} (\mathbf{state}(d, s) \wedge \mathbf{EF}(\mathbf{state}(d, ?_{s'}))) \wedge \mathbf{EFready}(b)) \tag{6.18}$$

Where we look for specific values for s' that would make the whole formula true when model checked against the process model. For each reported value for s', we can issue the anti pattern query shown in Figure 6.14.

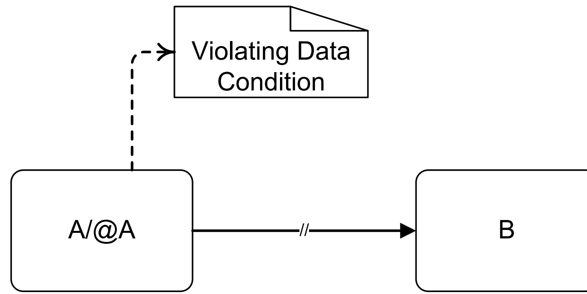


Figure 6.13: Conditional precedence anti pattern

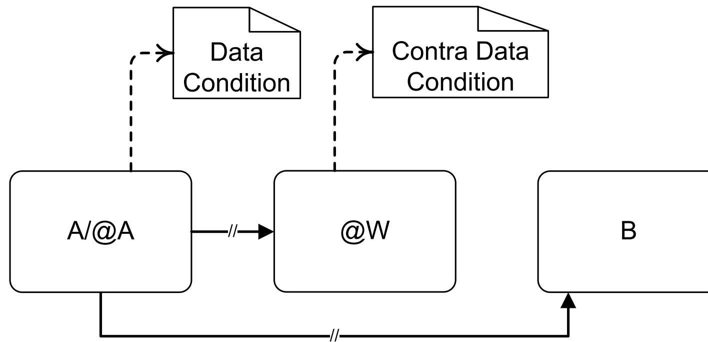


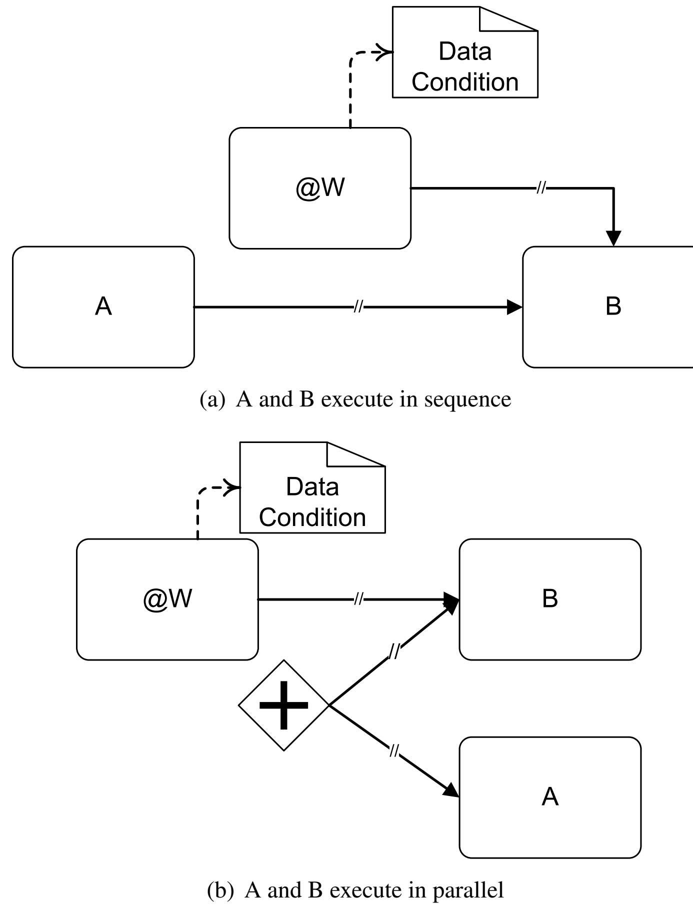
Figure 6.14: Conditional precedence anti pattern

6.4.4 Conditional Before-scope Violation

The case of conditional before-scope presence violation was already covered in Section 6.4.3. Here, we discuss the violation of conditional before-scope absence. This pattern requires that whenever an activity B is ready to execute under certain condition, another activity A must have never been executed before. Thus, the violation is similar to the general before-scope absence control flow pattern, there is a chance to execute A before B executes. Formally, the anti pattern CTL formula is

$$\mathbf{EF}(\mathbf{start} \wedge \mathbf{EF}((\mathbf{executed}(a)/effect) \wedge \mathbf{EF}(\mathbf{ready}(b) \wedge dataCondition))) \quad (6.19)$$

Following the same discussion in Section 6.2.2, it is not necessary that activity A, or activities producing the $effect$, is in sequence with B. Rather, it might be in a parallel thread to B. Thus, we can reuse the anti patterns for before-scope absence. However, we need to update these anti patterns to highlight the parts of the process producing the $dataCondition$ up to the point activity B is reached. These updated anti pattern queries are shown in Figure 6.15. The anonymous activity $@W$, attached to the data condition, has a path edge to activity B to highlight the execution path in the process through which B is ready to execute under the data condition. For each $\mathbf{state}(d, s)$ appearing in the data condition, an anonymous activity with a data flow edge to that $\mathbf{state}(d, s)$ is added to the anti pattern query with a path edge connecting it to activity B.



(a) A and B execute in sequence

(b) A and B execute in parallel

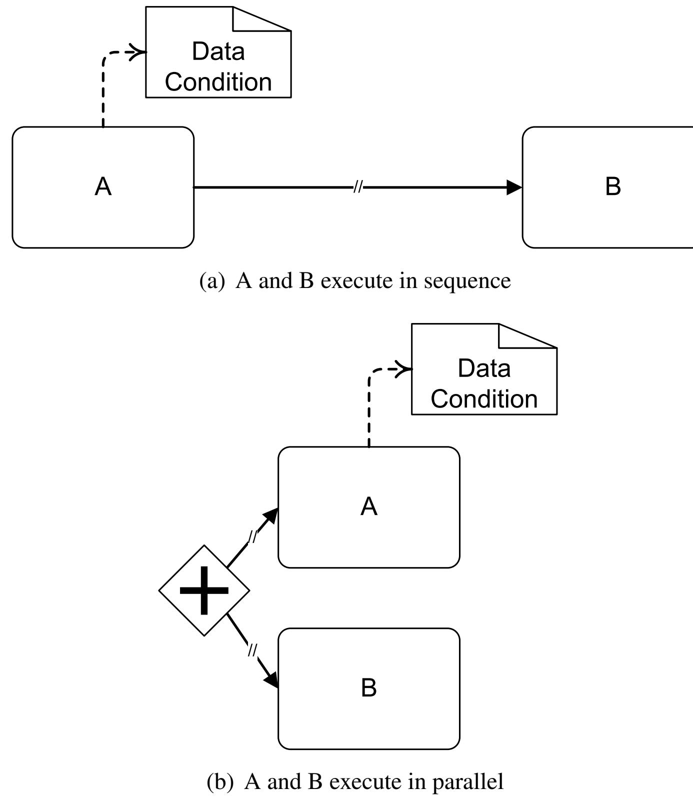
Figure 6.15: Conditional before-scope absence anti pattern

6.4.5 Conditional After-scope Violation

Violation to conditional after-scope (effect) presence was already discussed in Section 6.4.2. In this section we discuss the case of after-scope (effect) absence violation. The CTL formula for such anti pattern is

$$\begin{aligned} &\mathbf{EF}(\mathbf{executed}(a) \wedge stableDataCondition_{CTL} \wedge \\ &(\mathbf{E}[\neg \mathbf{end}\ \mathbf{U}\ ((\mathbf{executed}(b)/effect) \wedge \neg \mathbf{end})] \vee \mathbf{EG}(\neg \mathbf{end}))) \end{aligned} \tag{6.20}$$

From the formula above we notice that the violation occurs in the same case as for the control flow after-scope absence anti pattern, see Formula 6.6. The difference is that in the conditional case we require activity B, or an effect, to be absent only if activity A results in a certain data condition. Thus, we can derive the same anti pattern query to detect the violation on the structure of the process. The difference is that we need to highlight the resulting data condition of activity A. The updated anti pattern queries are shown in Figure 6.16.



(a) A and B execute in sequence

(b) A and B execute in parallel

Figure 6.16: Conditional after-scope absence anti pattern

In the above BPMN-Q query, activity B could be replaced with the effect, an anonymous activity attached to the data effect, to capture the violation to conditional effect absence pattern.

6.4.6 Conditional Between-scope Violation

There are three different patterns to express conditional between-scope absence compliance rules, cf. Section 5.2.5.6. Violation to these patterns have some commonalities to anti patterns discussed above.

The conditional between-scope absence type I, cf. Formula 5.39 requires that at the time an activity C is ready to execute and some $dataCondition$ holds; another activity A must have been executed before. Moreover, between A and C activity B must have never been executed. The CTL formula for this anti pattern is

$$\mathbf{EF}(\mathbf{executed}(a) \wedge \mathbf{EF}((\mathbf{executed}(b)/effect) \wedge \mathbf{EF}(\mathbf{ready}(c) \wedge \ dataCondition))) \vee \mathbf{E}[\neg \mathbf{executed}(a) \ \mathbf{U} \ \mathbf{ready}(c) \wedge \ dataCondition] \tag{6.21}$$

The violation for this pattern is similar to the violation of the control flow between-scope absence type II discussed in Section 6.2.4. Also, this anti pattern is similar to

the before-scope absence anti pattern. That is, absence is required only under certain data conditions. To detect the first possibility of violation in Formula 6.21, we reuse between-scope anti patterns in Figure 6.8. However, to highlight the data condition attached to activity C, we need to explicitly add that data condition to the anti pattern query as we did in Figure 6.15.

It is also possible to violate this pattern in a process model by reaching activity C and the $dataCondition$ is true but activity A was not executed before. In this case, the anti pattern in Figure 6.12 detects this case of violation.

The violation to conditional between-scope type II, cf. formula 5.41, occurs if the data condition does not hold once before reaching activity C or there is a chance to execute activity B in between. Formally, the anti pattern is captured in CTL as

$$\begin{aligned}&\mathbf{E}[\neg(\mathbf{executed}(a) \wedge stableDataCondition_{CTL}) \ \mathbf{U} \ \mathbf{ready}(c)] \quad \vee \\ &\mathbf{EF}(\mathbf{executed}(a) \wedge stableDataCondition_{CTL} \wedge \mathbf{EF}((\mathbf{executed}(b)/effect) \wedge \\ &\mathbf{EF}ready(c)))\end{aligned} \tag{6.22}$$

In part, the violation of this pattern is similar to violation to conditional precedence discussed in section 6.4.3, the first part of Formula 6.22. So, the discussion about the derivation of anti pattern queries for conditional precedence is also valid in the current case. All anti pattern queries derived for the conditional precedence can be employed to detect violation to type II between-scope absence violation.

To detect the other possibility of violation, i.e., the chance that activity B, or activities producing the $effect$, executes in between, the anti patterns for between-scope absence, in Figure 6.8, are reused. However, we need to update them by attaching the data condition to activity A, as we did in Figure 6.16.

Finally, the violation to type III conditional between-scope absence can be derived from the anti pattern CTL formula

$$\begin{aligned}&\mathbf{EF}(\mathbf{executed}(a) \wedge stableDataCondition_{CTL} \quad \wedge \\ &\mathbf{E}[\neg(\mathbf{executed}(c)/effect) \ \mathbf{U} \ (\mathbf{executed}(b)/effect2 \wedge \neg\mathbf{executed}(c)/effect)]) \vee \\ &\mathbf{EF}(\mathbf{executed}(a) \wedge stableDataCondition_{CTL} \quad \wedge \mathbf{EG}(\neg(\mathbf{executed}(c)/effect)))\end{aligned} \tag{6.23}$$

Either activity B executes, to-be-absent effect occurs, in between A and C or there is no chance to execute activity C (required effect) after the data condition holds. The anti patterns for between-scope absence, in Figure 6.8, can be reused. However, we need to update them by attaching the data condition to activity A. To detect the latter possibility of violation, the anti pattern for conditional presence can be reused, see Figure 6.11.

As discussed in Section 6.2.4, we have to model check the different causes of violations, disjuncts in Formulas 6.21,6.22 and 6.23, in order to determine which anti pattern query will be matched to the process to locate the violation.

6.5 Evaluation of Temporal Logic Queries

We have seen that for some data-dependent compliance rules; it is necessary, in order to explain violation, to investigate the process behavior for data conditions causing violation. For this, temporal logic queries are the means. In this section, we shed light on evaluation of temporal logic queries. This is necessary to make the visualization of violation for data-dependent rules possible.

Although there seems to be a generic temporal logic query solver [22], there were no publicly available prototype. However, according to [21, 20], it is possible to reduce the temporal logic query solving problem to $2^{2^{|AP|}}$ model checking problems, where AP is the set of atomic propositions used to express properties. We used that approach to implement a problem-specific temporal logic query solver for the explanation of data-dependent rules violations discussed in Sections 6.3, 6.4. We will show that we can reduce the temporal logic query solving problem into a linear number of model checking problems, using domain-specific knowledge.

From the discussion of violations for data-dependent rules, we need to issue temporal logic queries to explain violations in the following cases:

1. Data flow rules.
2. Conditional precedence rules.
3. Conditional between-scope absence type II .

The cases 2, 3 are quite similar.

6.5.1 Evaluating Data Flow Temporal Logic Queries

In Section 6.3 we showed that in order to discover violating data conditions we have to issue the temporal logic query of Formula 6.9. That is, we have to find what values (states) a data object d may assume each time some activity A is ready to execute. At a first glance, we can issue up to $2^{|state_d|}$ model checking questions representing the different disjunctions sd of data object states on the form*

$$\mathbf{AG}(\mathbf{ready}(a) \rightarrow \bigvee_{s \in sd} \mathbf{state}(d, s))$$

However, there is a chance to simplify this problem due to expressiveness of CTL as well as the nature of data states. From the discussion of Section 4.1.3.2, we know that in any execution state a data object can take exactly one value, i.e., data state. Moreover, with CTL one can issue *existential* formulas on the form, “Is there a chance to reach some state?”. Exploiting this knowledge, we can issue much simpler model checking questions to answer the temporal logic query of Formula 6.9. The simpler form is

$$\mathbf{EF}(\mathbf{ready}(a) \land \mathbf{state}(d, s)) \tag{6.24}$$

*Recall that $state_d$ is the set of all states a data object d can assume

With Formula 6.24, we check whether there is a reachable state where both propositions **ready**(a) and **state**(d, s) hold true. For each $s \in state_d$ where the corresponding formula is satisfied, **state**(d, s) is added to the disjunction representing the final answer to the temporal logic query (cf. Formula 5.18). Thus, we can answer the data flow anti pattern temporal logic query by a linear number of model checking problems rather than the doubly exponential number in the general case.

6.5.2 Evaluating Conditional Precedence Temporal Logic Queries

For the case of conditional precedence, as well as conditional between-scope absence type II, we have two temporal logic queries to answer (cf. Formulas 6.17, 6.18).

To solve the query of Formula 6.17, we follow the same argument followed in the previous subsection. Thus, we can answer the query by means of issuing a linear number of model checking problems. However, we can reduce the actual number of model checking questions exploiting the domain-specific knowledge. Recalling the notion of the domain knowledge, Section 4.4, each activity has postconditions representing its effect. Since we expect process models to be consistent with the context, any activity must produce the effect described in the domain knowledge. That is, to answer the temporal logic query of Formula 6.17, we need only to check the data states that are defined as postconditions for the activity rather than the whole set of data states a data object can assume.

To answer the temporal logic query of Formula 6.18, we already know from the domain knowledge the set of contradicting data states to a given data state **state**(d, s). Thus, we can issue a number of model checking problems equal to the number of the contradicting states.

6.6 Matching Anti Pattern Queries to Process Models With Multiple Activity Occurrences

The approach to highlight problematic parts of the process by means of BPMN-Q anti patterns queries works well in case that process models contains at most one occurrence of any activity. However, if it happens that activities have multiple occurrences, applying the approach as is might result in some false negatives. That is, it might highlight parts of the process as problematic while they are not. But, all problematic areas will be detected. We give an example to illustrate this situation. Consider the process model in Figure 6.17.

Checking that process against the compliance rule $\mathbf{AG}(\textbf{executed}(b) \rightarrow \mathbf{A}[\neg\textbf{executed}(c)\ \mathbf{U}\ \textbf{end}])$ fails. Applying the after-scope absence anti pattern in Figure 6.6 indicates the problematic part of the process as shown in Figure 6.18. The problem with the matched part is that it contains a false negative indicating a problem between activity $B2$ and $C3$. However, there is no chance to execute those two specific occurrences since they reside on two different choice branches. The reason for this false

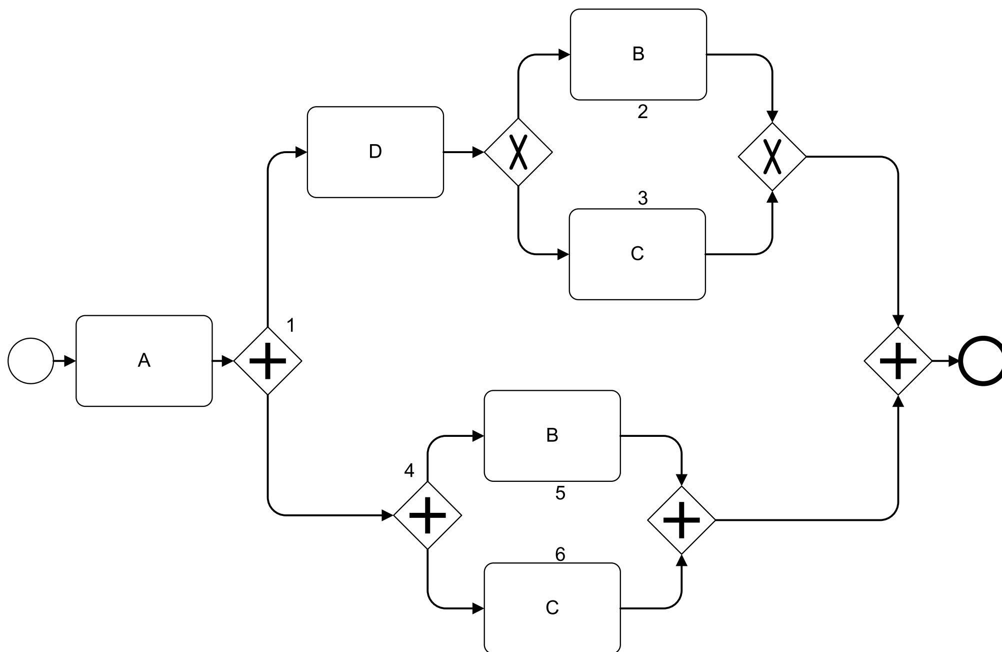


Figure 6.17: A process model with repeated activities

negative is the nature of query matching to process models. Recalling the discussion about query matching in Sections 4.2, 4.3, the AND split 1 has a path to $B2$ as well as $C3$. On the other hand, the anti pattern also highlighted the actual problematic areas. These are, parallel execution between $B2$ and $C6$. Also parallel execution of $B5$ and $C6$ was highlighted.

It is possible to avoid these false negatives. However, this comes with extra cost. To the case in hand, we need to avoid the matching between AND split 1, B2 and C3. This is possible if we start with a partially refined anti pattern query. As discussed in Section 4.2, a partially refined query is the one with some of its nodes are bound to nodes in the process model. So, having the two partially refined anti pattern queries in Figure 6.19 will not produce the false negative match. However, the identification of the specific activity occurrences that cause the violation requires investigating the process behavioral via temporal logic querying.

To enable querying the specific activity occurrences, the preparation of the state transition system, i.e., the Kripke structure needs adding more details. We need to add propositions on the form $ready_activity_ID$ and $executed_activity_ID$ for each execution state where the specific activity occurrence, with a specific ID, is ready or executed respectively. Actually, this information is already present in the mapping of process models to Petri nets, recall that each transition is assigned an ID similar to that of the respective control flow node in the process.

Assuming that the extra information is added, we can use temporal logic querying to identify the exact activity occurrences that caused the violation. For the specific example

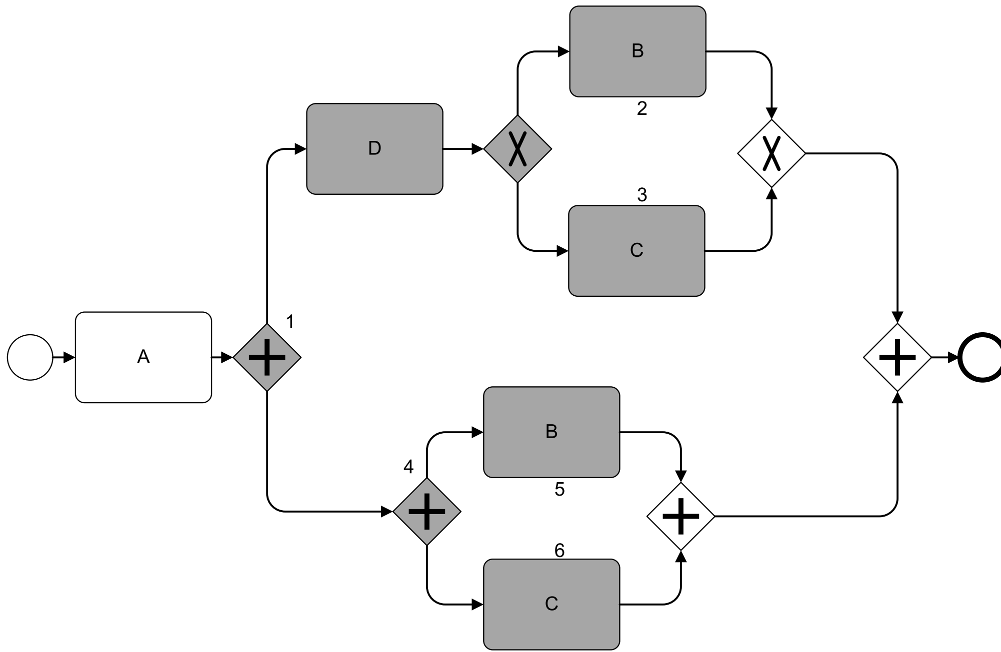
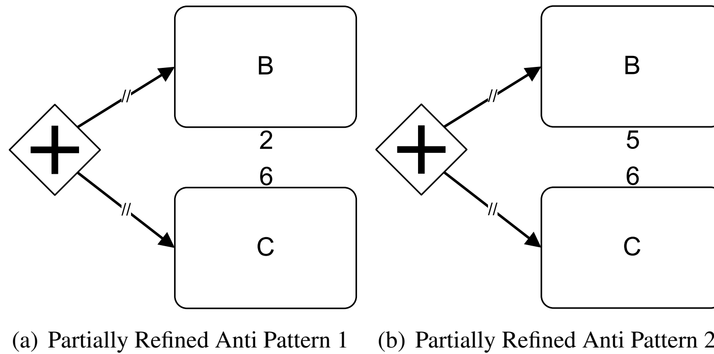


Figure 6.18: A match with false negatives



(a) Partially Refined Anti Pattern 1 (b) Partially Refined Anti Pattern 2

Figure 6.19: Partially refined anti patterns

in hand, we start from the after-scope absence anti pattern formula, cf. Formula 6.6. We repeat it here for ease of follow up.

$$\mathbf{EF}(\mathbf{executed}(b) \wedge (\mathbf{E}[\neg\mathbf{end}\ \mathbf{U}\ (\mathbf{executed}(c) \wedge \neg\mathbf{end})] \vee \mathbf{EG}(\neg\mathbf{end})))$$

The anti pattern formula states that a violation occurs whenever activity B occurs and afterwards C occurs before terminating the process. To identify the exact occurrences of B and C that caused the violation we issue the following TL query

$$\mathbf{EF}(\mathbf{executed}(?_{b_{ID}}) \wedge \mathbf{EF}(\mathbf{executed}(?_{c_{ID}})))$$

In the above query $?_{b_{ID}}$ indicates that we are interested in propositions $executed_b_ID$, i.e., all propositions that indicate execution of specific B occurrences, similarly is $?_{c_{ID}}$

for activity C. The answer for the above TL query against the process in Figure 6.17 yields the answer that makes the following CTL formulas true.

$$\mathbf{EF}(\mathbf{executed_}b_2 \wedge \mathbf{EF}(\mathbf{executed_}c_6))$$
$$\mathbf{EF}(\mathbf{executed_}b_5 \wedge \mathbf{EF}(\mathbf{executed_}c_6))$$

Based on the answer to the TL query we can formulate the partially refined queries as those in Figure 6.19.

The approach described above can be generalized for all anti patterns discussed in this chapter as follows:

1. Issue a temporal logic query based on the anti pattern formula.

2. Create partially refined anti pattern queries based on the result from the previous step.

3. Match the partially refined anti pattern queries to the process model.

It is clear that to avoid the false negative there is an extra cost on the form issuing more temporal logic queries to identify the exact cause of violations. However, there is a processing cost saving on the side of the BPMN-Q query processor because it starts from the partially refined queries rather than the totally unrefined one.

6.7 Back to Example

In Section 5.5, we showed how a set of compliance rules can be modeled as BPMN-Q behavioral queries. Some of these rules were violated by the process model of Figure 5.26. In this Section, we apply the anti patterns discussed in this chapter to explain violations on the process model structure. Recall that rules $R3$, $R5$ were the only rules violated. Thus, we add more rules that are intentionally violated by the model.

- $R3$: Opening an account must be of low risk.

- $R5$: If it is the first time to deal with the respondent bank, advanced due diligence study must be conducted.

- $R7$: Due diligence study must always be conducted.

- $R8$: Before opening an account, the respondent bank certificate must be valid

In the above list we repeated rules $R3$, $R5$ for convenience. For each violated rule, we explain the violation.

Explaining Violations to $R3$

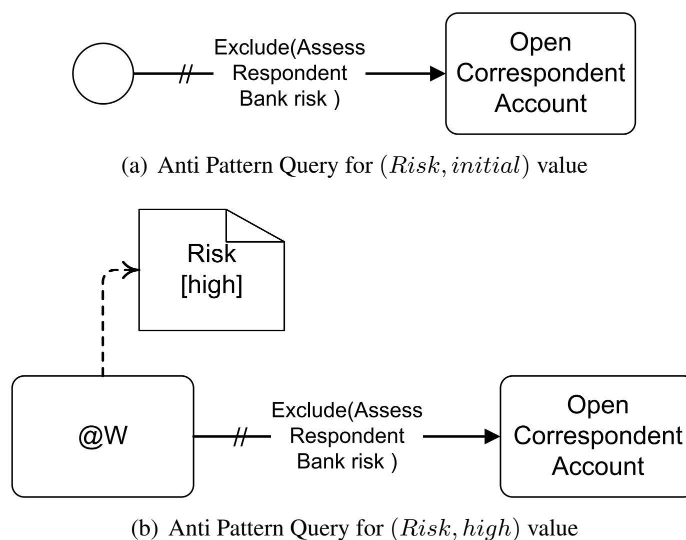
The rule $R3$ states that "Opening an account must be of low risk". The rule was modeled as the BPMN-Q query shown in Figure 5.29. This rule follows the data flow pattern. We learned also that the rule is not satisfied by the model in Figure 5.26. To explain this violation, we derive anti patterns according to the discussion in Section 6.3. That is we issue the following temporal logic query

$$\mathbf{AG}(\mathbf{ready}(Open\ Correspondent\ Account) \rightarrow \mathbf{state}(Risk, ?))$$

Evaluating the above temporal query against the open account process yields the following answer

$$\mathbf{state}(Risk, initial) \vee \mathbf{state}(Risk, high) \vee \mathbf{state}(Risk, low)$$

We can exclude the value $\mathbf{state}(Risk, low)$ from the answer above since it is the one originally included in the compliance rule. Now we have two remaining values, $\mathbf{state}(Risk, initial)$,$\mathbf{state}(Risk, high)$. Thus, we generate the two anti pattern queries of Figure 6.20.



(a) Anti Pattern Query for $(Risk, initial)$ value

(b) Anti Pattern Query for $(Risk, high)$ value

Figure 6.20: Anti patterns for $R3$

The structural matching of the query in Figure 6.20(a) to the open account process in Figure 5.26 is shown in Figure 6.21. This match shows that it is possible open a correspondent account where the "Risk" object is kept to its *initial* value. This is possible in the cases where the open account request is received from a respondent bank for which previous open account requests were handled.

On the other hand, matching the query in Figure 6.20(b) highlights the other possibility of violation to $R3$. In this case, although risk is assessed; it is still possible to open the account while "Risk" is *high*. The matching to the process is highlighted in Figure 6.22.

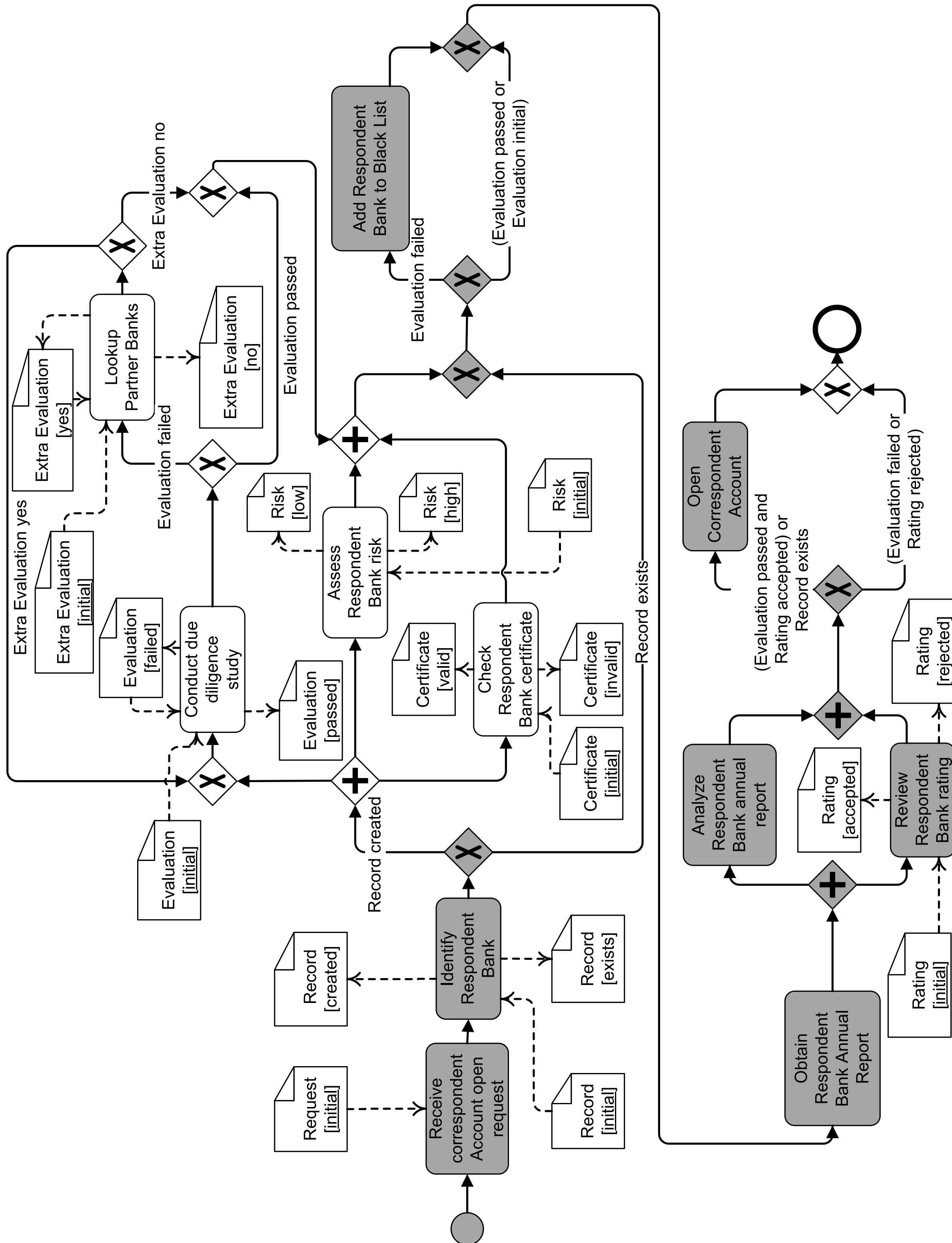


Figure 6.21: An execution scenario that violates $R3$

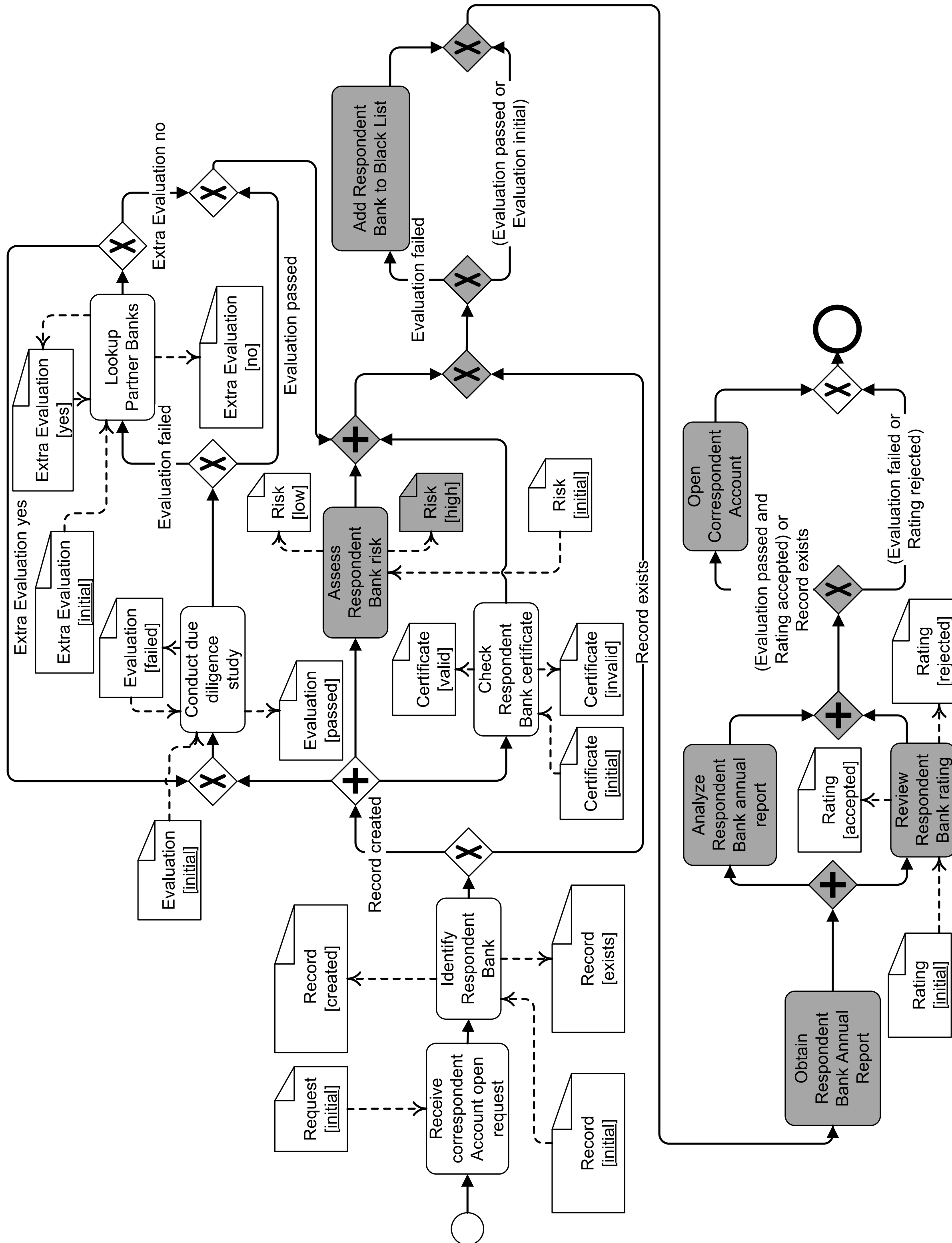


Figure 6.22: Another execution scenario that violates $R3$

Model checking $R3$ would return only one of the two violation scenarios shown above as a counterexample. In order to discover the other possibility of violation, the first one had to be fixed, model checking is run again. It is obvious that our approach provides complete feedback on all possible violations. Moreover, it requires only model checking once.

Explaining Violations to $R5$

Rule $R5$ follows the conditional presence pattern. Thus, an anti pattern could be derived by finding any execution path from the point where the condition holds to process end without visiting the activity “Conduct advanced due diligence study”. The corresponding anti pattern is shown in Figure 6.23. Moreover, in Section 5.5, we decided that rule $R5$ is structurally non-compliant (cf. Definition 5.8). Thus, the anti pattern would match the whole process after activity “Identify Respondent Bank” indicating that all execution paths from that point do not visit the “Conduct advanced due diligence study” activity. We did not show the match to save space.

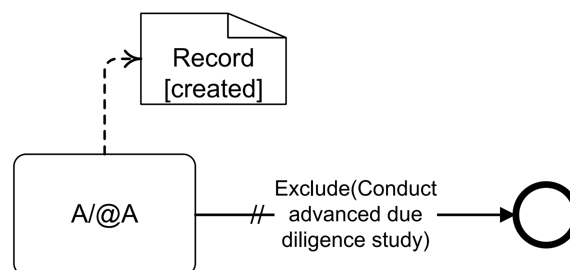


Figure 6.23: Anti pattern for $R5$

Explaining Violations to $R7$

Rule $R7$ is a global-presence rule. The anti pattern query of Figure 6.24 looks for execution paths from the start of the process to its end without visiting the activity “Conduct due diligence study” (cf. Section 6.2.1). The matching to open account process is highlighted in Figure 6.25.

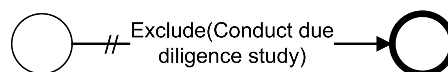


Figure 6.24: Anti pattern for $R7$

In the highlighted part, we can notice that not only “Conduct due diligence study” was excluded. Rather, all branches that are running in parallel to it. This is a feature implemented in the BPM-Q query processor. That is, whenever an activity is excluded in

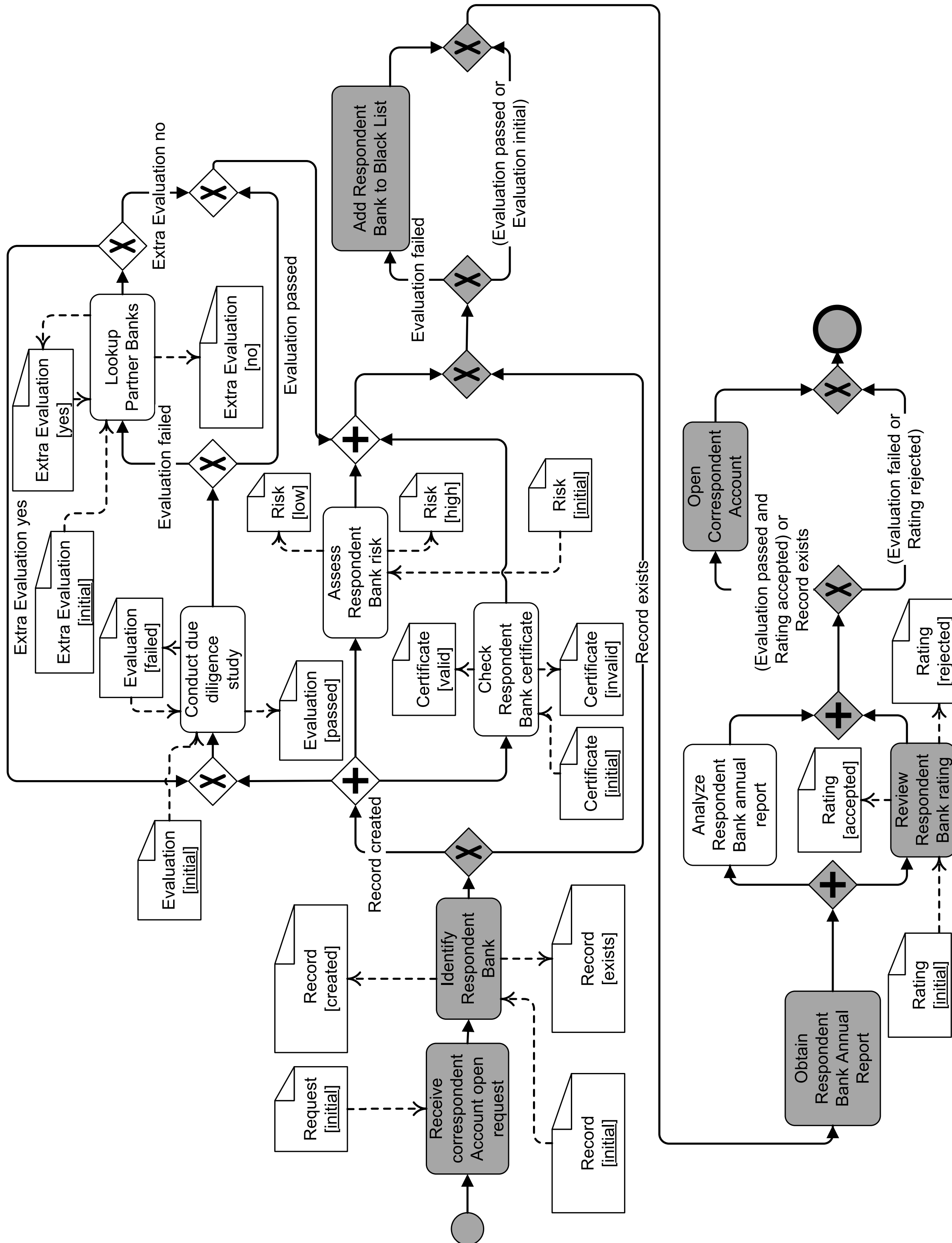


Figure 6.25: An execution scenario that violates $R7$

a path, all parallel threads have to be excluded. Although structurally this is not necessary. It is necessary to give meaningful feedback. That is, if there is an execution scenario that skips some activity, it also skips all activities that run in parallel to it.

Explaining Violations to $R8$

Rule $R8$ could be modeled as a conditional precedence using the template in Figure 5.18, replacing "A/@A" with "Check Respondent Bank certificate" and "B" with "Open Correspondent Account". The rule is violated. Thus, following the approach discussed in Section 6.4.3, we have to check several conditions in order to derive anti pattern queries.

Checking the weaker form of the rule $R8$, i.e., checking whether the "Open Correspondent Account" activity is always preceded by the "Check Respondent Bank certificate" activity, fails. Thus, we can derive the anti pattern query of Figure 6.26. That is, there is a chance to skip the execution of "Check Respondent Bank certificate" before reaching "Open correspondent Account". Matching that query to the open account process returns the same result as shown in Figure 6.25.

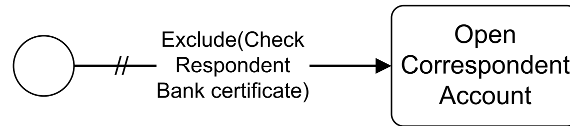


Figure 6.26: Anti pattern for $R8$

To check whether there are data-dependent violations, we issue the following temporal logic query

$$\textbf{EF}(\textbf{executed}(Check\ Respondent\ Bank\ certificate) \wedge$$
$$\textbf{state}(Certificate, ?_s) \wedge \textbf{EF}\ \textbf{ready}(Open\ Correspondent\ Account))$$

The formula above uses the template of Formula 6.17. The answer to the above formula is

$$\textbf{state}(Certificate, valid) \vee \textbf{state}(Certificate, invalid)$$

From the answer to the query we can derive the following anti pattern

The matching part of the open account process to the anti pattern in Figure 6.27 indicates that it is possible to execute "Check Respondent Bank certificate" activity with a result of an *invalid* "Certificate". Yet, the "Open Correspondent Account" activity is still reachable. We do not show a specific figure with the highlighted matching part to save space.

6.8 Summary & Outlook

In this chapter, we presented an approach to explain violations to the user by means of highlighting execution paths in the process that cause the violation. Violation scenarios

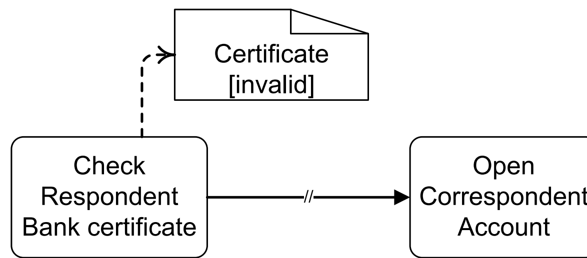


Figure 6.27: Another anti pattern for $R8$

were derived by negation of the CTL formula for the compliance pattern, see Section 6.1. Based on the negated formula, one or more violation scenarios could be derived. Each violation scenario is declaratively represented as a BPMN-Q structural query. Those queries (violation scenarios) are called anti patterns. Structurally matching the anti pattern query to the process directly highlights the part of the process where the problems are. We discussed through the chapter and showed in Section 6.7 that our approach provides exhaustive explanation to possible violations compared to interpreting the feedback of the model checker.

In all cases, we start deriving anti patterns once we learn that the compliance rule is not satisfied by the model. On one hand, control flow anti pattern, see Section 6.2, can be derived directly from the corresponding pattern (rule) query. On the other hand, for some data-dependent rules, see Section 6.3 and 6.4, the derivation of anti pattern queries require *querying* the behavior of the investigated process.

Temporal logic querying was the tool to *query* the behavior of process models. We showed in Section 6.5, how domain-specific knowledge reduced the complexity of evaluating temporal logic queries.

Finally, in Section 6.7 we built on the rules introduced in Section 5.5 and applied the approach of anti patterns to that case study.

Next, we provide a road map to (semi) automated resolution of violations.

Chapter 7

Resolution of Compliance Violation

In Chapters 5, 6, we discussed how to model, check and explain violations to compliance rules. Model checking and temporal logics were the formal background to our approach. In this chapter, we address the problem of providing suggestions to the user in order to remove violations and enforce compliance. To this end, model checking and temporal logic techniques fall short to cope with the new problem. Firstly, model checking deals with the behavioral model of the process while making changes to the process necessitates doing modifications on the structural level. Moreover, the changes to be made to the process should be kept as minimal as possible; it must keep the changed process model consistent with the domain knowledge. Finally, temporal logic is of a declarative nature that fitted well with the nature of compliance rules. When it comes to modifying the process structure, more details have to be added to keep the process operational. For instance, a compliance rule which states that "for each received insurance claim there has to be a reply sent to the client" abstracts from details like investigating and evaluating the claim and comparing it to the client policy before deciding to pay back the claim's amount. All these details must be present to make the process model operational.

While our approach in the previous two chapters was complete. That is, we were able to decide about compliance for each rule and we were able to explain violations. In this chapter, we give directions for automated resolution of violations. We investigate how far the domain knowledge can be exploited to help the user find resolutions to the violations. Even if it is not possible to automatically resolve the violation, we help the user assess the effort needed, in terms of changes to the way business is conducted and understood, to enforce compliance.

To simplify the discussion, we focus on resolving violations to after-scope presence, i.e., $\mathbf{G}(\textbf{executed}(a) \rightarrow \mathbf{F}(\textbf{executed}(b)))$. We refer to this rule simply as A *leads to* B. In this chapter, we discuss the possibilities of resolving violations to this pattern. Later on, we discuss how resolution to violations of other patterns can be approached.

To reach this objective, the rest of this chapter is organized as follows. In Section 7.1

we argue that we need a different tool set in order to address the violation resolution problem. We study the possible violations to A *leads to* B rule on the process structural level in Section 7.2. Based thereon, we suggest various resolutions and discuss their applicability in Section 7.3. Directions to resolve violations to other patterns is given in Section 7.4. Finally, Section 7.5 discusses the limitations of the approach.

7.1 Another Tool Set

We have two problems that temporal logics and model checking cannot address:

- Analyzing the violation on the structure of the process model
- Filling the gap between the abstract compliance rules and the detailed operational process models.

In order to resolve a violation, we have to make modifications to the structure of the process model. Modifications can be in the form of adding, removing, or moving activities to restore compliance. This necessitates the study of the execution relation between activities A and B, mentioned in the compliance rule. Process decomposition techniques provide a good starting point to analyze compliance violation, and thus resolution, on the structural level. We rely on the concept of process structure tree (PST) [144, 145], which is the process analogue of abstract syntax trees for programs, we give more details in Section 7.1.1.

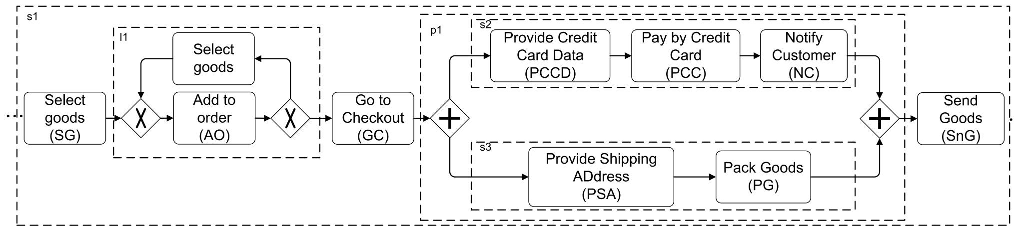
The second problem that appears when we need to provide automated resolution is the abstraction-level gap between compliance rules and operational process models. As stated earlier, process models must have a sufficient amount of details to make it operational and to meet its business objective. On the other hand, compliance requirements focus on a specific situation and leaves all other details out. Thus, we need a technique that can derive a detailed process that is yet compliant.

To fill the abstraction-level gap between compliance rules and detailed processes, we depend on automated planning techniques [97]. In automated planning, the planner is given three inputs, the initial state, the target state and the action space. It is the planner's task to find a sequence of actions that can make the system evolve from the initial state to the target state, see Section 7.1.2 for more details.

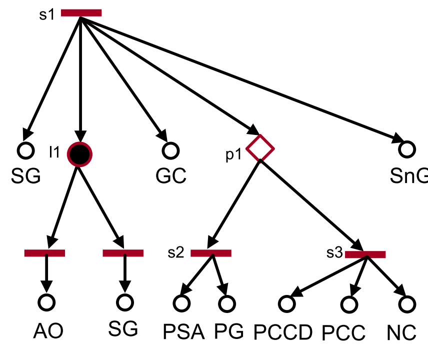
7.1.1 Process Structure Tree

The concept of a process structure tree is based on the unique decomposition of a process model into fragments. One approach is the decomposition of a model into *canonical single entry single exit (SESE) fragments*, formally described in [144, 61]. Informally, a SESE fragment is a fragment with exactly one incoming and exactly one outgoing edge. The node sets of two canonical SESE fragments are either disjoint or one contains the other. Following [144], we consider the maximal sequence of nodes to be a canonical

SESE fragment. If the node set of SESE fragment f_1 is the subset of the node set of SESE fragment f_2, then f_1 is the *child* of f_2 and f_2 is the *parent* of f_1. If f_1 is the child of f_2 and there is no f_3, such that f_3 is the child of f_2 and f_3 is the parent of f_1, f_1 is the *direct child* of f_2. Canonical SESE fragments can be organized into a hierarchy according to the parent-child relation. The hierarchy is represented with a directed tree called *process structure tree*. The tree nodes represent canonical SESE fragments. Figure 7.1 shows an example process model along with its decomposition into SESE fragments and their nesting.



(a) An example process model



(b) SESE fragments hierarchy (PST)

Figure 7.1: Decomposition of a process in SESE fragments and its process structure tree

Definition 7.1. *[Process structure tree]*

A process structure tree $PST = (N, E, r, t, cond)$ *is a tree, where:*

- N *is a finite set of* nodes, *where nodes correspond to canonical SESE fragments,*
- $E \subseteq (N \times (N \setminus \{root\}))$ *is the set of* edges. *Let tree nodes* $n_1, n_2 \in N$ *correspond to SESE fragments* s_1 *and* s_2 *respectively. An edge leads from* n_1 *to* n_2 *if SESE fragment* s_1 *is the direct parent of* s_2*,*
- $r \in N$ *is the* root *of the tree.* r *represents the main sequence of the process model,*

- $t : N \to \{\mathtt{act}, \mathtt{seq}, \mathtt{and}, \mathtt{xor}, \mathtt{loop}\}$ is a function assigning a type to each node in N: `act` corresponds to activities, `seq`—sequences, `and`, `xor`—blocks of corresponding type, `loop`,
- $cond : \{n : n \in N \wedge t(n) \in \{\mathtt{seq}, \mathtt{loop}\}\} \to 2^{2^{Pr}}$ is a function that assigns a condition for sequence and loop nodes, *where Pr is a set of atomic proposition.*

The definition distinguishes five node types: activities, sequences of activities, parallel blocks, exclusive choice blocks, and loops. While activities, sequences, and blocks are natural to understand, we assume loops to be a SESE fragment containing at least two gateways: a join and a split. The join is incident to the entry edge of the fragment, the split—to the exit edge of the fragment. The loop has two branches: one leading from the join to the split is always executed (we call it *mandatory*) and the other leading from the split to the join may be skipped (*optional*). It is allowed that one branch does not contain nodes. In the illustrations depicting PSTs, sequences are visualized with horizontal line, blocks—with diamond-shaped figure. If the node type is unimportant it is captured as a filled circle. Activities are leaf nodes in the tree and are represented with unfilled circles.

A process model may contain several occurrences of one activity (e.g. activity A). Then, the model's PST has the set of nodes which correspond to occurrences of A. To address such a set of nodes we denote it with $N_a \subset N$.

7.1.2 Automated Planning

A violation resolution often implies that a business process logic is changed. The severity of changes may vary. The task is always to come up with a compliant model, fulfilling the business goal. This implies that a process should be reorganized to assure that the requirements are satisfied. Given that the set of activities required to construct a compliant process is available, this task can be approached with techniques of automated planning [97]. Automated planning techniques are often employed by service-oriented architecture community for a service composition problem, e.g., in [86]. We demonstrate how a resolution of compliance violations can be expressed in terms of automated planning.

The problem of automated planning can be described as follows. Given a system in an initial state it is required to come up with a sequence of actions that brings the system to the goal state. The sought sequence of actions is called a plan. A system can be represented as a state-transition system which is a 3-tuple $\Sigma = (S, A, \gamma)$, where S is a finite set of states, A is a finite state of actions, and $\gamma : S \times A \to 2^S$ - a state transition function. A planning task for system $\Sigma = (S, A, \gamma)$, an initial state s_0, and a subset of goal states S_g is to find a sequence of actions $\langle a_1, a_2, \ldots, a_k \rangle$ corresponding to a finite sequence of transitions $(s_0, s_1, \ldots, s_k)$ such that $s_1 \in \gamma(s_0, a_1), s_2 \in \gamma(s_1, a_2), \ldots, s_k \in \gamma(s_{k-1}, a_k)$ and $s_k \in S_g$.

To formalize the resolution problem in terms of automated planning we need to explain what are $\Sigma = (S, A, \gamma)$, s_0, and S_g. The system Σ is a business environment,

where a business process is executed and which evolves as the next activity completes. Hence, actions in the planning task are associated with instances of activities described by the domain knowledge, while system states—with the states of the environment where a process executes. Function γ defines transition rules in the planning domain. In the process domain the context defines the preconditions and effects for every activity, Definition 4.10, which aligns with the transition function. A transition from the current state to the next state via application of an activity results in removing all the effects defined by the $post^{-}$ relation and adding the effects defined by the $post^{+}$ relation. The current state reflects the effects of all the activities which have taken place before. Initial state s_0 corresponds to the state of the environment before the first activity of the process took place. Set S_g consists of the states in which the business goal of the process is fulfilled and a compliance rule is not violated. The states can be described in terms of first order logic.

A limitation in planning is that generated plans have no possibility to represent choices. Although this is logical in plans used to move robots, it is considered a limitation when adapting the approach to the business process field. Another limitation is that planning algorithms always start from the initial state trying to reach the goal state. It is not possible, for instance, to start with a partial plan which the planner tries to make it a full plan by adding, removing, and/or reordering actions. This is only possible in case-based planning [26] where the planner should have a large enough repository of similar problems that are associated with their solutions. Unfortunately, this is not our case.

7.2 Catalog of Violations

As was stated earlier, we focus on after-scope presence rules, A *leads to* B. In Section 6.2, we discussed how to explain violations to these rules by anti patterns. In this section, we analyze the violations using PSTs and present the catalog of compliance violations [13]. For each violation, we give the name and briefly discuss the problem it addresses. The effort to resolve the violation depends on the violation scenario in the business process. That is, if A *leads to* B is violated by two process models $p1$ and $p2$ where in $p1$ B execution is skipped while in $p2$ there is no occurrence of B at all, then it is probably less effort to resolve the violation in $p1$ than in $p2$.

Since a process model might contain more than one occurrence of the activities A and B under investigation, we assume a priori knowledge about the pairing of such occurrences.

The A *leads to* B rule is violated in two cases: either there is no path leading from A to B, or there is a path leading from A to the process end, but not containing B. These two cases are captured declaratively by the anti pattern of Figure 6.7. However, analyzing the structure of a business process, using PSTs, we can derive more detailed violations.

The catalog of violation can be considered as an alternative approach to locate

violations on the process structure level, in contrast to anti patterns discussed in Chapter 6. However, we provide this catalog as the basis for the resolution algorithms discussed in the next section.

In order to formally describe the violations, we need to define some terms. These definitions depend on the process structure tree given in Definition 7.1.

Definition 7.2. *[Path]*
A path between two nodes $n_0, n_k \in N$, is a sequence of nodes $path(n_0, n_k) = (n_0, n_1, \ldots, n_k)$ where $(n_i, n_{i+1}) \in E, 0 \leq i < k$. The length of the path is the number of nodes in this path and we denote it with $|path(n_0, n_k)|$.

Since the compliance rule we address describes the execution ordering between two activities, determination of the block type in which occurrences of A and B execute helps determine the violations. The least common ancestor of two nodes represents the smallest scope (block) in which the nodes occur.

Definition 7.3. *[Least common ancestor]*
The least common ancestor of two nodes $n, m \in N$ in the $PST = (N, E, r, t)$ is a node $lca(n, m) = p$ where $p \in N \land p \in path(r, n) \land p \in path(r, m) \land \nexists p' : p' \in path(r, n) \land p' \in path(r, m) \land p \in path(r, p')$.

Also, given two occurrences of A and B respectively, the order in which they appear within a sequence block determines the possibility of violation.

Definition 7.4. *[Order]* The order of execution of a node $n \in N$ with respect to node $p \in N$, where $(p, n) \in E \land t(p) = \texttt{seq}$ is a function:
$order : N \times N \rightarrow \mathbb{N}$,
where the first argument is the parent node and the second—its child.

The notion of order can be extended to all the children of a node with type seq.

Definition 7.5. *[Order*]* The order of execution of a node $n \in N$ with respect to node $p \in N$ where $|path(p, n)| \neq 0 \land t(p) = \texttt{seq}$ is a function
$order^* : N \times N \rightarrow \mathbb{N}$,
defined as:

$$order^*(p, n) = \begin{cases} order(p, n) & \text{, if } (p, n) \in E, \\ order(p, k) & \text{, where } (p, k) \in E \land k \in path(p, n) \text{, if } \nexists (p, n) \in E. \end{cases}$$

Finally, we introduce a family of auxiliary functions $exec_{<type>}$. Every function checks if a given activity is *always* executed within the given fragment. To answer this question a recursive analysis of all fragment's children is done. The following definition formalizes this function family in terms of PST.

Definition 7.6. The family of functions $exec_{<type>} : N_{<type>} \times N_{act} \rightarrow \{true, false\}$ is defined as:

$$exec_{xor}(p,a) = \begin{cases} true, & \text{if } \bigwedge_{\forall x:(p,x)\in E} exec_{t(x)}(x,a) = true, \\ false, & \text{otherwise.} \end{cases}$$

where $p \in N_{xor}$ and $a \in N_{act}$.

$$exec_{and}(p,a) = \begin{cases} true, & \text{if } \bigvee_{\forall x:(p,x)\in E} exec_{t(x)}(x,a) = true, \\ false, & \text{otherwise.} \end{cases}$$

where $p \in N_{and}$ and $a \in N_{act}$.

$$exec_{seq}(p,a) = \begin{cases} true, & \text{if } \bigvee_{\forall x:(p,x)\in E} exec_{t(x)}(x,a) = true, \\ false, & \text{otherwise.} \end{cases}$$

where $p \in N_{seq}$ and $a \in N_{act}$.

$$exec_{loop}(p,a) = \begin{cases} true, & \text{if for the direct child node } x \text{ of } p \text{ laying on} \\ & \text{its mandatory branch holds } exec_{t(x)}(x,a) = true, \\ false, & \text{otherwise.} \end{cases}$$

where $p \in N_{loop}$ and $a \in N_{act}$.

$$exec_{act}(p,a) = \begin{cases} true, & \text{if } p \in N_a, \\ false, & \text{otherwise.} \end{cases}$$

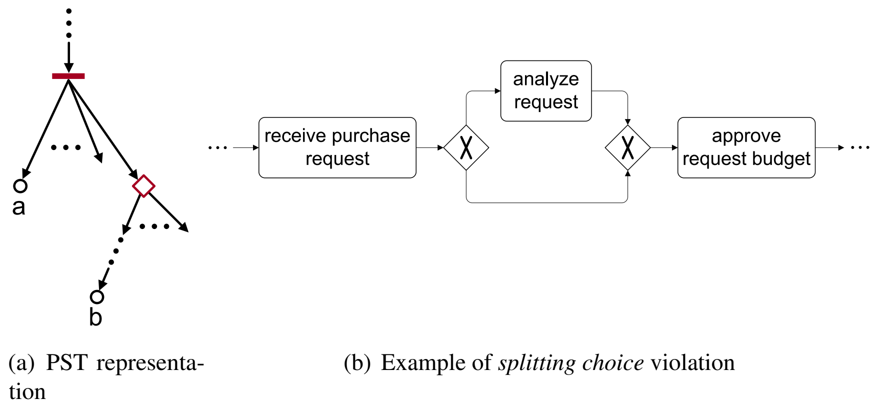
where $a, p \in N_{act}$.

With the help of the above defined terms in addition to the knowledge about A and B occurrence pairs, we can identify four ways of violating a A *leads to* B rule, splitting choice, different branches, inverse order or lack of activity. In the remaining of this section we describe each violation possibility in details.

7.2.1 Splitting Choice

This type of violation can be motivated by the following example. Assume we have a business process, containing the fragment presented in Figure 7.2(b). The fragment shows that once a purchase request is received, its budget should be approved; before the approval the request can be optionally analyzed. However, one can require that every received purchase request must be analyzed. This requirement can be formalized in the form of a compliance rule *Receive purchase request* `leads to` *Analyze request*. In the presented fragment this rule is violated, since after purchase request is received, the analysis can be skipped.

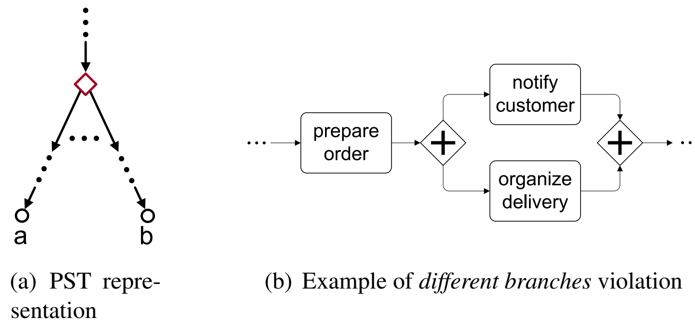
We call this type of violation *splitting choice*. `Leads to` compliance rule has a violation of type *splitting choice* if a model contains occurrences of both activities A and



(a) PST representation

(b) Example of *splitting choice* violation

Figure 7.2: Splitting choice violation



(a) PST representation

(b) Example of *different branches* violation

Figure 7.3: Different branches violation

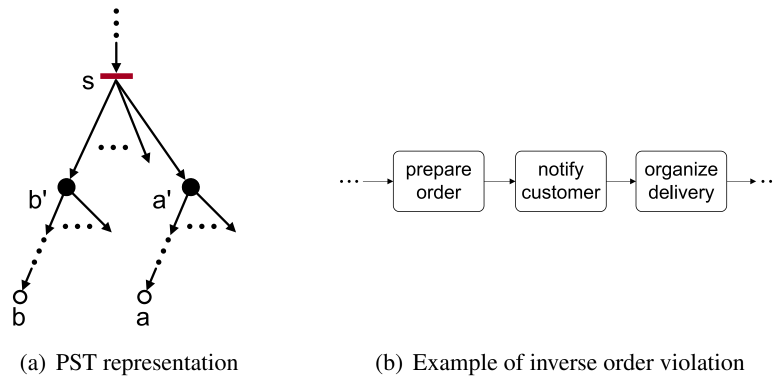
B connected with a path, but this path contains an XOR split. Thus, the process model provides an option not to execute B, once A is executed. Obviously, this contradicts to the semantics of the `leads to` rule. Another possibility for splitting choice violation occurs when activity B is on the optional branch of a loop block.

Definition 7.7. *[Splitting choice violation]*

A process model has a violation of compliance rule A `leads to` B *and this violation is of type* splitting choice *if the PST for this model contains nodes a and b corresponding to activities A and B, such that $s = lca(a, b) \wedge ((t(s) = seq \wedge \exists x \in path(lca(a, b), b) : t(x) = xor \wedge exec(x, b) = false) \vee (s \in N_{loop} \wedge exec(s, b) = false))$.*

7.2.2 Different Branches

Activities A and B can be located on different branches of a block. Independently of the block type, compliance rule A *leads to* B is violated. Indeed, semantics of an `and` block does not allow predicting the execution order of A and B. A `xor` block allows execution



(a) PST representation (b) Example of inverse order violation

Figure 7.4: Inverse order violation

of either only A, or only B. From the structural perspective violation of a rule follows from the fact that there is no execution path neither leading from A to B, nor from B to A (cf. Fig. 7.3(a)).

The example of *different branches* violation is shown in Figure 7.3(b). *notify customer* and *arrange delivery* activities are executed in parallel. Concurrent activities allow the company to shorten the execution time. However, once the company policy requires to notify a client about delivery details (i.e., *Arrange delivery* `leads to` *Notify customer* compliance rule is imposed), the business process becomes non-compliant.

Definition 7.8. *[Different branches violation]*

A process model has a violation of compliance rule A `leads to` B *and this violation is of type* different branches *if:*

- *the PST for this model contains nodes a and b corresponding to activities A and B, respectively,*
- $t(lca(a, b)) \in \{\mathtt{and}, \mathtt{xor}\}$.

7.2.3 Inverse Order

Probably the most challenging violation of rule*A leads to B* is the case when activities A and B appear in the inverse order. This means that a process model contains both activities A and B, connected with a path, but this path leads from B to A. Obviously, a compliance rule can not hold.

The inverse order violation can be illustrated by the following example. Assume that a company sends a notification with an order summary to the customer once the order is prepared. Afterwards, the company contacts its logistics partner to arrange the delivery. Fig. 7.4(b) captures a fragment of the model corresponding to this process. New business conditions require the company to include the delivery information in the notification, i.e. first the delivery should be organized. This requirement is captured in the

rule *organize delivery* `leads to` *notify customer*. In this case inverse order violation takes place. Fig. 7.4(a) captures this violation type in terms of PST and the following definition formalizes it.

Definition 7.9. *[Inverse order violation]*

A process model has a violation of compliance rule A `leads to` B *and this violation is of type* inverse order *if:*

- *the PST for this model contains nodes a and b corresponding to activities A and B, respectively,*
- $t(lca(a, b)) = seq$,
- $order^*(lca(a, b), b) < order^*(lca(a, b), a)$.

7.2.4 Lack of Activity

A process model is non-compliant to the A *leads to* B rule, if it has at least one occurrence of A and no occurrence of B. Consider a compliance rule *Receive purchase request* `leads to` *Close purchase request*. Checking the process model fragment in Figure 7.2(b) against this rule, we see that the rule is violated because *Receive purchase order* is executed, but *Close purchase order*—never, since it is missing in the process model.

Definition 7.10. *[Lack of activity violation]*

A process model has a violation of compliance rule A `leads to` B *and this violation is of type* lack of activity *if the PST for this model contains node a corresponding to activity A and for this node a $N_b = \emptyset$.*

7.3 Resolving Violations

In this section we explain how compliance rule violations can be resolved [12]. We base the resolution strategy on the violations catalog discussed in Section 7.2. For each violation case, we discuss a set of resolution alternatives. These alternatives are about adding, removing, or moving occurrences of activities A, B, and/or other dependent activities in order to gain compliance. The alternatives are evaluated based on the compliance rule, the process model, and the domain knowledge, cf. Section 4.4. For each violation case, we devote an algorithm that selects an alternative to resolve the violation, if there is a resolution. These algorithms to a large extent exploit automated planning.

According to each violation case, automated planning will be applied to 1) decide whether the required rule is possible to satisfy, using the available domain knowledge, 2) find required in-between activities to make the changes to the original process consistent. In some cases, the found resolution is valid only under restrictions to the behavior of activities.

We use the function $findPlan(init, goal, condition, context)$ in our algorithms to encapsulate the call for an AI planner. The plan is a partial ordering of activities [104]. A partial order plan is similar to a process fragment where activities are either in sequence or parallel blocks. The parameter $init$ describes the initial state for the planner. The $goal$ parameter determines which activity(ies) that have to be executed as goals for the planner. Moreover, the $condition$ parameter might be used to express extra constraints on the goal state of the planner. Finally, the $context$ parameter is the encoding of the domain knowledge. The domain knowledge is used by the planner to find a plan.

Before we go into details of violation resolution, we first explain about the way initial and goal states are calculated for the $findPlan$ function. Generally, for a rule $Source$ $leads$ to $Destination$, the $inital$ parameter reflects the execution of a set of activities from starting of the process up to and including the $source$ activity. $source$ is an occurrence of the Source activity in the rule, i.e., $source \in N_{Source}$. This is due to the fact that an activity may have more than one occurrence within the process model. We use the notion $source^{-}$ to reflect the execution history before $source$. Thus, calculating the initial state must include the execution history of all preceding activities from the start of the process up to and including the $source$. However, there might be more than one possible $initial$ state. This is due to the possibility of having choice blocks before the $source$ activity. In this case, each possible execution instance before $source$ is considered as an $initial$ state. Moreover, a plan has to be found for each of the possible instances in order to conclude that resolution is possible.

We assume that process models are consistent with the domain knowledge. In Section 4.4, we informally defined the notion of consistency between a process model and the domain knowledge. Here, we formally define that notion based on the process structure tree.

Definition 7.11. *[Business process consistency with the domain knowledge]*

A process model, its process structure tree, $PST = (N, E, r, t, cond)$ are consistent *with domain knowledge*
$\mathbb{C} = (N_{act}, A, T, asptype, con_{tt \in T}, pre_{tt \in T}, post_{tt \in T})$ *if:*

- Post conditions are used*:* $\forall n \in N_{act} \wedge \forall tt \in T \wedge post_{tt} \neq \emptyset \wedge \forall a \in A \wedge asptype(a) = tt \wedge (n, a) \in post_{tt} : n \in N \rightarrow (\exists m \in N_{act} : m \in N \wedge (m, a) \in pre_{tt} \wedge t(lca(m, n)) = \mathtt{seq} \wedge order(lca(m, n), n) < order(lca(m, n), m)) \vee (\nexists f \in N : order^{*}(r, n) < order^{*}(r, f) \wedge t(f) = \mathtt{act})$*,*

- No two contradicting activities execute in the same instance*:* $\forall n, m \in N_{act} \wedge (n, m) \in con_{t}t : N_{n} = \emptyset \vee N_{m} = \emptyset \vee \forall n' \in N_{n} \forall m' \in N_{m} t(lca(n', m')) = \mathtt{xor} \wedge \nexists s \in N(t(s) = \mathtt{loop} \wedge s \in path(r, lca(n', m')))$*.*

Definition 7.11 ensures the consistency between a process model and its PST on one hand, and the domain knowledge on the other hand by making sure that for any postcondition produced by an activity n, there is an upcoming activity m that will use

Activity	Precondition	Postcondition	
		Negative	Positive
Go to checkout	order [init]	order [init]	order [conf] pay. meth. [card]
Go to checkout	order [init]	order [init]	order [conf] pay. meth. [transfer]
Notify customer	order [conf] payment [received]	notification [init]	notification [sent]
Pay by credit card	pay. meth. [card] card data [filled]	payment [init]	payment [received]
Pay by bank transfer	pay. meth. [transfer] bank data [filled]	payment [init]	payment [received]
Provide bank data	pay. meth. [transfer] bank data [init]	bank data [init]	bank data [filled]
Provide credit card data	pay. meth. [card] card data [init]	card data [init]	card data [filled]
Prepare goods	order [conf]	goods [init]	goods [prepared]
Send goods	address [filled] payment [received] goods [prepared]	goods [prepared]	goods [sent]
Provide shipping address	order [conf] address[init]	address[init]	address [filled]
Cancel order	order [init]	order [init]	order [canceled]
Archive order	order [conf] goods [sent] payment [received]	order [conf]	order [archived]
Archive order	order [canceled]	order [canceled]	order [archived]

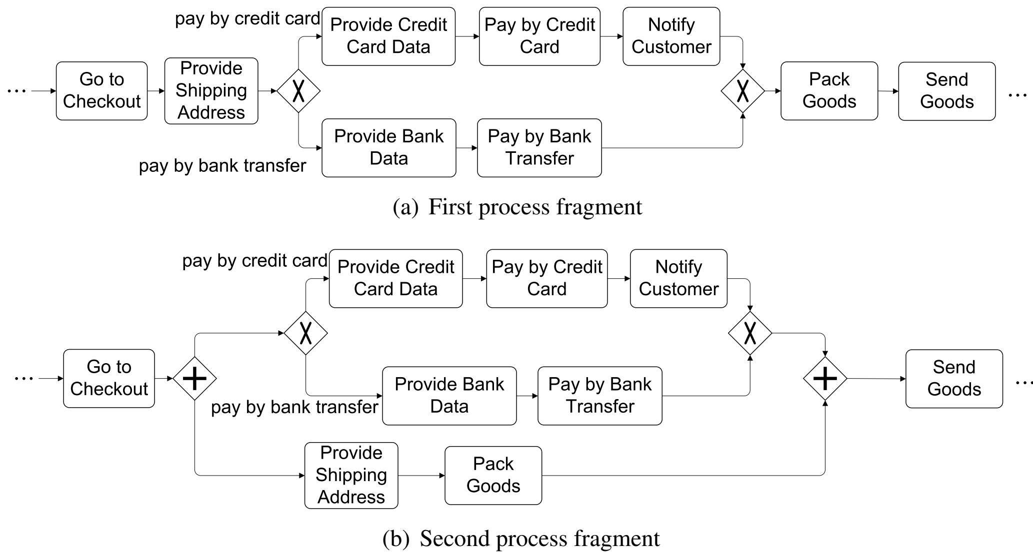
Table 7.1: Pre and post relations of the example domain knowledge

that output or the activity n is one of the last activities to execute in the process model. This indirectly states that the process will not deadlock due to unmet data conditions [8]. Also, for any two contradicting activities, there is no chance to execute them both in the same process instance. This is guaranteed by 1) either one of them appears in the process model, 2) if both appear they have to be on different branches of an XOR block and without any surrounding loop blocks.

7.3.1 Example Domain Knowledge

We introduce an example, to be used throughout the rest of the chapter to illustrate the ideas. The example includes the domain knowledge and business process fragments. The domain knowledge is defined by the tuple $(N_{act}, A, T, asptype, con_{t\in T}, pre_{t\in T}, post_{t\in T})$. The set of activities N_{act} is formed by {*Go to checkout*, *Notify customer*, *Pay by credit*

card, *Pay by bank transfer*, *Provide bank data*, *Provide credit card data*, *Prepare goods*, *Send goods*, *Provide shipping address*, *Cancel order*, *Archive order*}. In the example we consider the data aspect. Hence, set T contains one element *data*. The set of objects A is the set of data objects {*address*, *bank data*, *card data*, *goods*, *order*, *notification*, *payment*, *payment method*}*. Subsequently, function $asptype$ relates each of the data objects to type *data object*. Table 7.1 captures $pre_{t \in T}$ and $post_{t \in T}$ relations. An activity may have more than one pre- or postcondition. For instance, activity *Archive order* expects either a confirmed order, received payment, and sent goods, or it expects a canceled order. In this way, it is possible to express disjunctive preconditions. Activities *Pay by credit card* and *Pay by bank transfer* are the only contradicting activities. Based on the above



(a) First process fragment

(b) Second process fragment

Figure 7.5: Two process fragments consistent with the context

domain knowledge, the two process fragments shown in Figure 7.5 are consistent with the context.

7.3.2 Resolving Splitting Choice Violation

A violation resolution implies that a process model is modified in such a way that activity B is always executed after A. We aim at introducing local modifications to the model. Hence, we first identify the smallest fragment of a model, whose modification can be sufficient for the violation resolution. Afterwards, the fragment is modified in the way that execution of B is assured. In general, the following operations allow to resolve splitting choice violation:

1. Remove activity A from the process model;

*We omitted the data object states to avoid duplicates as they are present in Table 7.1

2. Move activity A to the branch with B directly before activity B;
3. Add activity B to every branch in the choice block;
4. Remove all the branches which do not contain activity B;
5. Add activity B in between activity A and the choice block, directly after A;
6. Add activity B directly after the block;
7. Move activity B in between activity A and the choice block;
8. Move activity B directly after the choice block

Removing activity A from the process model would achieve vacuous compliance. However, apart from the inconsistency that might appear due to removing A, in the specific case of splitting choice violation, the process logic already allows executing B after A in *some* cases. Thus, we should look for a way to execute B in all cases. Thus, the first two alternatives are not suitable in this case.

Adding B to every branch where it is missing would solve the problem. In any exclusive thread after A there is a chance to execute B. However, we have to make sure that there are no contradictions between B and other activities on branches lacking B, see Definition 7.11. If for a branch it is not possible to add B, the branch has to be removed, the fourth alternative. This could be justified by the fact that compliance requirements, in general, restrict the behavior of processes and they are supreme to the logic in the process. However, to guarantee consistency of the process, activities behavior, i.e., post conditions, has to be changed to guarantee that the process does not deadlock. That is, for each branch to be removed, the branch condition has to be analyzed in order to identify activities which produce these conditions as effects. Afterwards, the behavior of these activities has to be modified in order to be sure that unused effects will not be produced. The role of automated resolution is to identify the conditions and activities respectively. Modifications to activity behavior are considered as a task of a human expert.

The remaining alternatives duplicate or move the activity B some point after A out of the choice block. Duplication might not be possible in every case. For instance, if the activity B is about payment, it is not possible to duplicate it. Also, duplication or movement raises the issue of consistency again. In both cases, there is a chance that B will be in sequence with one of its contradicting activities on the other branches of the choice block.

Algorithm 22 provides our approach to resolve this violation. Initially, a block enclosing occurrences of activities A and B is sought, see line 1. If the enclosing block is a loop, the algorithm has to assure that B is executed after the loop. Automated planning attempts to construct a plan containing activity B. If the plan is constructed, it is inserted exactly after the loop block. Otherwise, the resolution cannot be performed. If the block is not a loop, we seek inside it for a choice block containing b on its branch, line 9. For each

```
    input  : m—process structure tree
    input  : a, b ∈ N—the occurrences of activities A and B, respectively
    input  : c—domain knowledge
    output : (m,r)—(updated process structure tree, restriction to activities behavior or
             conditions under which planning failed)
 1  s = lca (a,b) ;
 2  if t(s) = loop then
 3      plan = findPlan (a⁻, b, cond (s), c);
 4      if plan = ∅ then
 5          add a⁻ to r;
 6          add cond (s) to r;
 7          return (null,r);
 8      insert plan into m exactly after the loop exit;
 9  else if t(s) = seq then
10      x is the choice block containing b;
11      forall the branch is a branch of x with no b do
12          if branch has activities contradicting b then
13              remove branch from m ;
14              add cond (branch) to r ;
15          else
16              plan = findPlan (a⁻, b, cond (branch), c) ;
17              if plan = ∅ then
18                  remove branch from m;
19                  add cond (branch) to r ;
20              else
21                  add plan to m merging it into branch;
22  return (m,r);
```

Algorithm 7.1: Resolving splitting choice violation

branch missing B, we check if it has activities contradicting B, with respect to the domain knowledge $\mathbb{C}$. If a contradiction exists, the branch is removed, line 13. Meanwhile, the branch conditions are identified as restrictions to activity behavior, line:14. In case of no contradictions, we use automated planning based on the information of the domain knowledge to find a path from a to b under the branch condition, see line 16. If no plan could be found, the branch is removed from the model and the branch condition is added to the restrictions. Otherwise, the found plan is merged to the branch to enforce compliance.

Let us turn to the example process fragment in Figure 7.5. The process fragment violates the compliance rule *Go to checkout* `leads to` *Notify customer*. *Go to checkout* activity is succeeded by the choice block. While one branch of the choice block contains activity *Notify Customer*, the other does not. According to Algorithm 22, an occurrence of activity *Notify customer* is added to the branch where it was missing. The occurrence is added to the branch after activity *Pay by Bank Transfer*. On the other hand, a rule

Go to checkout `leads to` *Pay by credit card* is also violated. However, to resolve this violation using Algorithm 22, the lower branch has to be removed. In this case, the branch condition `payment method[transfer]` is identified as a restriction. That is, the resulting process is consistent only in the case that the postcondition `payment method[transfer]` of activity *Go to checkout* is removed.

7.3.3 Resolving Different Branches Violation

As stated earlier, a violation of this type takes place if the two activities are allocated on different branches of a block, independent of a block type, i.e., either an XOR or an AND block. However, the resolution strategy varies depending on the block type. The following operations allow to resolve the violation:

1. Remove activity A from the process model;
2. Add activity B to the branch with A directly after activity A (for `xor` block);
3. Move activity B directly after the block (for `and` block);
4. Move activity A directly before block (for `and` block)

In case that activity A and B are on different branches of an XOR block, removing A from the process solves the problem. But, we have to remove the whole branch where A belongs, to be sure that no remaining activities without unmet inputs. However, as discussed earlier, removing a branch necessitates making restrictions to preceding activity behavior. Otherwise, the process would deadlock due to unmet inputs.

The second alternative is to add B after A. This is possible in case there are no contradicting activities to B on that branch. However, we handle these situations as a lack of activity as we will show later. If this succeeds, it is preferable to the situation of removing A from the process model.

In case of an AND block, the resolution strategy aims at sequentializing A and B. To achieve the sequential execution of A and B, we move an occurrence of B from a block branch to the position exactly after the block. However, such a manipulation with an occurrence of B might introduce inconsistencies into the process model: there might be activities on the branch expecting B in the initial place. Hence, we move not only an occurrence of B, but the set of activities succeeding B on the branch and depending on B. An alternative strategy is to move A, together with preceding activities on which A depends, exactly before the block. The preference to one of these strategies can be given basing on the number of activities to be moved.

Algorithm 18 summarizes the resolution for different branches violation. First, the type of block is identified, line 1. In case of XOR block, the algorithm checks whether it is possible to add a new occurrence of B after A, by calling the lack of activity resolution algorithm. If there is a solution, the resulting plan is merged with the process model. Otherwise, the branch containing A is removed. In this case, the branch condition is

input : m—process structure tree
input : a, b $\in N$—the occurrences of activities A and B, respectively
input : c—domain knowledge
output : (m,r)—(updated process structure tree, restriction to activities behavior or conditions under which planning failed)

```
1  s = lca (a,b) ;
2  switch type(s) do
3      case AND
4          PRE_a is the set of all nodes that execute before a within the same thread;
5          POST_b is the set of all nodes that execute after b within the same thread;
6          if |PRE_a| <|POST_b| then
7              move PRE_a before the parallel block;
8          else
9              move POST_b after the parallel block;
10     case XOR
11         forall the branch is a branch of s with a, but no b do
12             result =resolve Lack of activity violation;
13             if result =∅ then
14                 remove branch from m;
15                 add condition (branch) to r ;
16             else
17                 merge result with m;
18 return (m,r);
```

Algorithm 7.2: Resolving different branches violation

identified as a restriction for activities behavior. On the other hand, if A and B are on different branches of an AND block, they are sequentialized.

Different branches violation is shown in Fig. 7.5(b), where *Pay by credit card* and *Pack goods* activities are executed in parallel. The resolution algorithm moves activity *Pack Goods* from the lower branch of the parallel block to the position between the AND join and *Send Goods* activity. Another case of different branches violation occurs when we have a rule *Pay by Bank Transfer leads to Notify Customer*. In this case, a resolution is by adding *Notify Customer* to the same branch as *Pay by bank transfer*, this is possible since *Pay by bank transfer* provides the `payment[received]` input of *Notify Customer*.

7.3.4 Resolving Inverse Order Violation

In case of inverse order violation, the process model semantics is opposite to the semantics of the rule and the process model requires considerable modifications. In general inverse order violation can be fixed with one of the following operations:

1. Remove activity A from the process model;

2. Add activity B directly after activity A;

```
input  : m—process structure tree
input  : a,b ∈ N—is the occurrences of A and B for which the violation has to be
         resolved
input  : c—domain knowledge
output : (m,r)—(updated process structure tree,restriction to activities behavior or
         conditions under which planning failed)
1  let pre be the node before b;
2  let post be the node after a;
3  plan1 = findPlan (pre⁻, a, ¬ b,c);
4  if plan1 = ∅ then
5      if ∃ s ∈ path(lca(a, b), a) ∧ t(s) = xor then
6          let x be the branch containing a;
7          remove x from m;
8          add condition (x) to r;
9      else
10         add pre⁻ to r;
11         return (null,r);
12 else
13     forall the n
       ∈ N ∧ order(lca(a, b), b) < order(lca(a, b), n) < order(lca(a, b), a) do
14         Delete n from m;
15     Merge plan1 with m;
16     plan2 =findPlan (a⁻, b ∧ post, true,c);
17     Merge plan2 with m;
18 return (m,r);
```

Algorithm 7.3: Resolving inverse order violation

3. Move activity A to the position directly before activity B;

4. Move activity B to the position directly after A

The decision to remove activity A from the process model can be pursued in case the removal does not affect the consistency. This could be the case where A is in an XOR block where other branches could be taken to complete the objective behind the process. As stated earlier, this puts restrictions on the behavior of activities. In other cases, where A is mandatory, i.e., not in a choice block, removing A hinders the consistency of the process model.

The second alternative, to add a new occurrence of B right after A, is not possible in cases like duplicating payment activities. For instance, a process model that has send goods and receive payment activities in inverse order. Otherwise, adding of a duplicate could be reduced to a lack of activity violation.

The remaining two alternatives investigate the possibility to reorder the occurrences of A and B. Reordering of the activities A and B is possible if there are no dependencies of activity A on B. If a dependency exists, reordering introduces inconsistencies into

the process model. To check whether reordering is possible, we attempt to construct a model fragment from the process start to the point where A is executed. To come up with this fragment, we construct a plan. In contrast to the initial model, activity B should not appear in this plan. The initial state of the planning task reflects the process state directly before activity B is executed. The goal state describes that activity A must be executed. If the planner comes up with the plan, the reordering is possible. Otherwise, there is no chance to reorder since actually A depends on the result of B.

Once reordering turns out to be possible, the resulting plan must be complemented to assure execution of B. The second planning task's initial state is the first plan's goal state. The new goal state describes the process state directly after an execution of B in the initial process. The two plans are inserted into the model. The model is free of contradictions and inconsistencies, since we just reordered activities, the resolution is completed.

Algorithm 18 summarizes our approach to resolve inverse order violation. First it tries to reorder A and B. If this is not possible and A is in a choice block. It removes A and identifies restrictions to activity behavior that are necessary to ensure a consistent result. On the other hand, if A is mandatory, the algorithm fails.

The inverse order violation can be illustrated by the process fragment in Fig. 7.5(a), where the company sends a notification with an order summary to a customer. Afterwards, the company contacts its logistics partner to pack and send goods. New business conditions might require the company to include the delivery information in the notification, i.e., first the goods should be packed. This requirement is captured in the rule *Pack goods* `leads to` *Notify customer* rule. This violation is of type *inverse order*.

In the example of *Pack goods* `leads to` *Notify customer* rule violation the resolution strategy moves the occurrence of *Notify Customer* activity to the position after *Pack Goods* activity.

7.3.5 Lack of Activity

A process model contains a violation to *A* `leads to` *B* of type lack of activity, if it contains at least one occurrence of A and no occurrence of B. The following operations allow to resolve lack of activity violation:

1. Remove activity A from the process model;
2. Add activity B directly before A

Consider a compliance rule *Go to checkout* `leads to` *Archive order*. Checking the process model fragment in Fig. 7.5(a) against this rule, we see that this rule is violated, since *Archive order* is missing in that fragment.

To resolve a violation of this type we introduce an occurrence of B into the process model exactly after an occurrence of A. If the process model does not contain activities contradicting to B, we construct a plan using *findPlan(A, B, true, context)*. The plan

```
input  : m—process structure tree
input  : a ∈ N—is the occurrences of A for which the violation has to be resolved
input  : b ∈ N—a new occurrence of B to be added to the tree
input  : c—domain knowledge
output : (m,r)—(updated process structure tree,restriction to activities behavior or
         conditions under which planning failed)
1  CON_b is the set of activities contradicting b based on c that appear in m;
2  if CON_b ≠ ∅ then
3      forall the x ∈ CON_b do
4          if type(lca(x,a)) = choice then
5              if findPlan (a⁻, b, true, c) ≠ ∅ then
6                  insert findPlan (a⁻, b, true, c) after a in m;
7              else
8                  add a⁻ to r;
9                  return (null,r);
10         else
11             Find pre, post;
12             if pre = null ∨ post = null then
13                 add pre = null ∨ post = null to r; return (null,r);
14             FRAG_x contains activity x and its tightly coupled activities;
15             FRAG_b contains activity b and the results of findPlan (pre⁻, b,
               true, c) and findPlan (b⁻, post, true, c);
16             insert FRAG_x and FRAG_b in a choice block between pre and post;
17 else
18     if findPlan (a⁻, b, true, c) ≠ ∅ then
19         insert findPlan (a⁻, b, true, c) after a in m;
20     else
21         add a⁻ to r;
22         return (null,r);
23 return (m,r);
```

Algorithm 7.4: Resolving lack of activity violation

is merged into the process model directly after an occurrence of A. In case there is an activity contradicting to B, let it be C, the resolution requires extra actions. The actions depend on the relations between occurrences of A and C:

1. occurrences of A and C are allocated on different branches of a choice block;
2. occurrences of A and C are allocated on different branches of a parallel block;
3. an occurrence of C is allocated before A;
4. an occurrence of C is allocated after A.

In the first case, B can be added to the process model exactly after A. As the branch with an occurrence of A does not contain activities contradicting B, B can be introduced to this branch without any conflicts.

In the latter three cases the process model contains occurrences of activities contradicting B. We propose to introduce an occurrence of B into the model in such a way that B and C appear on different branches of a choice block. We first seek for a SESE fragment containing an occurrence of C and activities tightly coupled with C. Such a process fragment contains activities which are transitively dependent on C or on which only C transitively depends. The fragment is preceded by an activity, let it be pre, and succeeded by an activity—$post$. We aim at complementing the process model with a branch, alternative to the identified fragment with C and containing B. We can obtain such a sequence as a result of a planning task, requiring it to fit between pre and $post$ and containing B. Finally, we introduce the choice block with the two branches into the model: the branch with C and the branch with plan containing B.

Identification of the activities dependent on C is based on the analysis of the domain knowledge and can be found as the closure of all the activities transitively depending on C. This closure is prefixed with an activity pre. To identify pre, we start from the activity immediately preceding C. If there is no such activity on the branch of C, we go in the tree one level up. Either we find an activity or we reach the topmost level indicating the start of the process. In the latter case, we cannot identify pre and thus it is not possible to insert B as an alternative. Otherwise, from pre it is possible to find a plan where B is executed. Similarly, we can find $post$, if there is one. Whenever pre or $post$ are empty, we are not able to proceed since the whole process in the closure of C.

As the result of the described model transformation, the violation type is no longer *lack of activity*. Instead, it changes either to *inverse order*, *splitting choice*, or *different branches* violation. The resolutions of these violation types have been already discussed. Notice that a *lack of activity* violation can be reduced to *different branches* violation and vice versa. However, there is no mutual dependency between them, as we reduce these violations to not intersecting subcases of violations. Algorithm 23 summarizes the approach.

Let us return to the example with a compliance rule *Go to checkout* `leads to` *Archive Order* and the process fragment in Fig. 7.5(a). According to the resolution strategy and information in the domain knowledge, activity *Archive Order* can be added after goods are sent.

The resolution strategy for *lack of activity* violation reduces the violation to the previous three cases: *inverse order*, *splitting choice*, and *different branches*. Hence, its properties, i.e., model compliance, freedom of contradictions and inconsistencies originate from the properties of resolution methods for the named violation types. Figure 7.6 illustrates how a lack of activity violation is reduced to other violation types and is resolved.

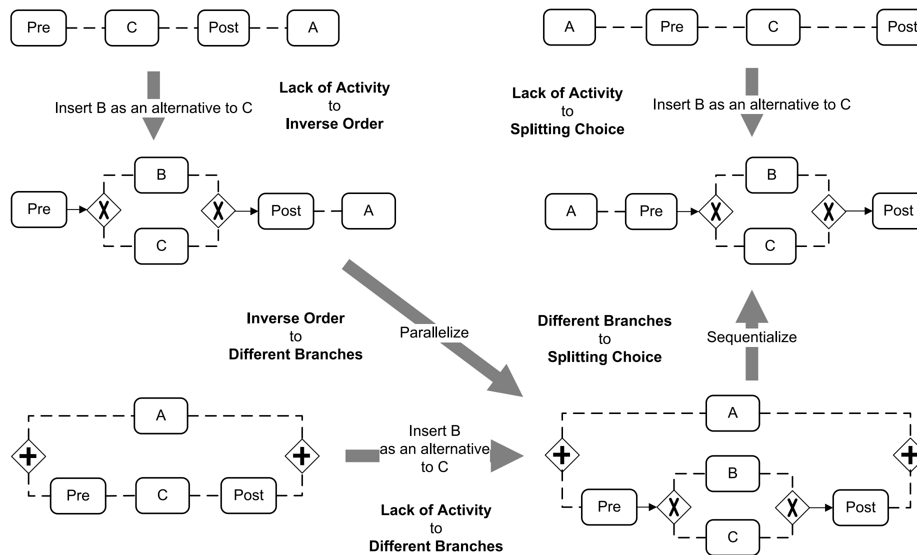


Figure 7.6: Resolving lack of activity violation in case of contradictions

7.3.6 The Overall Resolution Process

According to the discussion above, it might be the case that to resolve some violation is to first transform it from a violation type to another. For instance, in some cases of lack of activity violation, we transform it first to a splitting choice violation. Later on, we apply the algorithm of splitting choice to completely resolve the violation. Thus, the compliance violation resolution is of iterative nature. In each iteration, the violation type is recognized and the appropriate violation resolution algorithm is applied. This process is repeated until either no further violations are recognized or at one step the resolution algorithm either fails to resolve the violation or to transform it to another type.

Algorithm 15 summarizes our approach to resolve compliance violations. In each cycle, Algorithm 15 identifies the violation as either splitting choice, different branches, inverse order or lack of activity. Afterwards, the corresponding algorithm is invoked. Each algorithm returns a result on the form of a pair of an updated process structure tree and a set of conditions that either represent conditions that must be taken care of as restrictions to activity behaviors or conditions under which automated planning failed to find a solution.

The result in each cycle can be one of three possibilities. Firstly, the resolution algorithm returns a non empty compliant process structure tree and an empty restriction set. This case is a complete success to resolve the violation, using the available domain knowledge. The resulting process tree, and thus the process model, is guaranteed to be both compliant and consistent with the domain knowledge. The second possibility is that a non empty compliant tree and a non empty restriction set. In this case, the user is informed about the changes that have to be made to activities, post conditions that have to be removed in order to the returned tree to be consistent. This case actually

```
input  : m—process structure tree
input  : A Leads to B
input  : c—domain knowledge
output : (m,r)—(updated process structure tree, restrictions or conditions under
         which planning failed)
1  violation=getViolationType (m,A,B);
2  while violation ≠ none do
3      switch violation do
4          case Splitting Choice
5              (m, r) = resolveSplittingChoice(m, a, b, c);
6          case Different Branches
7              (m,r)=resolveDifferentBranches (m,a,b,c);
8          case Inverse Order
9              (m,r)=resolveInverseOrder (m,a,b,c);
10         case Lack of Activity
11             (m,r)=resolveLackActivity (m,a,b,c);
12     if m =null ∨ r ≠ null then
13         return (m,r);
14     violation=getViolationType (m,A,B);
15 return (m,r);
```

Algorithm 7.5: Compliance violation resolution

indicates some sort of contradiction between the logic implied by the compliance rule and the current understanding of the business domain. With this warning about possible contradictions, users might change their perception of the domain due to new compliance requirements or they can identify that there is a problem within the compliance rule itself. The third and last possibility of the resolution is an empty tree and a non-empty restriction set. In this case, the restriction set reports the conditions under which planning failed. In this case, there is a clear contradiction between what the common understanding of the business domain is and what is stated in the compliance rule. To overcome this obstacle, the user has to make major changes to the domain knowledge or to the compliance rule.

7.4 Directions to Resolve Violations to Other Patterns

So far, we focused on resolving violations to the after-scope presence pattern. In this section, we give an overview on how to approach the resolution of other patterns' violations.

The global-scope presence pattern is similar to the after-scope pattern, where an activity B has to be the response for the start of the process. The chances for violation are either *splitting choice* or *lack of activity*. Algorithms 22, 23 can be applied to resolve these violations.

The before-scope presence, *A precedes B*, violations are symmetric to the after-scope presence violations. Rather, the emphasis will be on the side of the activity A. The whole catalog of violation is applicable, with redefinition on the activity A side. However,

to enable automated planning, the pre and post conditions of activities, in the domain knowledge, is reversed. After a plan is found, it is reversed again to have it in the original flow.

A violation to an absence pattern, e.g., after-scope absence, occurs when there is a chance to execute A and B in parallel or in sequence, where A executes first. To resolve this violation, we might check first if B is optional in the process model. If this is the case, the whole XOR branch is removed. Following the discussion about Algorithm 22, we can argue that this solution results in a consistent process model, under certain restrictions. On the other hand, if B is mandatory, similar to the approach of resolving *lack of activity* violation, we try to identify the scope of effect of B. Using automated planning, we try to find a plan where B is avoided. Otherwise, it is not possible to resolve the violation automatically. Similar strategy could be used to handle violations to other absence patterns.

Resolving violations to conditional patterns is straightforward. Conditions can be reflected directly in the initial and/or goal states of the required plan. However, merging the resulting plan with the original process must explicitly add a choice block, whose condition is the same as the conditional rule, where the missing activities are added.

7.5 Discussion

A violation resolution implies that a process model structure is changed. To control model structural modifications we employ the concept of SESE fragments. As a consequence, the approach is limited to process models, which are block-structured. To neglect this limitation advanced decomposition techniques, as described in [110, 108], can be employed. However, these approaches do not promise to reveal structure in unstructured parts within processes. Thus, in case of unstructured parts of the process, user attention is drawn to that problem and it is on her side to manually resolve the violation.

Besides restoring compliance in business process models, another objective is to keep the modified process models operational. To maintain operation-ability, automated planning was employed to 1) identify whether it is possible to restore compliance 2) find sequences of activities that have to be added in order to restore compliance. For the first case, if it is not possible under the given compliance rule to find a sequence of activities execution that restores compliance, it is indicated to the user that there is an inherent contradiction between the compliance rule and the general domain knowledge. For the second case, if a sequence of activities is found, it is merged with the process model to make it both compliant and operational.

Although the changes introduced by the resolution algorithms result in consistent and compliant processes models, an updated model might look unnatural for a human reader. For instance, sequentialization of two branches in case of different branches violation, leaves only one branch in the parallel block. Thus, a resulting process model needs to be refactored to develop a naturally looking process model. Process refactoring techniques

proposed in [147] can be applied.

Chapter 8

Implementation

Developing prototypes is currently an important step for research in general and for the business process management community in specific. With an implementation, concepts can be better communicated with audience and more valuable feedback could be gained. Moreover, testing with real world scenarios refines and reshapes concepts.

This chapter describes a prototypical implementation of the contributions discussed in Chapters 4, 5 and 6. We describe an architecture where the user models compliance rules as BPMN-Q queries and at the end receives a decision about every relevant process whether it is compliant or not. Moreover, in case of non compliance the user is informed about parts of the process that cause the violation. Thus, we have a set of components to realize a compliance checking scenario. These components are

- Compliance rules editor (BPMN-Q query editor),
- BPMN-Q query processor,
- Business process model editor,
- Business process model repository,
- Mapping of process models to Petri nets,
- Petri net state space generation,
- Model checker.

Obviously, the objective is to keep implementing new components as minimal as possible. We had to implement the query processor as it is one of the core contributions of this thesis. Moreover, we had to implement an editor for creating compliance rules as queries. However, we needed to access a repository of process models to query and to check for compliance. For the process model to Petri net mapping, we followed the

mapping described in Definition 4.4. Low level Petri net Analyzer (Lola) [126] is a Petri net analysis tool and a model checker as well. Lola supports CTL model checking. Thus, to model check, it implicitly generate the Petri net state space.

Oryx* [29] is a Web-based business process modeling and repository developed at the Chair of Business Process Technology, Hasso Plattner Institute. With the front end, users can model processes with a variety of process modeling languages, e.g., BPMN, EPC, Petri net. Moreover, Oryx can be further extended by means of stencil sets, server-side and client-side plugins. The extension allows integrating research prototypes in a single framework. In our case, integrating BPMN-Q [124] and compliance checking steps into Oryx saves us the effort of building a new editor and a process repository.

To describe how we integrated BPMN-Q into Oryx and how we realized the compliance checking scenario, we start by describing the Oryx architecture and its extension mechanisms in Section 8.1. Integrating BPMN-Q query editor and query processor along with other software components to enable compliance checking is described in Section 8.2 also with description of the interactions between the different components. Section 8.3 describes a use case where the different steps of compliance checking are explained with snapshots from the prototype.

8.1 Oryx Architecture and Extensions

Oryx is a Web-based graphical modeling tool and a repository for process models. Using a standard web browser, users can login to Oryx, create, modify and share process models. Sharing means that the model creator can give access to users as read-only, read/write or can make process models public. Via plugins, more functionality can be added to the editor. For instance, process models can be checked for being error-free, step through support for process simulation, exporting process models to different formats.

Oryx does not only allow extensions by adding more functionality to existing ones. Rather, it allows defining new modeling languages. This is achieved via stencil set definition. A stencil set is a collection of files describing the abstract and concrete syntax of the modeling language. JavaScript Object Notation `json` files describe the abstract syntax of the language in terms of properties of nodes and edges used for modeling as well as connectivity rules among them. To describe the concrete syntax, Scalable Vector Graphics `svg` files describe how each node and edge will look like when the shape is dragged to the drawing canvas in Oryx.

Figure 8.1 describes the architecture of Oryx. Through a Web browser, the user can call Oryx. The core of Oryx is a set of JavaScript functions that is loaded in memory when Oryx is called. Depending on the requested process model, the process model, the stencil set and the associated plugins are loaded in memory. Client side plugins as well as standard editing functionality, e.g., copy, paste, etc., can access the memory copy of the process model. Oryx back end is the interface between the process model repository and

*http://oryx-project.org

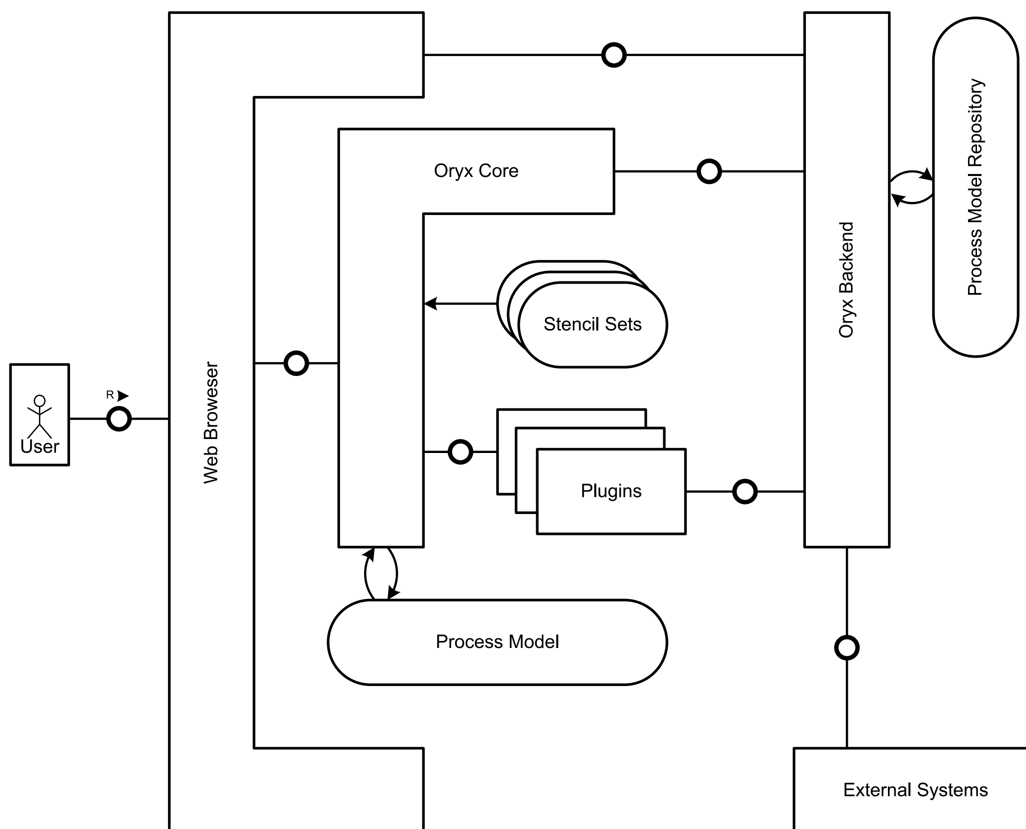


Figure 8.1: Oryx architecture

the front end. Each time a server side plugin is called communication occurs between the back end and the requester from the front end. An example server-side plugin is the mapping of BPMN models to Petri nets. In some cases, server-side plugins need to call external systems to Oryx.

8.2 Integrating BPMN-Q and Enabling Compliance Checking

The BPM-Q query processor is implemented in Java to match queries to process models as was described in Section 4.2.2. However, in order to let users specify compliance rules as queries, we developed a query editor, using a BPMN similar syntax, for BPMN-Q queries. This was achieved via defining a BPMN-Q stencil set that is available for Oryx users. While the stencil set is at the client side and the query processor is reachable via a server-side plugin, we implemented a client-side plugin that allows calling the query processor at the server side.

Lola is considered as an external system to Oryx that is called by the query processor for compliance checking. Before calling Lola, the query processor calls another sever-side

plugin, the BPMN to Petri net mapping, in order to obtain the Petri net representation of the process to be examined. The Petri net is then written to a `.net` file that is passed to Lola. Figure 8.2 shows how BPMN-Q and components necessary for compliance checking are fitted into Oryx architecture.

Figure 8.2: BPMN-Q architecture for Oryx

Figure 8.2 shows the static view on the components for compliance checking, in remaining part of this section we describe the sequence of calls to the different components in order to check for compliance.

First and foremost, the user draws the compliance query in the same way she draws a process model. A query can be saved in the repository for later use. Moreover, the user attaches tags to the query that will be used in identifying process models to check, as was stated in Section 4.2. To initiate the compliance checking, the user calls , via a tool bar button, the client-side plugin *Query Evaluator Initiator*. This step brings to the user a window where she selects the type of query she wants to apply. For compliance checking, the user has to select `Process Compliance Query`. After pressing the submit button, the query is serialized and shipped to the server-side plugin *Query Evaluator Servlet* as well as the querying command. That servlet instantiates the query processor and passes the query and the command to it. Within the query processor and based on the querying command, process models matching the tags associated with the query, cf. Definition 4.6, are imported from the Oryx repository and translated into the internal representation of the query processor. Complex queries are decomposed into the basic ones as was discussed in Section 5.2.6. For each basic rule, the investigated process is checked for structural compliance or non-compliance as was discussed in Section 5.4.2. If for any of the basic rules no structural decision can be made, we turn to model checking.

At that point, the query processor calls the *BPMN to Petri net Mapping* component, which implements the mapping described in Section 4.1.3, to obtain the Petri net representation for the investigated process model. In order to model check the process, we need to generate the reachability graph of the Petri net. Lola is the means to obtain the reachability graph as one of the things Lola can do. As stated earlier, Lola can be used as a model checker. Lola has to be configured correctly in order to do the job. To instruct Lola to work as a model checker, the `userconfig.H` file has to be edited where `#define MODELCHECKING` is uncommented. Afterwards, Lola can be invoked from the query processor with the following command `lola file.net`. the parameter `file.net` contains the Petri net specification in Lola format and the CTL formula to be checked. The CTL formula is obtained from the BPMN-Q query as was discussed in Chapter 5. Moreover, the CTL formulas are expressed on the markings of the places of the Petri net. For instance, the proposition $Evaluation_failed$, which describes that the "Evaluation" data object assumes the state $failed$, has to be replaced on the corresponding place within the net. Also, the proposition $ready_conductDueDiligence$ has to be replaced with the place whose marking means that the transition corresponding to that activity is enabled. Also, all implications in the formula on the form $p \rightarrow q$ have to be rewritten on the form $\neg p \vee q$. That is because Lola syntax does not accept implication symbols.

If model checking succeeds, i.e., the process is compliant, the process model uri and the information that the model is compliant is send back to the *Query Evaluator Servlet*. Otherwise, the query processor generates anti patterns, as was discussed in Chapter 6, for failing rules and matches them structurally to the investigated process. At this point, the query processor return the process model uri, the process sub-graph matching the anti pattern and the information that there is a violation to the *Query Evaluator Servlet*.

The above procedure is repeated for each process to be investigated. Upon completion of the compliance checking, the *Query Evaluator Servlet* receives information about compliant and non compliant processes. To this end, the servlet invokes the *Response Presenter* at the client side to screen the result to the user.

8.3 Example

In this section we describe an example starting with the modeling a compliance rule until the feedback is returned to the user.

Figure 8.3 shows the BPMN-Q editor within Oryx. Using the stencil set, we build a complex compliance rule. Indeed, the rule combines rules $R3$, $R4$ from Section 5.5.

To start the processing of the query, the user invokes the *Query Evaluation Initiator* plugin via the tool bar. The processing options window appears as shown in Figure 8.4.

To initiate compliance checking, the process compliance query option is chosen in the previous step. With this option, the query processor is instructed to investigate all candidate process models for compliance. Another option is to specify the investigation of a specific process model, `Run compliance query against a specific`

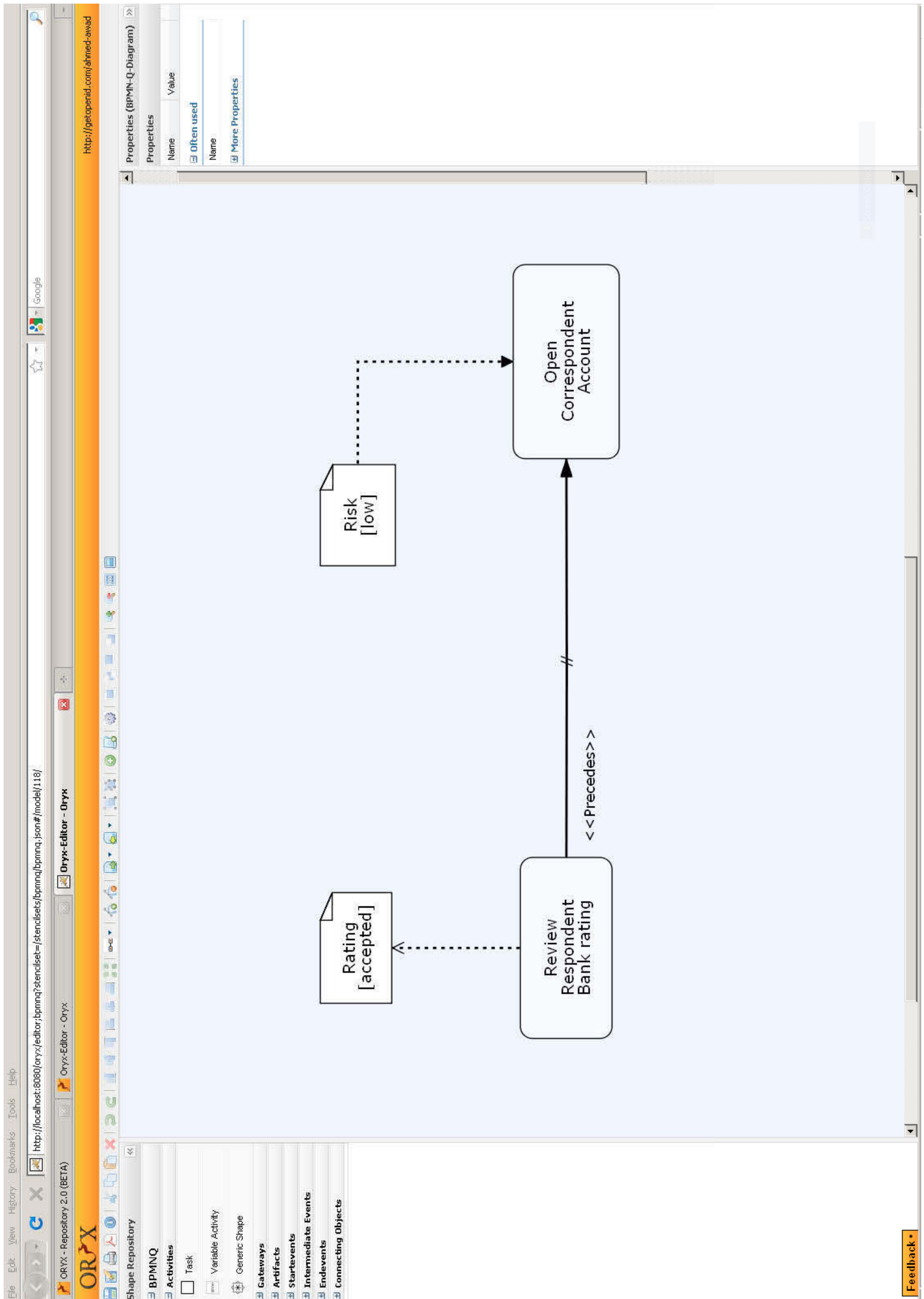


Figure 8.3: A compliance rule as a BPMN-Q query

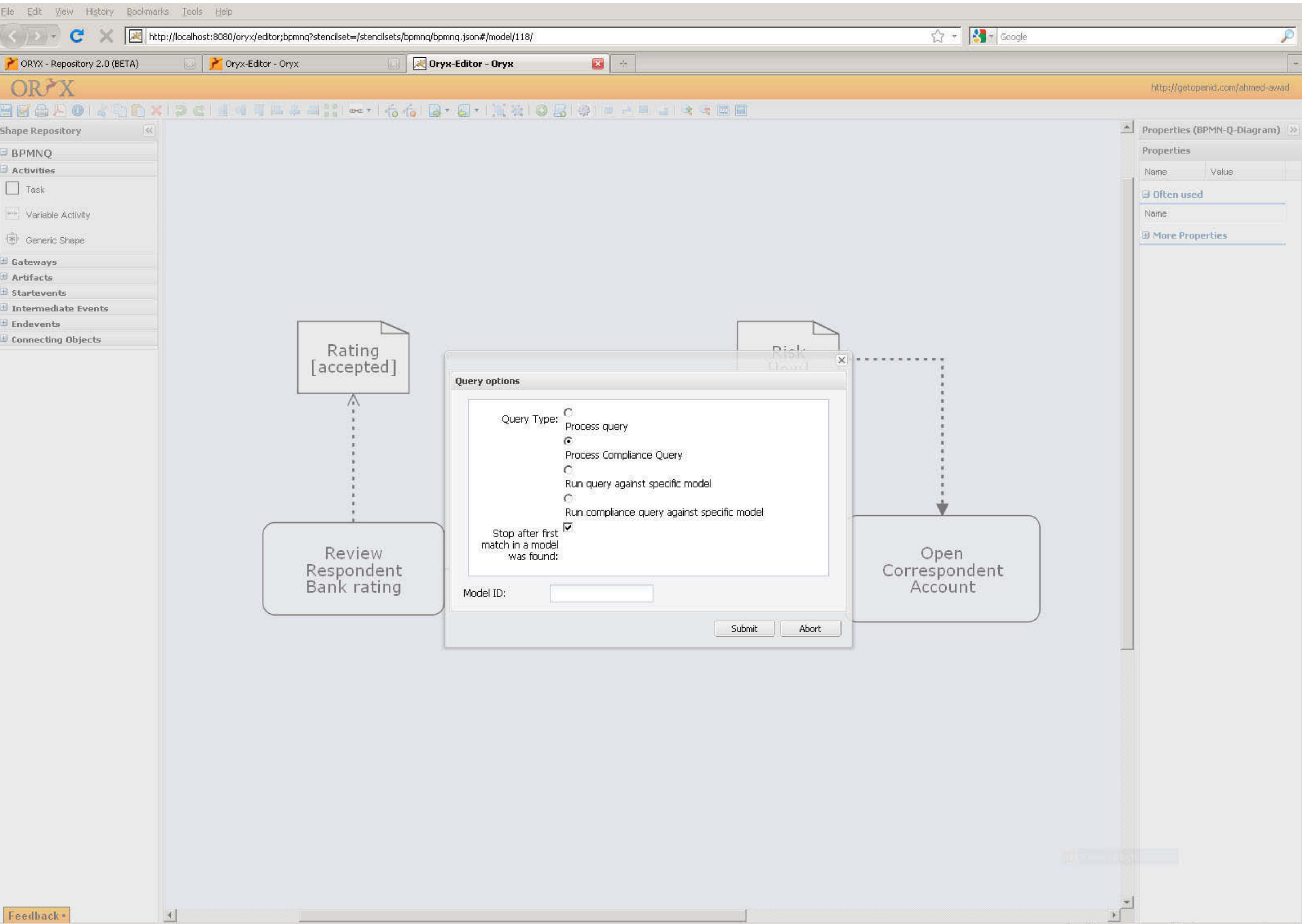


Figure 8.4: Processing options for queries

`process model`. In the latter case, the user is asked to enter the uri pointing to that specific process to be investigated.

With the pressing of the submit button, the query and the processing options are shipped to the back end. After doing the steps described in the previous section for compliance checking, the user gets feedback in the form of process model previews as shown in Figure 8.5. By double clicking on a certain preview, the user gets the feedback about that specific process, whether compliant or not. In case of violations, the violating part is highlighted as shown in Figure 8.6

As can be noticed, the generation of anti patterns is totally transparent to the user. Only the matches to the anti patterns are returned as highlighting of violating parts of the process to the user.

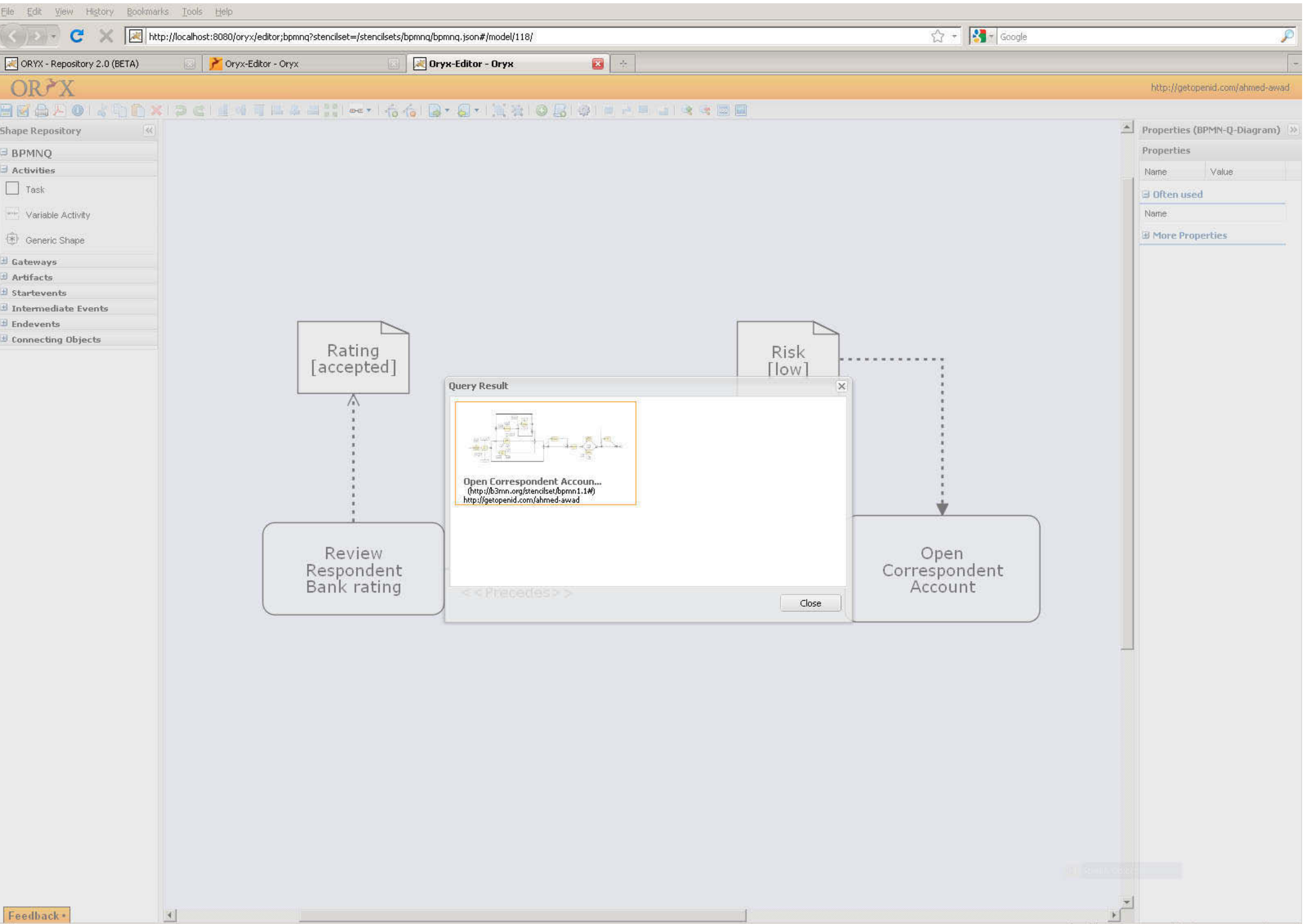


Figure 8.5: Results of evaluating the query in Figure 8.3

Figure 8.6: A violation to the query in Figure 8.3 highlighted on the process model

Chapter 9

Discussion

In this chapter we provide a critical discussion about the contributions in this thesis. Our discussion will illustrate strengths and limitations of our approach and revisit the assumptions introduced in Section 2.3. Also, we discuss how far we were aligned with the requirements discussed in Section 2.2.

Summary

We introduced a pattern-based approach for compliance checking. Using BPMN-Q visual queries to express compliance requirements, we have provided a graphical notation to express compliance rules, $Req.$ 9. The visual nature of the language, that is very close to the way process models are expressed, makes it easy for business people to understand and discuss about it. Also, each query is stored on its own as an artifact that can be associated with arbitrary metadata that allows tracking of a compliance requirement, $Req.$ 2. This helps the organization assess the number of control objectives it has established to meet compliance requirements. Finally, formal checking is possible since each pattern is mapped to a temporal logic formula, $Req.$ 1.

When a set of compliance rules are related to a process model, the conjunction of these rules are checked against the process. However, in some cases, a rule might be required to hold when another rule does not hold. That is, the compliance to one rule is required only when the process is violating another rule [103]. Currently this is not addressed. Modeling this reparation relationship requires the definition of a new modeling construct. On the technical level, we can check the disjunction of rules rather their conjunction.

Checking consistency among compliance rules, $Req.$ 3, was divided into redundancy and conflict checks. While conflict checking is complete, assuming that sufficient domain knowledge is present. Redundancy checking is based on establishing a sort of logical equivalence between the different compliance patterns. Thus, we cannot claim that redundancy checking is complete.

To explain violations to compliance rules, in general, we used a two step approach where the first step is to query the process behavior about the causes of the violation and in the second step BPMN-Q anti pattern queries are generated to highlight parts of the process causing violation. Although there is a processing overhead due to the need to issue several temporal logic queries, this could be justified by the need to be complete. That is, we need to provide the user with every possible cause of violation, $Req.$ 6. This, in turn, will save and focus the effort of the user compared to the alternative of the counterexample-driven violation explanation. In the latter case, the user has to repeat a check-explain-resolve each time model checking identifies a violation, which also contains a repeated cost of model checking.

To resolve violations, $Req.$ 7, we discussed several algorithms that were dedicated to special kinds of compliance violations. These approaches depend heavily on the domain knowledge. This is no surprise as the very first requirement to have an effective compliance management approach is to have a deep knowledge about the business domain [65]. While our approach to resolve violations is not complete, it helps the user identify areas of conflict and missing or incomplete knowledge about the domain.

Applicability

In Section 2.3, we have established a set of assumptions that make our approach applicable in real life. Here, we discuss the relaxation of some of these assumptions and how that relaxation affects the applicability of the approach. The success of establishing an effective compliance management is directly related to the level of maturity of the organization [66]. Thus, the applicability of our approach in real life depends on the level of maturity of the organization. The entry maturity level is that organizations have explicit views on their business operations in terms of process models [5, 65]. This has scoped the thesis from the beginning. However, in many cases, organizations might not have that level of maturity. In such cases, organizations might benefit from process mining techniques [136] to help automate the discovery of business processes. Yet, this depends on whether the organization has an automated support for its daily business operations. If that is not the case, there is no way but starting a manual business process discovery using traditional requirements elicitation techniques, as in [65].

To correlate compliance rules with business processes, we assumed the availability of a common set of tags that can be used to annotate both compliance rules and business processes. This is also is related to the level of maturity of the organization. That is, the agreement on a set of terms that have a common meaning. If this is not the case, users can benefit from the querying nature of BPMN-Q to identify process models that are subject to compliance checking. However, the success of this approach assumes a common glossary to label activities in both queries and processes.

We assume that the organization has a glossary of vocabularies, activity labels, that is agreed upon. For instance, an activity with the label “Open account” is understood by all people in the domain and is agreed upon its business value. If this is not the case, there

is a range of semi automated approaches that can fill this gap. For instance, in [10], we show how information retrieval (IR) techniques can be used to overcome this problem. Other approaches that use semantic annotations [69, 54] can be applied to annotate both activities in process models and in compliance queries. There are recent publications that look up labels of activities in a repository of business process models in order to find semantic similarity [129]. We believe that the result of such research will be beneficial to organizations and helps accelerate building their own glossary of activity labels.

Depending on the compliance rule and the domain knowledge, we might be able to suggest a remedy to the process model that restores compliance. With our semi automated approach we can find, in some cases, a resolution to the violation automatically. If this is not possible, we give the compliance officer directions about what is missing to resolve violations. We believe that this helps the compliance officer assess the amount of change needed within the organization's view on the business in order to achieve compliance. We admit that having the domain knowledge available in the form we suggest could only be available in very highly mature organizations. However, we believe that providing a guide about what knowledge to collect allows organizations to incrementally build their domain knowledge. We also see that this is a necessary step that helps organization gain more insight about its business operations and in the same time provides better capability to respond to future compliance requirements.

Compliance for Declarative Business Processes

Declarative business process modeling is a way to allow flexibility in processes. Processes are modeled by specifying a set of execution ordering constraints on a set of activities [107, 105]. Compliance rules discussed in this thesis can be integrated to provide a compliant execution of process instances. In that case, having an execution engine that is faithful to the execution constraints, including compliance rules, there is no chance for violation. Thus, there is no need to check for violation and will be no need also to resolve them. However, there is still the need to identify inconsistencies between rule sets.

The compliance patterns we identified overlap with those identified by Pesic in [105]. Yet, data flow and conditional rules are unique to our approach. When integrated with declarative approaches, more expressiveness is gained to provide finer grain restrictions on process execution.

Future Work

There are open issues that are subject to future work.

Business Process Synthesis

As compliance requirements and business objectives are developed separately, there is a chance of conflicts. Our approach to check conflict freedom of a set of compliance rules relied on finding computations, Büchi automaton, that satisfies the conjunction of

the rules. The idea was to decide conflict freedom if the resulting automaton has an accepting run. However, the decision was bound by the availability of sufficient domain knowledge. We see an opportunity to help gain better understanding of the domain and better communication between compliance experts on the one hand and business experts on the other hand by means of process synthesis. From a Büchi automaton, we can derive a process template that can serve as an artifact for the communication. Both the compliance expert and the process expert can decide whether the resulting template is of value. The more precise the resulting template the better communication can be reached among experts. The technical approach to generate process template out of compliance specification is seen as a future work.

Supporting Compliance at Other Process Life Cycle Phases

Under the assumption that organizations strictly follow their documented process models either in the form of automated processes or by work procedures followed by employees, we can assume a compliant execution of processes, when violations are discovered and corrected at process definition. However, due to the dynamic nature of execution environments, it is likely to have exceptions where violation can occur.

To guarantee compliance at automated process execution, we can foresee two directions to *monitor* the status of compliance. The first approach is via instant monitoring of running processes. We believe that work in [115] constitutes the first step to realize an instant monitoring scheme. Another approach benefits from the well developed process mining [140, 138] techniques to assess the compliance of completed instances. The integration with process mining can widen the scope of applicability to organizations that do not have explicit process execution engines. The major difference between the two approaches above is that with instant monitoring we are still able to prevent violations or at least have an informed violation in extreme cases. On the other hand, with process mining, we can only report about violations after they have occurred.

Using execution logs, it is also possible not only to identify violations on process definition level, the case of process mining, but also to quantify the number of violations, in terms of specific instances that have violations. In that case, querying rather than mining the logs is needed. This can be approached by the notion of anti patterns introduced in this thesis. An anti pattern can be mapped to some sort of a query that is checked against the execution log. The number of matches identify the amount of violating instances. We believe that this can be then integrated into business process intelligence tools to provide top management with a realistic view of their compliance status.

Compliance Verification Enhancement

On a technical level, regarding the thesis in hand, there is a need to enhance the verification approach discussed in this thesis. Model checking was the main technique to decide about compliance, although we discussed simple structural checks about non compliance.

However, model checking is known to suffer from state space explosions. Within process models, the cause of state space explosion is having parallel threads. In literature, partial order reduction techniques were developed to overcome this problem when verifying properties concerned with deadlock-freedom of processes. The techniques were simply concerned with picking only one sequence of states from the start of the parallel thread to its end rather than needlessly investigating all possible sequences. While this is acceptable for deadlock-freedom checking, it is not suitable for properties related to compliance rules. For instance, if we check a rule on the form *A Leads to B* and it happens that A and B are on different branches in a parallel thread, using partial order reduction can sometimes mistakenly report compliance if the sequence in which B executes after A is picked.

To overcome this problem, we believe that a hybrid approach for checking is needed. That is, structural checking should be investigated first. The approach of using structural decomposition techniques is a promising starting point. Actually, the violation catalog discussed in Chapter 7 constitutes the starting point. Depending on the compliance rule, e.g., control flow rules, and the degree of structuredness of business process models, compliance can efficiently be decided. In case it is not possible to structurally decide, only the unstructured part of the process models needs to be investigated via state space exploration. However, there are still open points regarding conditional rules and how to correctly map unstructured parts of the process to behavioral models. This is left for future work.

Bibliography

[1] Business Process Modeling Notation 1.2 (BPMN 1.2) Specification, Final Adopted Specification. Technical report, OMG, 2009. [cited at p. 19, 34, 41, 42, 44, 57]

[2] COMPAS: Compliance-driven Models, Languages, and Architectures for Service, 2008. 7th Framework Programme Information and Communication Technologies. [cited at p. 26]

[3] W.M.P. van der Aalst. Verification of Workflow Nets. In *Application and Theory of Petri Nets 1997, volume 1248 of Lecture Notes in Computer Science*, pages 407–426, Berlin, 1997. Springer Verlag. [cited at p. 5, 18, 44]

[4] Ashish Agrawal, Mike Amend, Manoj Das, Mark Ford, Chris Keller, Matthias Kloppmann, Dieter König, Frank Leymann, Ralf Müller, Gerhard Pfau, Karsten Plösser, Ravi Rangaswamy, Alan Rickayzen, Michael Rowley, Patrick Schmidt, Ivana Trickovic, Alex Yiu, and Matthias Zeller. *WS-BPEL Extension for People (BPEL4People), Version 1.0*. 2007. [cited at p. 34]

[5] Rakesh Agrawal, Christopher M. Johnson, Jerry Kiernan, and Frank Leymann. Taming Compliance with Sarbanes-Oxley Internal Controls Using Database Technology. In Ling Liu, Andreas Reuter, Kyu-Young Whang, and Jianjun Zhang, editors, *ICDE*, page 92. IEEE Computer Society, 2006. [cited at p. 38, 180]

[6] Marco Alberti, Federico Chesani, Marco Gavanelli, Evelina Lamma, Paola Mello, Marco Montali, and Paolo Torroni. Expressing and Verifying Business Contracts with Abductive Logic Programming. *Int. J. Electron. Commerce*, 12(4):9–38, 2008. [cited at p. 36]

[7] Ahmed Awad. BPMN-Q: A Language to Query Business Processes. In Manfred Reichert, Stefan Strecker, and Klaus Turowski, editors, *EMISA*, volume P-119 of *LNI*, pages 115–128. GI, 2007. [cited at p. 18, 52, 55]

[8] Ahmed Awad, Gero Decker, and Niels Lohmann. Diagnosing and repairing data anomalies in process models. In Stefanie Rinderle-Ma, Shazia Sadiq, and Frank Leymann, editors, *BPM 2009 Workshops*, volume 43 of *Lecture Notes in Business Information Processing*, pages 5–16. Springer-Verlag, September 2009. [cited at p. 44, 45, 48, 154]

[9] Ahmed Awad, Gero Decker, and Mathias Weske. Efficient Compliance Checking Using BPMN-Q and Temporal Logic. In Marlon Dumas, Manfred Reichert, and Ming-Chien Shan, editors, *BPM*, volume 5240 of *Lecture Notes in Computer Science*, pages 326–341. Springer, 2008. [cited at p. 67]

[10] Ahmed Awad, Artem Polyvyanyy, and Mathias Weske. Semantic Querying of Business Process Models. In *EDOC*, pages 85–94. IEEE Computer Society, 2008. [cited at p. 52, 181]

[11] Ahmed Awad and Frank Puhlmann. Structural Detection of Deadlocks in Business Process Models. In Witold Abramowicz and Dieter Fensel, editors, *BIS*, volume 7 of *Lecture Notes in Business Information Processing*, pages 239–250. Springer, 2008. [cited at p. 52]

[12] Ahmed Awad, Sergey Smirnov, and Mathias Weske. Resolution of compliance violation in business process models: A planning-based approach. In Robert Meersman, Tharam S. Dillon, and Pilar Herrero, editors, *OTM Conferences (1)*, volume 5870 of *Lecture Notes in Computer Science*, pages 6–23. Springer, 2009. [cited at p. 152]

[13] Ahmed Awad, Sergey Smirnov, and Mathias Weske. Towards Resolving Compliance Violations in Business Process Models. In Shazia Sadiq, Marta Indulska, and Michael zur Muehlen, editors, *GRCIS*, volume 459, pages 18–32. CEUR-WS.org, 2009. [cited at p. 147]

[14] Ahmed Awad, Matthias Weidlich, and Mathias Weske. Specification, verification and explanation of violation for data aware compliance rules. In Luciano Baresi, Chi-Hung Chi, and Jun Suzuki, editors, *ICSOC/ServiceWave*, volume 5900 of *Lecture Notes in Computer Science*, pages 500–515. Springer, 2009. [cited at p. 67]

[15] Ahmed Awad, Matthias Weidlich, and Mathias Weske. Consistency Checking of Compliance Rules. In *BIS*, volume 47 of *Lecture Notes in Business Information Processing*, pages 106–118. Springer, 2010. [cited at p. 93]

[16] Ahmed Awad and Mathias Weske. Visualization of compliance violation in business process models. In Stefanie Rinderle-Ma, Shazia Sadiq, and Frank Leymann, editors, *BPM 2009 Workshops*, volume 43 of *Lecture Notes in Business Information Processing*, pages 182–193. Springer, 2009. [cited at p. 67]

[17] Jos C. M. Baeten and W. P. Weijland and. *Process Algebra*. Cambridge Tracts in Theoretical Computer Science Series. Cambridge University Press, 1990. [cited at p. 41]

[18] Roderick Bloem, Kavita Ravi, and Fabio Somenzi. Efficient decision procedures for model checking of linear time logic properties. In Nicolas Halbwachs and Doron Peled, editors, *CAV*, volume 1633 of *Lecture Notes in Computer Science*, pages 222–235. Springer, 1999. [cited at p. 64]

[19] Riccardo Bonazzi and Yves Pigneur. Compliance Management in Multi-actor Contexts. In Shazia Sadiq, Marta Indulska, Michael zur Muehlen, Eric Dubois, and Paul Johannesson, editors, *Governance, Risk and Compliance in Information Systems*, volume 459, pages 91–105. CEUR-WS, 2009. [cited at p. 25, 89]

[20] Glenn Bruns and Patrice Godefroid. Temporal logic query checking. In *LICS '01: Proceedings of the 16th Annual IEEE Symposium on Logic in Computer Science*, page 409, Washington, DC, USA, 2001. IEEE Computer Society. [cited at p. 64, 65, 130]

[21] William Chan. Temporal-Locig Queries. In *CAV*, volume 1855 of *Lecture Notes in Computer Science*, pages 450–463. Springer, 2000. [cited at p. 64, 65, 130]

[22] Marsha Chechik and Arie Gurfinkel. TLQSolver: A Temporal Logic Query Checker. In *CAV*, volume 2725 of *Lecture Notes in Computer Science*, pages 210–214. Springer, 2003. [cited at p. 65, 130]

[23] Alessandro Cimatti, Edmund M. Clarke, Fausto Giunchiglia, and Marco Roveri. NUSMV: A new symbolic model checker. *International Journal on Software Tools for Technology Transfer*, 2(4):410–425, 2000. [cited at p. 68]

[24] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, 1999. [cited at p. 17, 60, 67, 93, 98]

[25] Financial Services Commission. Guidelines on anti-money laundering & counter-financing of terrorism, 2007. [cited at p. 4, 12, 13, 68, 103]

[26] Michael T. Cox, Héctor Muñoz-Avila, and Ralph Bergmann. Case-based planning. *Knowledge Eng. Review*, 20(3):283–287, 2005. [cited at p. 147]

[27] Florian Daniel, Fabio Casati, Vincenzo D'Andrea, Emmanuel Mulo, Uwe Zdun, Schahram Dustdar, Steve Strauch, David Schumm, Frank Leymann, Samir Sebahi, Fabien De Marchi, and Mohand-Said Hacid. Business Compliance Governance in Service-Oriented Architectures. In Irfan Awan, Muhammad Younas, Takahiro Hara, and Arjan Durresi, editors, *AINA*, pages 113–120. IEEE Computer Society, 2009. [cited at p. 26, 27]

[28] Pallab Dasgupta. *A Roadmap for Formal Property Verification*. Springer, 2006. [cited at p. 94, 95, 100]

[29] Gero Decker, Hagen Overdick, and Mathias Weske. Oryx - An Open Modeling Platform for the BPM Community. In Marlon Dumas, Manfred Reichert, and Ming-Chien Shan, editors, *BPM*, volume 5240 of *Lecture Notes in Computer Science*, pages 382–385. Springer, 2008. [cited at p. 170]

[30] Juliane Dehnert and Peter Rittgen. Relaxed soundness of business processes. In Klaus R. Dittrich, Andreas Geppert, and Moira C. Norrie, editors, *CAiSE*, volume 2068 of *Lecture Notes in Computer Science*, pages 157–170. Springer, 2001. [cited at p. 5, 18]

[31] Remco M. Dijkman, Marlon Dumas, and Chun Ouyang. Semantics and analysis of business process models in BPMN. *Inf. Softw. Technol.*, 50(12):1281–1294, 2008. [cited at p. 18, 45, 46]

[32] Yurdaer Doganata and Francisco Curbera. Effect of Using Automated Auditing Tools on Detecting Compliance Failures in Unmanaged Processes. In *BPM '09: Proceedings of the 7th International Conference on Business Process Management*, Lecture Notes in Computer Science, pages 310–326, Berlin, Heidelberg, 2009. Springer-Verlag. [cited at p. 38]

[33] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Patterns in property specifications for finite-state verification. In *ICSE*, pages 411–420, 1999. [cited at p. 67, 68, 71]

[34] Cindy Eisner and Doron Peled. Comparing Symbolic and Explicit Model Checking of a Software System. In *Proceedings of the 9th International SPIN Workshop on Model Checking of Software*, pages 230–239, London, UK, 2002. Springer-Verlag. [cited at p. 67]

[35] E. Allen Emerson and Joseph Y. Halpern. Decision procedures and expressiveness in the temporal logic of branching time. *J. Comput. Syst. Sci.*, 30(1):1–24, 1985. [cited at p. 61, 67]

[36] Gregor Engels, Christian Soltenborn, and Heike Wehrheim. Analysis of uml activities using dynamic meta modeling. In Marcello M. Bonsangue and Einar Broch Johnsen, editors, *FMOODS*, volume 4468 of *Lecture Notes in Computer Science*, pages 76–90. Springer, 2007. [cited at p. 31]

[37] Amy P. Felty and Kedar S. Namjoshi. Feature specification and automated conflict detection. *ACM Trans. Softw. Eng. Methodol.*, 12(1):3–27, 2003. [cited at p. 94]

[38] Christian Flender and Thomas Freytag. Visualizing the Soundness of Workflow Nets. In *13th Workshop Algorithms and Tools for Petri Nets, AWPN 2006*, pages 47–52, 2006. [cited at p. 113]

[39] Alexander Förster, Gregor Engels, and Tim Schattkowsky. Activity Diagram Patterns for Modeling Quality Constraints in Business Processes. In Lionel C. Briand and Clay Williams, editors, *MoDELS*, volume 3713 of *Lecture Notes in Computer Science*, pages 2–16. Springer, 2005. [cited at p. 18, 31, 35, 67]

[40] Alexander Förster, Gregor Engels, Tim Schattkowsky, and Ragnhild Van Der Straeten. A pattern-driven development process for quality standard-conform business process models. In *IEEE Symposium on Visual Languages and Human-Centric Computing VL*, 2006. [cited at p. 31, 35]

[41] Alexander Förster, Gregor Engels, Tim Schattkowsky, and Ragnhild Van Der Straeten. Verification of Business Process Quality Constraints Based on Visual Process Patterns. In *TASE*, pages 197–208. IEEE Computer Society, 2007. [cited at p. 31, 35]

[42] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, 1995. [cited at p. 67]

[43] Paul Gastin and Denis Oddoux. Fast LTL to Büchi Automata Translation. In *CAV*, volume 2102 of *Lecture Notes in Computer Science*, pages 53–65. Springer, 2001. [cited at p. 93]

[44] Rob Gerth, Doron Peled, Moshe Y. Vardi, and Pierre Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *In Protocol Specification Testing and Verification*, pages 3–18. Chapman & Hall, 1995. [cited at p. 93]

[45] Sepideh Ghanavati, Daniel Amyot, and Liam Peyton. Towards a Framework for Tracking Legal Compliance in Healthcare. In John Krogstie, Andreas L. Opdahl, and Guttorm Sindre, editors, *CAiSE*, volume 4495 of *Lecture Notes in Computer Science*, pages 218–232. Springer, 2007. [cited at p. 38]

[46] Aditya Ghose and George Koliadis. Auditing business process compliance. In Bernd J. Krämer, Kwei-Jay Lin, and Priya Narasimhan, editors, *ICSOC*, volume 4749 of *Lecture Notes in Computer Science*, pages 169–180. Springer, 2007. [cited at p. 33, 35]

[47] Christopher Giblin, Alice Y. Liu, Samuel Müller, Birgit Pfitzmann, and Xin Zhou. Regulations Expressed As Logical Models (REALM). In *Proceeding of the 2005 conference on Legal Knowledge and Information Systems*, pages 37–48, Amsterdam, The Netherlands, The Netherlands, 2005. IOS Press. [cited at p. 36]

[48] Christopher Giblin, Samuel Müller, and Birgit Pfitzmann. From Regulatory Policies to Event Monitoring Rules: Towards Model-Driven Compliance Automation. Research Report RZ 3662, IBM Zurich Research Laboratory, 2006. [cited at p. 36]

[49] Adrian Giurca, Dragan Gasevic, and Kuldar Taveter. *Handbook of Research on Emerging Rule-Based Languages and Technologies: Open Solutions and Approaches*, volume I. INFORMATION SCIENCE REFERENCE, May 2009. [cited at p. 28, 89]

[50] Stijn Goedertier and Jan Vanthienen. Designing Compliant Business Processes with Obligations and Permissions, 2nd Workshop on Business Processes Design (BPD'06), Proceedings. In Johann Eder and Schahram Dustdar, editors, *Business Process Management Workshops*, volume 4103 of *Lecture Notes in Computer Science*, pages 5–14. Springer Verlag, 2006. [cited at p. 29]

[51] Guido Governatori, Jörg Hoffmann, Shazia Wasim Sadiq, and Ingo Weber. Detecting regulatory compliance for business process models through semantic annotations. In Danilo Ardagna, Massimo Mecella, and Jian Yang, editors, *Business Process Management Workshops*, volume 17 of *Lecture Notes in Business Information Processing*, pages 5–17. Springer, 2008. [cited at p. 31]

[52] Guido Governatori and Zoran Milosevic. Dealing with contract violations: formalism and domain specific language. In *EDOC*, pages 46–57. IEEE Computer Society, 2005. [cited at p. 29, 36]

[53] Guido Governatori, Zoran Milosevic, and Shazia Sadiq. Compliance checking between business processes and business contracts. In *EDOC*, pages 221–232, Washington, DC, USA, 2006. IEEE Computer Society. [cited at p. 31, 35]

[54] Guido Governatori and Shazia Sadiq. *Handbook of Research on BPM*, chapter The Journey to Business Process Compliance, pages 426–454. IGI Global, 2009. [cited at p. 31, 35, 181]

[55] Jun Han, Yan Jin, Zheng Li, Tan Phan, and Jian Yu. Guiding the Service Composition Process with Temporal Business Rules. In *ICWS*, pages 735–742. IEEE Computer Society, 2007. [cited at p. 29]

[56] Hans-Jörg Happel and Ljiljana Stojanovic. Ontoprocess - a prototype for semantic business process verification using swrl rules. In *3rd European Semantic Web Conference (ESWC2006)*, June 2006. [cited at p. 32]

[57] Thomas E. Hartman. *The Cost of Being Public in the Era of Sarbanes-Oxley*. [Chicago, Ill.] : Foley & Lardner, June 2006. [cited at p. 4]

[58] Jörg Hoffmann, Ingo Weber, and Guido Governatori. On compliance checking for clausal constraints in annotated process models. *Information Systems Frontiers*, Special Issue on Governance, Risk, and Compliance, available online, 2009. [cited at p. 32]

[59] Gerard J. Holzmann. The Model Checker SPIN. *Software Engineering*, 23(5):279–295, 1997. [cited at p. 68]

[60] Chung-Wah Norris Ip. *State Reduction Methods for Automatic Formal Verification*. PhD thesis, Department of Computer Science, Stanford University, 1996. [cited at p. 98]

[61] R. Johnson, D. Pearson, and K. Pingali. The program structure tree: computing control regions in linear time. *SIGPLAN Not.*, 29(6):171–185, 1994. [cited at p. 144]

[62] Vandana Kabilan, Paul Johannesson, and Dickson M. Rugaimukamu. Business Contract Obligation Monitoring through Use of Multi Tier Contract Ontology. In Robert Meersman and Zahir Tari, editors, *OTM Workshops*, volume 2889 of *Lecture Notes in Computer Science*, pages 690–702. Springer, 2003. [cited at p. 36]

[63] Martin Kähmer, Maike Gilliot, and G. Muller. Automating privacy compliance with expdt. In *CEC/EEE*, pages 87–94. IEEE, 2008. [cited at p. 37]

[64] Dimitris Karagiannis. A business process-based modelling extension for regulatory compliance. In Martin Bichler, Thomas Hess, Helmut Krcmar, Ulrike Lechner, Florian Matthes, Arnold Picot, Benjamin Speitkamp, and Petra Wolf, editors, *Multikonferenz Wirtschaftsinformatik*. GITO-Verlag, Berlin, 2008. [cited at p. 19]

[65] Dimitris Karagiannis, John Mylopoulos, and Margit Schwab. Business process-based regulation compliance: The case of the sarbanes-oxley act. In *RE*, pages 315–321. IEEE, 2007. [cited at p. 5, 180]

[66] Matthias Kehlenbeck, Thorben Sandner, and Michael H. Breitner. Application and economic implications of an automated requirement-oriented and standard-based compliance monitoring and reporting prototype. In *ARES*, pages 468–474. IEEE Computer Society, 2010. [cited at p. 180]

[67] G. Keller, M. Nüttgens, and A.W. Scheer. Semantische Prozessmodellierung auf der Grundlage "Ereignisgesteuerter Prozessketten (EPK)". Technical Report 89, Institut für Wirtschaftsinformatik, Saarbrücken, 1992. [cited at p. 41, 42]

[68] Marwane El Kharbili, Ana Karla Alves de Medeiros, Sebastian Stein, and W.M.P. van der Aalst. Business Process Compliance Checking: Current State and Future Challenges. In Peter Loos, Markus Nüttgens, Klaus Turowski, and Dirk Werth (Hrsg.), editors, *Modellierung betrieblicher Informationssysteme (MobIS 2008). Modellierung zwischen SOA und Compliance Management*, volume P-141 of *LNI*, pages 107–113. GI, November 2008. [cited at p. 13]

[69] Marwane El Kharbili and Sebastian Stein. Policy-Based Semantic Compliance Checking for Business Process Management. In Peter Loos, Markus Nüttgens, Klaus Turowski, and Dirk Werth, editors, *Proceedings of Workshops colocated with the MobIS-2008 conference: EPK-2008, KobAS-2008 and ModKollGP-2008*, pages 178–192. CEUR-WS.org, November 2008. [cited at p. 24, 25, 27, 181]

[70] Marwane El Kharbili, Sebastian Stein, Ivan Markovic, and Elke Pulvermüller. Towards a Framework for Semantic Business Process Compliance Management. In Shazia Sadiq, Marta Indulska, Michael zur Muehlen, Xavier Franch, Ela Hunt, and Remi Coletta, editors, *Proceedings of the 1st International Workshop on Governance, Risk and Compliance: Applications in Information Systems (GRCIS'08)*, pages 1–15. CEUR-WS.org, June 2008. [cited at p. 25, 27]

[71] Nadzeya Kiyavitskaya, Nicola Zeni, Travis D. Breaux, Annie I. Antón, James R. Cordy, Luisa Mich, and John Mylopoulos. Automating the Extraction of Rights and Obligations for Regulatory Compliance. In *ER '08: Proceedings of the 27th International Conference*

on Conceptual Modeling, pages 154–168, Berlin, Heidelberg, 2008. Springer-Verlag. [cited at p. 20, 38]

[72] Natallia Kokash and Farhad Arbab. Formal behavioral modeling and compliance analysis for service-oriented systems. In Frank S. de Boer, Marcello M. Bonsangue, and Eric Madelain, editors, *FMCO*, volume 5751 of *Lecture Notes in Computer Science*, pages 21–41. Springer, 2008. [cited at p. 33, 35]

[73] Akhil Kumar and Rong Liu. A Rule-Based Framework Using Role Patterns for Business Process Compliance. In *RuleML*, volume 5321 of *Lecture Notes in Computer Science*, pages 58–72. Springer, 2008. [cited at p. 33, 35]

[74] Jochen Malte Küster, Ksenia Ryndina, and Harald Gall. Generation of Business Process Models for Object Life Cycle Compliance. In Gustavo Alonso, Peter Dadam, and Michael Rosemann, editors, *BPM*, volume 4714 of *Lecture Notes in Computer Science*, pages 165–181. Springer, 2007. [cited at p. 30, 70]

[75] Ralf Laue and Ahmed Awad. Visualization of Business Process Modeling Anti Patterns. In P. Bottoni, E. Guerra, J. de Lara, T. Margaria, J. Padberg, and G. Taentzer, editors, *Proceedings of the 1st International Workshop on Visual Formalisms for Patterns (VFfP 2009), Corvallis, OR (USA) (to be published)*. European Association of Software Science and Technology, 2009. [cited at p. 52]

[76] Ying Liu, Samuel Müller, and Ke Xu. A static compliance-checking framework for business process models. *IBM SYSTEMS JOURNAL*, 46(2):335–362, 2007. [cited at p. 18, 32, 35]

[77] Alessio Lomuscio, Hongyang Qu, and Monika Solanki. Towards Verifying Contract Regulated Service Composition. In *ICWS '08: Proceedings of the 2008 IEEE International Conference on Web Services*, pages 254–261, Washington, DC, USA, 2008. IEEE Computer Society. [cited at p. 31]

[78] Volkmar Lotz, Emmanuel Pigout, Peter M. Fischer, Donald Kossmann, Fabio Massacci, and Alexander Pretschner. Towards systematic achievement of compliance in service-oriented architectures: The master approach. *Wirtschaftsinformatik*, 50(5):383–391, 2008. [cited at p. 37]

[79] R. Lu, Sh. Sadiq, and G. Governatori. Measurement of Compliance Distance in Business Processes. *Inf. Sys. Manag.*, 25(4):344–355, 2008. [cited at p. 31, 35]

[80] Ruopeng Lu, Shazia Wasim Sadiq, and Guido Governatori. Compliance Aware Business Process Design. In *Business Process Management Workshops*, volume 4928 of *Lecture Notes in Computer Science*, pages 120–131. Springer, 2007. [cited at p. 31, 35]

[81] Linh Thao Ly, Kevin Göser, Stefanie Rinderle-Ma, and Peter Dadam. Compliance of Semantic Constraints- A Requirements Analysis for Process Management Systems. In Shazia Sadiq, Marta Indulska, Michael zur Muehlen, Xavier Franch, Ela Hunt, and Remi Coletta, editors, *Proceedings of the 1st International Workshop on Governance, Risk and Compliance: Applications in Information Systems (GRCIS'08)*, pages 16–30. CEUR-WS.org, June 2008. [cited at p. 13, 25, 27]

[82] Linh Thao Ly, Stefanie Rinderle, and Peter Dadam. Semantic Correctness in Adaptive Process Management Systems. In Schahram Dustdar, José Luiz Fiadeiro, and Amit P.

Sheth, editors, *Business Process Management*, volume 4102 of *Lecture Notes in Computer Science*, pages 193–208. Springer, 2006. [cited at p. 32, 35, 60]

[83] Linh Thao Ly, Stefanie Rinderle, and Peter Dadam. Integration and verification of semantic constraints in adaptive process management systems. *Data Knowl. Eng.*, 64(1):3–23, 2008. [cited at p. 32, 35]

[84] Linh Thao Ly, Stefanie Rinderle-Ma, Kevin Göser, and Peter Dadam. On Enabling Integrated Process Compliance with Semantic Constraints in Process Management Systems Requirements, Challenges, Solutions. *Information System Frontiers*, 2009. [cited at p. 25, 27, 28]

[85] Monika Maidl. The common fragment of ctl and ltl. In *FOCS*, pages 643–652, 2000. [cited at p. 64]

[86] Annapaola Marconi, Marco Pistore, and Paolo Traverso. Automated Composition of Web Services: the ASTRO Approach. *IEEE Data Eng. Bull.*, 31(3):23–26, 2008. [cited at p. 146]

[87] Axel Martens. Analyzing web service based business processes. In Maura Cerioli, editor, *FASE*, volume 3442 of *Lecture Notes in Computer Science*, pages 19–33. Springer, 2005. [cited at p. 5]

[88] Kenneth L. McMillan. *Symbolic Model Checking An approach to the state explosion problem*. PhD thesis, Carnegie Mellon University, May 1992. [cited at p. 98]

[89] Jan Mendling, Karsten Ploesser, and Mark Strembeck. Specifying Separation of Duty Constraints in BPEL4People Processes. In Witold Abramowicz and Dieter Fensel, editors, *BIS*, volume 7 of *Lecture Notes in Business Information Processing*, pages 273–284. Springer, 2008. [cited at p. 34]

[90] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, part i/ii. *Information and Computation*, 100:1–77, 1992. [cited at p. 41]

[91] Zoran Milosevic, Audun Jøsang, Theodosis Dimitrakos, and Mary Anne Patton. Discretionary Enforcement of Electronic Contracts. In *EDOC*, pages 39–50. IEEE Computer Society, 2002. [cited at p. 36]

[92] Zoran Milosevic, Shazia Wasim Sadiq, and Maria E. Orlowska. Towards a Methodology for Deriving Contract-Compliant Business Processes. In Schahram Dustdar, José Luiz Fiadeiro, and Amit P. Sheth, editors, *Business Process Management*, volume 4102 of *Lecture Notes in Computer Science*, pages 395–400. Springer, 2006. [cited at p. 29]

[93] Zoran Milosevic, Shazia Wasim Sadiq, and Maria E. Orlowska. Translating business contract into compliant business processes. In *EDOC*, pages 211–220. IEEE Computer Society, 2006. [cited at p. 29]

[94] Kioumars Namiri and Nenad Stojanovic. Pattern-Based Design and Validation of Business Process Compliance. In Robert Meersman and Zahir Tari, editors, *OTM Conferences (1)*, volume 4803 of *Lecture Notes in Computer Science*, pages 59–76. Springer, 2007. [cited at p. 23, 27, 67]

[95] Kioumars Namiri and Nenad Stojanovic. Using control patterns in business processes compliance. In Mathias Weske, Mohand-Said Hacid, and Claude Godart, editors, *WISE Workshops*, volume 4832 of *Lecture Notes in Computer Science*, pages 178–190. Springer, 2007. [cited at p. 23]

[96] Kioumars Namiri and Nenad Stojanovic. Towards a formal framework for business process compliance. In Martin Bichler, Thomas Hess, Helmut Krcmar, Ulrike Lechner, Florian Matthes, Arnold Picot, Benjamin Speitkamp, and Petra Wolf, editors, *Multikonferenz Wirtschaftsinformatik*. GITO-Verlag, Berlin, 2008. [cited at p. 24, 27]

[97] Dana Nau, Malik Ghallab, and Paolo Traverso. *Automated Planning: Theory & Practice*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2004. [cited at p. 19, 144, 146]

[98] OASIS. Universal Description Discovery and Integration UDDI, 2004. [cited at p. 59]

[99] Congress of the United States. *Public Company Accounting Reform and Investor Protection Act (Sarbanes-Oxley Act)*. Public Law 107-204, 116 Stat. 745, 2002. [cited at p. 3, 4, 6, 38]

[100] OMG. Uml 2.0 superstructure specification. Technical report, 2004. [cited at p. 31, 41, 42]

[101] Basel Committee on Banking Supervision. Basel ii accord, 2004. [cited at p. 3]

[102] Martin Ouimet. Formal Software Verification: Model Checking and Theorem Proving. Technical Report ESL-TIK-00214, Massachusetts Institute of Technology, 2008. [cited at p. 18]

[103] Vineet Padmanabhan, Guido Governatori, Shazia Wasim Sadiq, Robert Colomb, and Antonino Rotolo. Process Modelling: The Deontic Way. In Markus Stumptner, Sven Hartmann, and Yasushi Kiyoki, editors, *APCCM*, volume 53 of *CRPIT*, pages 75–84. Australian Computer Society, 2006. [cited at p. 179]

[104] J. Scott Penberthy and Daniel S. Weld. UCPOP: A Sound, Complete, Partial Order Planner for ADL. In *KR*, pages 103–114, 1992. [cited at p. 153]

[105] Maja Pesić. *Constraint-Based Workflow Management System: Shifting Control to Users*. PhD thesis, Technische Universiteit Eindhoven, 2008. [cited at p. 181]

[106] Maja Pesic, Helen Schonenberg, and Wil M. P. van der Aalst. DECLARE: Full Support for Loosely-Structured Processes. In *EDOC*, pages 287–300. IEEE Computer Society, 2007. [cited at p. 41]

[107] Maja Pesic and Wil M. P. van der Aalst. A Declarative Approach for Flexible Business Processes Management. In Johann Eder and Schahram Dustdar, editors, *Business Process Management Workshops*, volume 4103 of *Lecture Notes in Computer Science*, pages 169–180. Springer, 2006. [cited at p. 181]

[108] Aretem Plyvyanyy, ussi Vanhatalo, and Hagen Voelzer. Simplified Computation and Generalization of the Refined Process Structure Tree . Technical Report RZ3745, IBM, 2009. [cited at p. 166]

[109] Amir Pnueli. The temporal logic of programs. In *SFCS '77: Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, pages 46–57, Washington, DC, USA, 1977. IEEE Computer Society. [cited at p. 61]

[110] Artem Polyvyanyy, Sergey Smirnov, and Mathias Weske. The Triconnected Abstraction of Process Models. In Umeshwar Dayal, Johann Eder, Jana Koehler, and Hajo A. Reijers, editors, *BPM*, volume 5701 of *Lecture Notes in Computer Science*, pages 229–244. Springer, 2009. [cited at p. 166]

[111] Frank Puhlmann and Mathias Weske. Investigations on soundness regarding lazy activities. In *BPM 2006*, Lecture Notes in Computer Science 4102, pages 145–160. Springer, 2006. [cited at p. 5]

[112] Manfred Reichert and Peter Dadam. ADEPT$_{\text{flex}}$-Supporting Dynamic Changes of Workflows Without Losing Control. *J. Intell. Inf. Syst.*, 10(2):93–129, 1998. [cited at p. 32]

[113] Wolfgang Reisig. *Petri Nets*. Springer, EATCS Monographs on Theoretical Computer Science edition, 1985. [cited at p. 45, 101]

[114] Stefanie Rinderle, Manfred Reichert, and Peter Dadam. Flexible Support of Team Processes by Adaptive Workflow Systems. *Distributed and Parallel Databases*, 16(1):91–116, 2004. [cited at p. 32]

[115] William N. Robinson. Implementing Rule-Based Monitors within a Framework for Continuous Requirements Monitoring. In *HICSS*. IEEE Computer Society, 2005. [cited at p. 182]

[116] Mohsen Rouached, Olivier Perrin, and Claude Godart. A Contract Layered Architecture for Regulating Cross-Organisational Business Processes. In Wil M. P. van der Aalst, Boualem Benatallah, Fabio Casati, and Francisco Curbera, editors, *Business Process Management*, volume 3649, pages 410–415, 2005. [cited at p. 36]

[117] Steffen Ryll. Querying the Data Perspective of Business Process Models. Master's thesis, Business Process Technology Group, Hasso Plattner Institute, University of Potsdam, Germany, March 2009. [cited at p. 52, 55]

[118] Ksenia Ryndina, Jochen M. Küster, and Harald C. Gall. Consistency of Business Process Models and Object Life Cycles. In *Models in Software Engineering*, number 4364 in Lecture Notes in Computer Science, pages 80–90. Springer, 2007. [cited at p. 75]

[119] Stefan Sackmann, Martin Kähmer, Maike Gilliot, and Lutz Lowis. A classification model for automating compliance. In *CEC/EEE*, pages 79–86. IEEE, 2008. [cited at p. 6, 7]

[120] Shazia Sadiq, Maria Orlowska, Wasim Sadiq, and Cameron Foulger. Data Flow and Validation in Workflow Modeling. In *ADC '04: Proceedings of the 15th Australasian database conference*, pages 207–214, Darlinghurst, Australia, Australia, 2004. Australian Computer Society, Inc. [cited at p. 5, 44, 45]

[121] Shazia Wasim Sadiq, Guido Governatori, and Kioumars Namiri. Modeling Control Objectives for Business Process Compliance. In Gustavo Alonso, Peter Dadam, and Michael Rosemann, editors, *BPM*, volume 4714 of *Lecture Notes in Computer Science*, pages 149–164. Springer, 2007. [cited at p. 11, 23, 27]

[122] Wasim Sadiq and Maria E. Orlowska. Analyzing process models using graph reduction techniques. *Inf. Syst.*, 25(2):117–134, 2000. [cited at p. 44]

[123] Motoshi Saeki and Haruhiko Kaiya. Supporting the Elicitation of Requirements Compliant with Regulations. In *CAiSE '08: Proceedings of the 20th international conference on Advanced Information Systems Engineering*, pages 228–242, Berlin, Heidelberg, 2008. Springer-Verlag. [cited at p. 38]

[124] Sherif Sakr and Ahmed Awad. A Framework for Querying Graph-Based Business Process Models. In *Proceedings of the 19th International World Wide Web Conference (WWW)*, 2010. (To Appear). [cited at p. 18, 170]

[125] Daniel Schleicher, Tobias Anstett, Frank Leymann, and Ralph Mietzner. Maintaining Compliance in Customizable Process Models. In Robert Meersman, Tharam S. Dillon, and Pilar Herrero, editors, *OTM Conferences (1)*, volume 5870 of *Lecture Notes in Computer Science*, pages 60–75. Springer, 2009. [cited at p. 30]

[126] Karsten Schmidt. LoLA: A Low Level Analyser. In *Application and Theory of Petri Nets 2000: 21st International Conference, ICATPN 2000, Aarhus, Denmark, June 2000. Proceedings*, volume 1825, page 465. Springer Berlin / Heidelberg, 2000. [cited at p. 170]

[127] Rainer Schmidt, Christian Bartsch, and Roy Oberhauser. Ontology-based representation of compliance requirements for service processes. In Martin Hepp, Knut Hinkelmann, Dimitris Karagiannis, Rüdiger Klein, and Nenad Stojanovic, editors, *Proceedings of the Workshop on Semantic Business Process and Product Lifecycle Management held in conjunction with the 3rd European Semantic Web Conference (ESWC 2007), Innsbruck, Austria, June 7, 2007*, volume 251 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2007. [cited at p. 24, 27]

[128] Dennis Shasha, Jason Tsong-Li Wang, and Rosalba Giugno. Algorithmics and Applications of Tree and Graph Searching. In *Symposium on Principles of Database Systems*, pages 39–52, 2002. [cited at p. 54]

[129] Sergey Smirnov, Matthias Weidlich, Jan Mendling, and Mathias Weske. Action Patterns in Business Process Models. In Luciano Baresi, Chi-Hung Chi, and Jun Suzuki, editors, *ICSOC/ServiceWave*, volume 5900 of *Lecture Notes in Computer Science*, pages 115–129. Springer, 2009. [cited at p. 181]

[130] Pnina Soffer, Maya Kaner, and Yair Wand. Assigning Ontology-Based Semantics to Process Models: The Case of Petri Nets. In Zohra Bellahsene and Michel Léonard, editors, *CAiSE*, volume 5074 of *Lecture Notes in Computer Science*, pages 16–31. Springer, 2008. [cited at p. 54]

[131] Christian Soltenborn and Gregor Engels. Analysis of uml activities with dynamic meta modeling techniques. In Thomas Kühne, editor, *MoDELS Workshops*, volume 4364 of *Lecture Notes in Computer Science*, pages 329–330. Springer, 2006. [cited at p. 31]

[132] Ulrich Stern and David L. Dill. Parallelizing the murhi verifier. In Orna Grumberg, editor, *CAV*, volume 1254 of *Lecture Notes in Computer Science*, pages 256–278. Springer, 1997. [cited at p. 98]

[133] Sherry X. Sun and J. Leon Zhao. Developing a Workflow Design Framework Based on Dataflow Analysis. In *HICSS '08: Proceedings of the 41st Annual Hawaii International Conference on System Sciences*, page 19, Washington, DC, USA, 2008. IEEE Computer Society. [cited at p. 45]

[134] Sherry X. Sun, J. Leon Zhao, Jay F. Nunamaker, and Olivia R. Liu Sheng. Formulating the Data-Flow Perspective for Business Process Management. *Info. Sys. Research*, 17(4):374–391, 2006. [cited at p. 5, 44]

[135] W. M. P. van der Aalst, B. F. van Dongen, J. Herbst, L. Maruster, G. Schimm, and A. J. M. M. Weijters. Workflow mining: a survey of issues and approaches. *Data Knowl. Eng.*, 47(2):237–267, 2003. [cited at p. 37]

[136] W. M. P. van der Aalst, B. F. van Dongen, J. Herbst, L. Maruster, G. Schimm, and A. J. M. M. Weijters. Workflow mining: a survey of issues and approaches. *Data Knowl. Eng.*, 47(2):237–267, 2003. [cited at p. 180]

[137] Wil M. P. van der Aalst. Formalization and verification of event-driven process chains. *Information & Software Technology*, 41(10):639–650, 1999. [cited at p. 18, 45]

[138] Wil M. P. van der Aalst. Business alignment: using process mining as a tool for Delta analysis and conformance testing. *Requir. Eng.*, 10(3):198–211, 2005. [cited at p. 182]

[139] Wil M. P. van der Aalst and Twan Basten. Inheritance of workflows: an approach to tackling problems related to change. *Theor. Comput. Sci.*, 270(1-2):125–203, 2002. [cited at p. 32]

[140] Wil M. P. van der Aalst, H. T. de Beer, and Boudewijn F. van Dongen. Process Mining and Verification of Properties: An Approach Based on Temporal Logic. In Robert Meersman, Zahir Tari, Mohand-Said Hacid, John Mylopoulos, Barbara Pernici, Özalp Babaoglu, Hans-Arno Jacobsen, Joseph P. Loyall, Michael Kifer, and Stefano Spaccapietra, editors, *OTM Conferences (1)*, volume 3760 of *Lecture Notes in Computer Science*, pages 130–147. Springer, 2005. [cited at p. 37, 182]

[141] Wil M. P. van der Aalst and Ana Karla A. de Medeiros. Process Mining and Security: Detecting Anomalous Process Executions and Checking Process Conformance. *Electr. Notes Theor. Comput. Sci.*, 121:3–21, 2005. [cited at p. 37]

[142] Wil M. P. van der Aalst and Maja Pesic. Decserflow: Towards a truly declarative service flow language. In Frank Leymann, Wolfgang Reisig, Satish R. Thatte, and Wil M. P. van der Aalst, editors, *The Role of Business Processes in Service Oriented Architectures*, volume 06291 of *Dagstuhl Seminar Proceedings*. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), S chloss Dagstuhl, Germany, 2006. [cited at p. 41]

[143] Wil M. P. van der Aalst and Arthur H. M. ter Hofstede. Yawl: yet another workflow language. *Inf. Syst.*, 30(4):245–275, 2005. [cited at p. 45]

[144] J. Vanhatalo, H. Völzer, and F. Leymann. Faster and More Focused Control-Flow Analysis for Business Process Models Through SESE Decomposition. In Bernd J. Krämer, Kwei-Jay Lin, and Priya Narasimhan, editors, *Service-Oriented Computing - ICSOC 2007, Fifth International Conference, Vienna, Austria, September 17-20, 2007, Proceedings*, volume 4749 of *Lecture Notes in Computer Science*, pages 43–55. Springer, 2007. [cited at p. 144]

[145] Jussi Vanhatalo, Hagen Völzer, and Jana Koehler. The Refined Process Structure Tree. In Marlon Dumas, Manfred Reichert, and Ming-Chien Shan, editors, *BPM*, volume 5240 of *Lecture Notes in Computer Science*, pages 100–115. Springer, 2008. [cited at p. 144]

[146] Moshe Y. Vardi. Branching vs. linear time: Final showdown. In Tiziana Margaria and Wang Yi, editors, *TACAS*, volume 2031 of *Lecture Notes in Computer Science*, pages 1–22. Springer, 2001. [cited at p. 64]

[147] Barbara Weber and Manfred Reichert. Refactoring Process Models in Large Process Repositories. In *CAiSE*, volume 5074 of *Lecture Notes in Computer Science*, pages 124–139. Springer, 2008. [cited at p. 167]

[148] Ingo Weber, Guido Governatori, and Jörg Hoffmann. Approximate Compliance Checking for Annotated Process Models. In *Proceedings of the first international workshop of governance, risk and compliance in information systems GRCIS*, volume 339, pages 46–60. CEUR-Workshop, 2008. [cited at p. 32, 33, 35]

[149] Hans Weigand and Willem-Jan van den Heuvel. Cross-organizational Workflow Integration Using Contracts. *Decision Support Systems*, 33(3):247–265, 2002. [cited at p. 36]

[150] Mathias Weske. Formal Foundation and Conceptual Design of Dynamic Adaptations in a Workflow Management System. In *HICSS '01: Proceedings of the 34th Annual Hawaii International Conference on System Sciences (HICSS-34)-Volume 7*, page 7051, Washington, DC, USA, 2001. IEEE Computer Society. [cited at p. 32]

[151] Mathias Weske. *Business Process Management*. Springer, 2007. [cited at p. 59]

[152] Christian Wolter, Philip Miseldine, and Christoph Meinel. Verification of Business Process Entailment Constraints Using SPIN. In Fabio Massacci, Samuel T. Redwine Jr., and Nicola Zannone, editors, *ESSoS*, volume 5429 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 2009. [cited at p. 12, 34, 35]

[153] Christian Wolter and Andreas Schaad. Modeling of Task-Based Authorization Constraints in BPMN. In Gustavo Alonso, Peter Dadam, and Michael Rosemann, editors, *BPM*, volume 4714 of *Lecture Notes in Computer Science*, pages 64–79. Springer, 2007. [cited at p. 12, 34, 35]

[154] Bwolen Yang, Randal E. Bryant, David R. O'Hallaron, Armin Biere, Olivier Coudert, Geert Janssen, Rajeev K. Ranjan, and Fabio Somenzi. A performance study of BDD-based model checking. In *Formal Methods in Computer-Aided Design*, pages 255–289, 1998. [cited at p. 67]

[155] Frederick Yip, Nandan Parameswaran, and Pradeep Ray. Rules and ontology in compliance management. In *EDOC*, pages 435–442. IEEE Computer Society, 2007. [cited at p. 24, 27]

[156] Jian Yu, Jun Han, Paolo Falcarin, and Maurizio Morisio. Using Temporal Business Rules to Synthesize Service Composition Process Models. In Marten van Sinderen, editor, *ACT4SOC*, pages 85–94. INSTICC Press, 2007. [cited at p. 30]

[157] Jian Yu, Yanbo Han, Jun Han, Yan Jin, Paolo Falcarin, and Maurizio Morisio. Synthesizing Service Composition Models on the Basis of Temporal Business Rules. *J. Comput. Sci. Technol.*, 23(6):885–894, 2008. [cited at p. 30]

[158] Jian Yu, Tan Phan Manh, Jun Han, Yan Jin, Yanbo Han, and Jianwu Wang. Pattern Based Property Specification and Verification for Service Composition. In Karl Aberer, Zhiyong Peng, Elke A. Rundensteiner, Yanchun Zhang, and Xuhui Li, editors, *WISE*, volume 4255 of *Lecture Notes in Computer Science*, pages 156–168. Springer, 2006. [cited at p. 32, 35]

[159] Martijn Zoet, Richard J. Welke, Johan Versendaal, and Pascal Ravesteyn. Aligning risk management and compliance considerations with business process development. In Tommaso Di Noia and Francesco Buccafurri, editors, *EC-Web*, volume 5692 of *Lecture Notes in Computer Science*, pages 157–168. Springer, 2009. [cited at p. 8, 11]

[160] L. Zuck. *Past Temporal Logic*. PhD thesis, Weizmann Intitute, Rehovet, Israel, August 1986. [cited at p. 61, 67]

[161] Michael zur Muehlen and Danny Ting-Yi Ho. Risk Management in the BPM Lifecycle. In Christoph Bussler and Armin Haller, editors, *Business Process Management Workshops*, volume 3812, pages 454–466, 2005. [cited at p. 23]

[162] Michael zur Muehlen, Marta Indulska, and Gerrit Kamp. Business Process and Business Rule Modeling Languages for Compliance Management: A Representational Analysis. In John C. Grundy, Sven Hartmann, Alberto H. F. Laender, Leszek A. Maciaszek, and John F. Roddick, editors, *ER (Tutorials, Posters, Panels & Industrial Contributions)*, volume 83 of *CRPIT*, pages 127–132. Australian Computer Society, 2007. [cited at p. 38]