ulm university universität uulm

Fakultät für Ingenieurwissenschaften und Informatik
Institut für Datenbanken
und Informationssysteme
Leiter: Prof. Dr. Peter Dadam

SeaFlows – A Compliance Checking Framework for Supporting the Process Lifecycle

Dissertation zur Erlangung des Doktorgrades Dr.rer.nat. der Fakultät
für Ingenieurwissenschaften und Informatik der Universität Ulm

Vorgelegt von:
Linh Thao Ly aus Hanoi, Vietnam

November 2012

Amtierender Dekan:	Prof. Dr. Klaus Dietmayer
Gutachter:	Prof. Dr. Stefanie Rinderle-Ma
	Prof. Dr. Friedrich von Henke
	Prof. Dr. Mathias Weske
Tag der Promotion:	25.04.2013

So viele Menschen haben dazu beigetragen, dass diese Arbeit nun in dieser Form vorliegt. Dafür möchte ich ihnen danken. Meiner Doktormutter, Prof. Dr. Stefanie Rinderle-Ma, danke ich für die umfassende und engagierte Betreuung meiner Arbeit. Unsere Besprechungen und Diskussionen haben mich stets weitergebracht und motiviert. Die Zeit, die ich mit ihrer Forschungsgruppe an der Universität Wien verbringen durfte, gehört zu meinen positivsten Erfahrungen im Kontext dieser Arbeit. Prof. Dr. Peter Dadam danke ich für seine fortwährende Unterstützung während meiner gesamten Arbeit und seine stets offene Tür, wenn ich Besprechungsbedarf hatte. Wenn ich an praktikable Lösungskonzepte denke, werde ich immer seine Metapher der Mrs.-Software-Haus im Hinterkopf haben. Bei Prof. Dr. Friedrich von Henke bedanke ich mich für die vielen erhellenden Besprechungen, aus denen ich stets motivierter als zuvor herausging, und sein Interesse an meiner Arbeit und meinem Fortschritt. Prof. Dr. Manfred Reichert danke ich für seine tatkräftige und konstruktive Unterstützung.

Meinen Kollegen vom Institut für Datenbanken und Informationssysteme der Universität Ulm, die mir Vorreiter, Weggefährten und/oder Leidensgenossen waren, danke ich für das freundschaftliche Arbeitsklima. Besonders danke ich David Knuplesch für die aufschlussreichen wie unterhaltsamen Whiteboard-Diskussionen und Julian Tiedeken für die äußerst angenehme Büronachbarschaft. Auch den Kollegen der Forschungsgruppe Workflow Systems and Technology der Universität Wien danke ich für die freundliche Aufnahme. Jürgen Mangler und Conrad Indiono danke ich besonders für unsere schöne Zusammenarbeit und die lustigen abendlichen Skype-Meetings.

Prof. Dr. Mathias Weske danke ich für seine Bereitschaft, meine Arbeit zu begutachten, und die Gelegenheit, seine Arbeitsgruppe zu besuchen. Prof. Dr. Jacobo Torán danke ich dafür, dass er sich meiner Amateur-Fragen zur Model-Theorie annahm. Dr. Holger Pfeifer danke ich für die fruchtbare Zusammenarbeit zum Thema Model Checking. Frank Schmid danke ich für die Unterstützung bei der Evaluation meiner Konzepte.

Ohne die fähigen studentischen Hilfskräfte, Praktikanten und Diplomanden hätte es die prototypische Implementierung, die in dieser Arbeit vorgestellt wird, nicht in dieser Form gegeben. Dafür danke ich besonders Philipp Merkel, David Knuplesch, Simon Straub und Thomas Spindler.

Meine Familie weiß, wie dankbar ich ihr bin.

Zusammenfassung Stetige wachsende gesetzliche Regularien aber auch zunehmende interne Qualitätsanforderungen machen Compliance für heutige Unternehmen zu einer echten Überlebensfrage. Compliance betrifft immer mehr die heutzutage stark arbeitsteilig ausgerichteten Geschäftsprozesse. Beziehen sich Regularien auf Geschäftsprozesse, wird von Business Process Compliance gesprochen. Ein Schlüssel zur effektiver Unterstützung von Business Process Compliance ist die Automatisierung mittels IT. Da Prozess-Management-Systeme verschiedene Sichten eines Geschäftsprozesses vereinigen, stellen sie eine ideale Umgebung für die Integration automatisierter Compliance-Kontrollen dar. In der vorliegenden Arbeit wird ein Rahmenwerk zur durchgängigen Unterstützung von Business Process Compliance vorgestellt. Ziel ist es, Prozess-Management-Systeme um die Fähigkeit zu erweitern, automatisierte Compliance-Prüfungen von Prozessmodellen und laufenden Prozessinstanzen vorzunehmen. Entscheidend für die praktische Anwendbarkeit eines solchen Frameworks wird nicht zuletzt dessen Fähigkeit sein, umfassende und verständliche Prüf-Diagnosen zu erzeugen.

Basierend auf einer sorgfältigen Anforderungs- und Literaturanalyse wurde im Rahmen des teilweise von der DFG geförderten Projekts *SeaFlows* (Förderungsnummer RI 1882/1-1) ein Framework zur automatisierten Compliance-Prüfung erarbeitet. Im Detail wird in der vorliegenden Arbeit eine graphische Sprache zur deklarativen Formulierung von Compliance-Regeln vorgestellt. Basierend auf Modellierungsprimitiven, die sich der Notation von Knoten und Kanten bedienen, können komplexe Compliance-Regeln, sogenannte Compliance Rule Graphs (CRGs), formuliert werden. Um automatisierte Compliance-Prüfungen zu ermöglichen, werden CRGs operationalisiert. Hierzu wird die Graphstruktur von CRGs ausgenutzt, um mögliche Zustände von CRGs derart zu repräsentieren, dass sie leicht interpretiert werden können. Dazu werden Ausführungszustände eingeführt, um CRGs adäquat zu markieren. Mittels spezieller Regeln, die die Zustandsmarkierungen in geeigneter Weise ändern, kann der Compliance-Zustand für beobachtete Prozessereignisse inkrementell aktualisiert werden. Dies ermöglicht die Anwendung zur Exploration von Prozessmodellen und zur Laufzeitüberwachung von Prozessinstanzen.

Der vorliegende Ansatz unterstützt Prüfungen zur Modellier- und zur Laufzeit mit denselben Mechanismen. Transformationen modellierter Regeln in unterschiedliche Repräsentationen für Modellprüfungen oder für Laufzeitüberwachung werden überflüssig. Auf Prozessmodellebene deckt die Compliance-Prüfung mittels des vorgestellten Frameworks die vom Prozessmodell ermöglichten Compliance-Zustände auf. Zur Laufzeit können Ausführunsgereignisse von Prozessinstanzen beobachtet werden, um den effektiven Compliance-Zustand zu überwachen. Dabei kann auch das im zugrundeliegenden Prozessmodell definierte zukünftige Verhalten berücksichtigt werden. Aufgrund der Kodierung der Zustände direkt auf Grundlage der CRGs lassen sich diese jederzeit leicht interpretieren. Dies ermöglicht insbesondere die Generierung verständlicher Erklärungen im Falle von Compliance-Verletzungen, die bei der Lokalisierung der Quellen von Compliance-Verletzungen und deren Behebung entscheidend beitragen können. Wenn Compliance-Verletzung noch nicht manifest sind, können aus den Zustandsmarkierungen leicht Aktivitäten abgeleitet werden, die zur Regelerfüllung erforderlich sind. Dies kann für präventive Maßnahmen ausgenutzt werden. Darüber hinaus können nicht nur Aussagen zur allgemeinen Einhaltung von Regeln, sondern auch Diagnosen für individuelle Regelaktivierungen im Prozess getroffen werden. Insgesamt adressiert das vorgestellte Rahmenwerk von der Modellierung prozessbezogener Compliance-Anforderungen bis hin zu automatisierten Compliance-Prüfungen von modellierten, laufenden und vergangenen Prozessen den gesamten Prozesslebenszyklus.

Abstract Compliance-awareness is undoubtedly of utmost importance for companies nowadays. Even though an automated approach to compliance checking and enforcement has been advocated in recent literature as a means to tame the high costs for compliance-awareness, the potential of automated mechanisms for supporting business process compliance is not yet depleted. Business process compliance deals with the question whether business processes are designed and executed in harmony with imposed regulations. In this thesis, we propose a compliance checking framework for automating business process compliance verification within process management systems (PrMSs). Such process-aware information systems constitute an ideal environment for the systematic integration of automated business process compliance checking since they bring together different perspectives on a business process and provide access to process data. The objective of this thesis is to devise a framework that enhances PrMSs with compliance checking functionality. As PrMSs enable both the design and the execution of business processes, the designated compliance checking framework must accommodate mechanisms to support these different phases of the process lifecycle.

A compliance checking framework essentially consists of two major building blocks: a compliance rule language to capture compliance requirements in a checkable manner and compliance checking mechanisms for verification of process models and process instances. Key to the practical application of a compliance checking framework will be its ability to provide comprehensive and meaningful compliance diagnoses. Based on the requirements analysis and meta-analyses, we developed the SeaFlows compliance checking framework proposed in this thesis. We introduce the compliance rule graph (CRG) language for modeling declarative compliance rules. The language provides modeling primitives with a notation based on nodes and edges. A compliance rule is modeled by defining a pattern of activity executions activating a compliance rule and consequences that have to apply once a rule becomes activated. In order to enable compliance verification of process models and process instances, the CRG language is operationalized. Key to this approach is the exploitation of the graph structure of CRGs for representing compliance states of the respective CRGs in a transparent and interpretable manner. For that purpose, we introduce execution states to mark CRG nodes in order to indicate which parts of the CRG patterns can be observed in a process execution. By providing rules to alter the markings when a new event is processed, we enable to update the compliance state for each observed event.

The beauty of our approach is that both design and runtime can be supported using the same mechanisms. Thus, no transformation of compliance rules in different representations for process model verification or for compliance monitoring becomes necessary. At design time, the proposed approach can be applied to explore a process model and to detect which compliance states with respect to imposed CRGs a process model is able to yield. At runtime, the effective compliance state of process instances can be monitored taking also the future predefined in the underlying process model into account. As compliance states are encoded based on the CRG structure, fine-grained and intelligible compliance diagnoses can be derived in each detected compliance state. Specifically, it becomes possible to provide feedback not only on the general enforcement of a compliance rule but also at the level of particular activations of the rule contained in a process. In case of compliance violations, this can explain and pinpoint the source of violations in a process. In addition, measures to satisfy a compliance rule can be easily derived that can be seized for providing proactive support to comply. Altogether, the SeaFlows compliance checking framework proposed in this thesis can be embedded into an overall integrated compliance management framework.

Contents

1 Introduction

Business process compliance refers to the design and execution of business processes in harmony with laws and regulations [RMLD08, SGN07, MRM11]. Even though business process compliance has just become a focus in the business process management (BPM) research community in the last years, the variety of compliance regulations indicates that compliance is not a completely new topic. Compliance requirements can be found in various domains. The financial sector, for example, is considered the most heavily regulated industry [ASI10]. Consider, for example, regulatory packages such as the Sarbanes-Oxley Act (SOX) or BASEL III that have been introduced to strengthen customers' confidence in bank processes. In healthcare, medical guidelines and clinical pathways describe desired treatment procedures that should be followed when treating patients [LR07, RMLD08]. Finally, collections of quality controls and standards, e.g., Six Sigma or ITIL, are of particular importance to many business processes. Further compliance requirements with particular relevance to Germany and the European Union such as the Telemediengesetz are summarized in [ALS11].

Compliance-awareness has become a major issue for organizations nowadays. Even though compliance-awareness still comes at high costs and is predominantly viewed as a burden, non-compliance is often no option [SGN07]. Not complying can not only cause damage to an organization's reputation and, thus, harm the business success but can also lead to severe penalties and even legal actions [ASI10, KSMP08a]. Clearly, legal requirements are a major driver of compliance efforts taken by organizations. However, compliance efforts may be motivated by not only external laws and regulations but also by internal reasons. For example, organizations may install guidelines and business rules as a means of quality assurance for their processes [GOR11].

Except for domain-specific guidelines or best practices that often specify constraints at the operational level, the bulk of existing compliance requirements is rather of informal and abstract nature [SKGL08, KSMP08a, SGN07]. Thus, their implementation requires the interpretation of the requirements and their application to the particular business process and business IT through adequate compliance controls. Traditional audits for after-the-fact detection are often conducted through manual checks by expensive consultants [LSG07]. Hence, the potential of automated compliance controls that can help to enforce and maintain compliance should be seized. However, hard-coded solutions that necessitate costly adaptations in case of changed requirements lack sustainability and, thus, can become a source of high costs [SGN07, KSMP08a]. A recent study [ASI10] indicates a clear need for "affordable IT and IS (information system) tools that facilitate compliance management self-audits and compliance monitoring activities in general".

Process management systems (PrMSs) provide the means to design, implement, deploy, and execute business processes and to monitor and control their execution. As PrMSs integrate different perspectives of a business process, such as the data perspective, the organizational perspective, and the actual workflow, they constitute a suitable environment to hook in and integrate compliance controls. An application-neutral framework for developing and integrating automated compliance controls with PrMSs would be a useful complement that occupies the niche for tools facilitating compliance monitoring activities as identified by Abdullah et al. [ASI10] in the context of PrMSs. Against this background, this chapter sets out the goals and solutions of this thesis. It specifically describes the objectives, the assumptions made with respect to the considered kind of compliance rules, and the structure of this thesis. The chapter is structured as follows. Section 1.1 discusses how compliance controls can be implemented in PrMSs and highlights the kinds of compliance rules that we address in this thesis. The major objectives of our work are described in Chapter 1.2. Chapter 1.3 describes how this thesis is organized.

1.1. Compliance controls in process management systems

Fig. 1.1 illustrates the general approach to the derivation of compliance controls and measures from compliance requirements. It employs the terminology utilized in literature [SKGL08, NS08]. Due to their rather abstract and implementation-independent nature, compliance requirements (for example, stemming from laws or regulatory packages) first have to be interpreted in order to derive concrete *control objectives* tailored to the specific business process [SKGL08, NS08, KSMP08a]. This is typically a manual task as it requires expertise in the particular domain and sound knowledge of the respective business process as well as of the regulations. The control objectives derived are still informal and may be used for documentation and communication purposes. In addition, control objectives are the basis for the implementation of *compliance controls and measures.* The usage of business rules, access control policies, or particular procedures for process automation can be considered examples of concrete control measures [GLM$^+$05]. Though not suitable for automation, manual audits also constitute compliance controls that can be utilized to assess or enforce the achievement of compliance objectives. The particular choice of suitable compliance controls and measures obviously depends largely on the particular business process and its implementation (e.g., level of automation as well as availability and suitability of data for automated controls).

In literature, several frameworks were proposed that aim at providing support for the compliance management lifecycle [The11, KSMP08a, GLM$^+$05]. These frameworks aim at supporting the systematic integration and management of compliance concerns within an organization. Due to

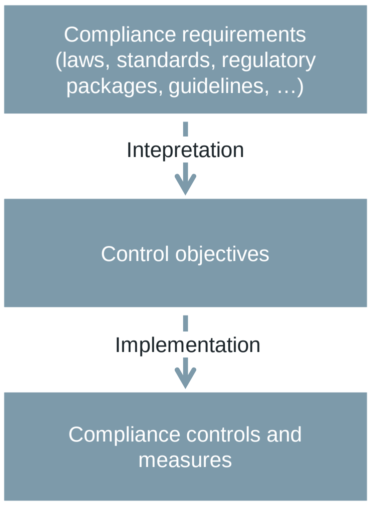

Compliance requirements (laws, standards, regulatory packages, guidelines, …)
Intepretation
Control objectives
Implementation
Compliance controls and measures

Figure 1.1.: General approach to the derivation of compliance controls and measures from abstract compliance requirements

their emphasis on systematic compliance management, we refer to such systems as *compliance management frameworks*. Beyond the management and documentation of compliance concerns as also enabled in commercial governance, risks and compliance tools, such as ARIS Risk & Compliance Manager, these frameworks also envisage support for the process of deriving concrete controls from compliance requirements for target systems, such as access rules [GMP06] or monitoring rules [HMZD11]. Thus, a compliance management framework can accommodate multiple concrete options to create or generate compliance controls to support a variety of target systems. In this context, the compliance management framework ensures the preservation of traceability to the relevant compliance requirements.

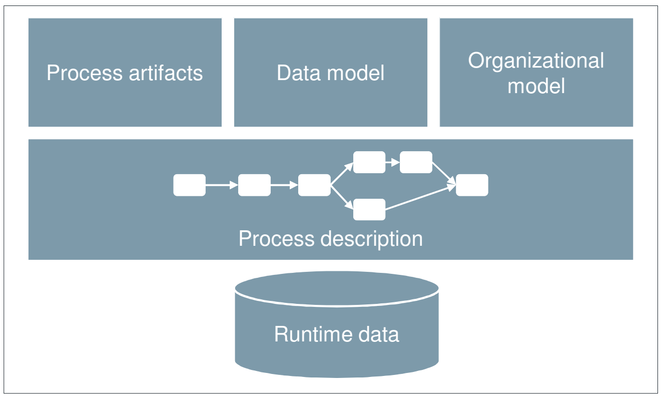


Figure 1.2.: Common perspectives of a business process captured by a PrMS

A *process management system* (PrMS) is a general environment for the design, implementation, and execution of business processes and, thus, constitutes a suitable target system for compliance controls. It accommodates and integrates different perspectives of a business process. The most common perspectives are illustrated in Fig. 1.2. The data perspective describes the data processed by a business process. The organization perspective captures the organizational structure such as organizational positions or roles. The process artifacts perspective comprises process artifacts such as process activities and services. In the process flow perspective, the process description (often captured by a process graph) defines the actual *workflow*, i.e., which activities are executed by whom, and can be considered the linkage of different perspectives. The process description is referred to as *process model* in the following. Example 1.1 describes the derivation of concrete compliance controls in the context of a PrMS from a control objective.

Example 1.1 (From regulations to compliance controls):
This example is adopted from the COMPAS project [The10a, The10c]. According to Section 404 of the Sarbanes-Oxley Act, organizations are supposed to establish an effective system of internal controls to manage risks [AJKL06]. Applied to loan application processes in the banking industries, a common risk is that loans are granted to customers without sufficient credit worthiness. This may not only result in financial deficits but may also damage an organization's reputation. To devise controls for managing that risk, an adequate compliance objective could be "High loan requests have to be processed by supervisors". The objective is still informal and does not yet imply a particular implementation.

For its specific loan application process, a banking organization will have to decide on suitable measures or controls to achieve or monitor the objective. Interpretation of the objective with regard to the loan application process may result in the following informal constraint on the process space: "If the loan request exceeds €1 million, the supervisor will check the credit worthiness of the customer, otherwise the post processing clerk will do the check.". This constraint involves the process data, the process flows as well as organizational aspects. A suitable measure to enforce compliance with the constraint is, for example, to design the implemented loan application process accordingly. This can be achieved by, for example, verifying that the model of the loan application process enforces the constraint. Moreover, running instances of the loan application process can be checked against the constraint to detect violations. Specifically, if it becomes evident that the supervisor does not check the credit worthiness in a process instance associated with a loan request exceeding €1 million, the constraint is obviously violated in this process instance. Finally, we can analyze terminated instances of the loan application process for after-the-fact detection constraint violations.

While multiple further refinement steps may become necessary in order to derive concrete compliance controls from the above-mentioned compliance objective, this example nevertheless illustrates very well the process depicted in Fig. 1.1.

As illustrated in Example 1.1, compliance requirements and corresponding control objectives typically impose *constraints* on the process space that concern the different perspectives captured by a PrMS [MRM11, RMM11]. In the context of PrMS, the development of automated compliance controls comes down to devising measures and controls to check and / or enforce compliance constraints as exemplified in Example 1.1. Checking constraints as a means to verify compliance is also commonly referred to as *compliance checking* or *compliance verification* in literature. Clearly, automated compliance controls within PrMS can only build upon the aspects captured in the process space. Concrete constraints from a purchase-to-delivery scenario involving different perspectives of a business process are provided in Example 1.2. Constraints that exclusively refer to process data can be checked in a straightforward manner by adopting techniques known from information systems research. For example, by applying database technology [AJKL06] such as triggers, certain data conditions (e.g., “Every invoice must have at least one item.”) can be enforced at the database level. Moreover, a multitude of commercial tools such as ARIS Process Performance Manager (ARIS PPM) are available that allow for defining conditions and indicators (e.g., key performance indicators) on process data and for monitoring and reporting on them. Constraints on solely organizational aspects (cf. Example 1.2) can, for example, be enforced or monitored by applying suitable authorization rules [RMR09, MRM11, RMM11]. Constraints involving the process flows perspective, i.e., that directly constrain the process execution, are more interesting and challenging as their checking and enforcement can seize the process-awareness of PrMSs. Their importance to business process compliance is attested by a multitude of related work (e.g., [ADW08, ASW09, LMX07, YMHJ06, GK07, MMC+11, MMWA11, The09b, The10c]).

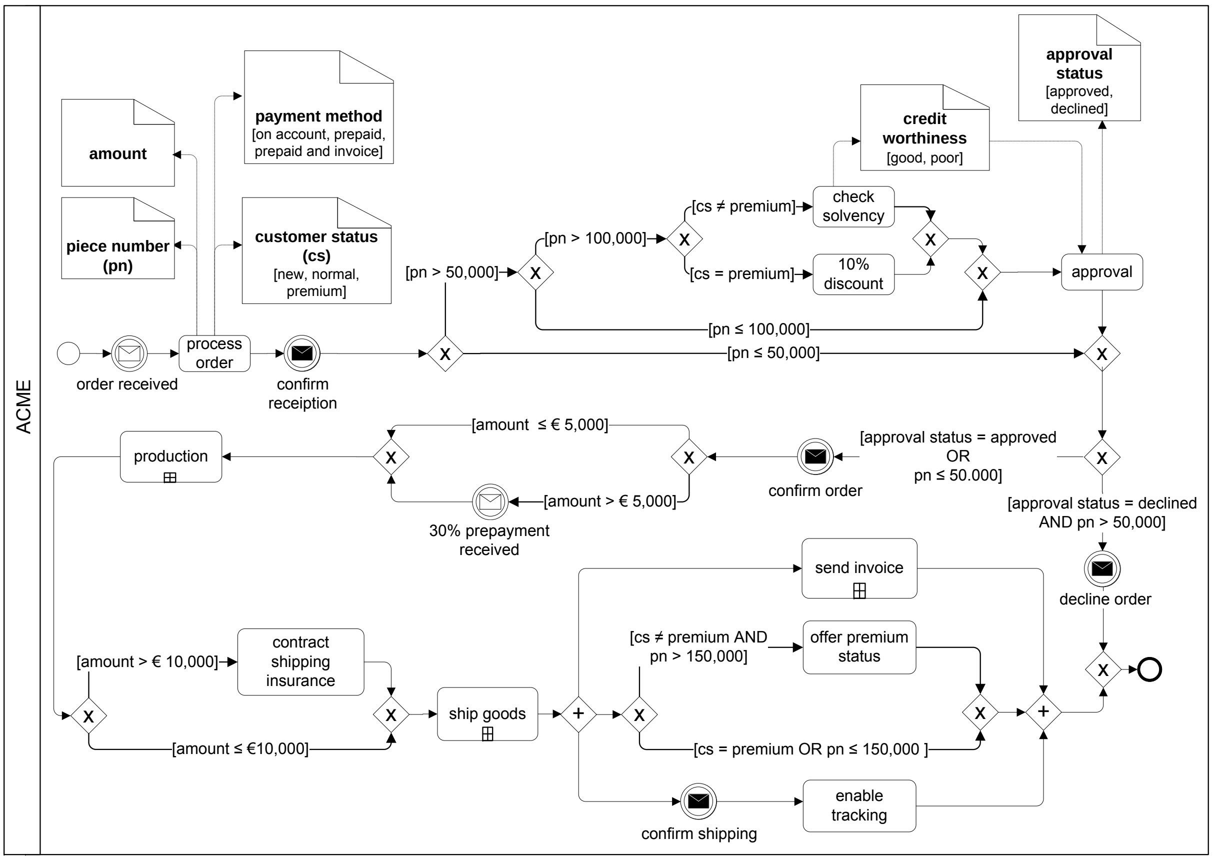


Figure 1.3.: Process model describing a simple order-to-delivery process in BPMN [OMG11]

Example 1.2 (Compliance constraints):
Consider the process model depicted in Fig. 1.3 describing a simple order-to-delivery process in BPMN. Below, constraints on the different aspects of an order-to-delivery process like the one depicted in Fig. 1.3 are given:

On process data

c_1 The payment method for purchase orders of non-premium customers with total amount exceeding €5,000 must not be *payment on account.*

On organizational aspects

c_2 For each department of the organizational model, the role *head of department* must be assigned to a staff member of the department.

On process flows

c_4 Each order shall either be confirmed or declined.

c_5 Goods should be shipped within two weeks after confirming the order or after receiving the prepayment.

c_6 Premium customer status shall only be offered after a prior solvency check.

On a combination of aspects

c_7 Orders with a piece number beyond 50,000 shall be approved by the head of sales before they are confirmed.

c_8 For orders of a non-premium customer with a piece number beyond 80,000, a solvency check is necessary before assessing the order.

c_9 Orders with total amount beyond €10,000 require contracting shipping insurance before shipping.

Note that these constraints can be imposed on other aspects in different process implementation settings. However, against the background of a typical process implementation within `PrMS`s, this classification of the constraints is comprehensible. Other constraints may refer to process execution properties that are reflected in the process' runtime performance data such as quality of services (QoS) requirements [THO+10]. Furthermore, separation-of-duty (SoD) and binding-of-duty (BoD) constraints can be considered further constraint examples whose implementation may affect different aspects of a business process.

In this thesis, we focus on constraints that impose conditions on the process flows, particularly on the occurrence and temporal ordering of activity executions, and thus define conditions that compliant process executions must satisfy. For example, constraints such as c_4 - c_6 in Example 1.2 constitute normative statements on how a process execution is supposed to be like (i.e., what must or must not be done in a process execution and in which order). Note that these constraints may also involve data and organizational aspects as context of activity executions in the process flows (e.g., c_7 - c_9 in Example 1.2). Adopting the terminology from literature, we refer to such constraints as *compliance rules* in the remainder of this thesis. Compliance rules imposed on process executions (referred to as *process instances* in the following), in turn, also impose certain conditions on the process model (typically in the form of a process graph), which defines possible concrete process instances (cf. Fig. 1.4). Thus, a process model bears a violation of an imposed compliance rule if it enables the execution of at least one process instance that violates the compliance rule.

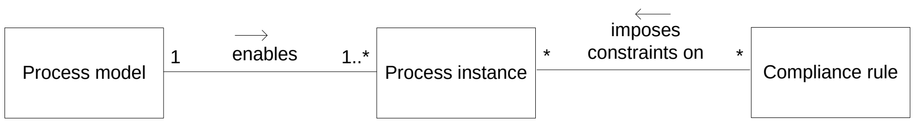


Figure 1.4.: Relations between process models, process instances, and compliance rules

1.2. Objectives

In order to foster compliance automation, it is desirable for PrMSs to provide mechanisms to enforce or at least check the implemented business processes for compliance with imposed compliance rules. Adequate mechanisms to answer the question whether a business process is designed and executed in harmony with imposed compliance rules would constitute an appealing complement to the current functionality of PrMSs. Instead of hard-coded solutions that can become a source of high costs for compliance-aware organizations [KSMP08a], we envisage an application-neutral framework that hooks into existing PrMSs and enables the requisite checks. The objective of this thesis is to devise such a framework that equips PrMSs with compliance checking functionality. As PrMSs enable both the design and the execution of business processes, the designated compliance checking framework must accommodate mechanisms to support these different phases of the *process lifecycle*. In order to compass a compliance checking framework for PrMSs as described, two major working packages have to be tackled:

Working package 1: How to model compliance rules in an adequate manner?

Working package 2: How to automatically verify compliance at process design and process runtime?

In the SeaFlows[1] project, partially funded by the German Research Foundation (DFG)[2], we addressed these working packages. In this thesis, we introduce concepts for modeling compliance rules such that they can be utilized to constrain and verify compliance of business process models and process instances. The concepts proposed in this thesis serve as building blocks for a compliance checking framework that interoperates with PrMSs and extends the latter with compliance checking functionalities at both process design and process runtime.

1.3. Structure of the thesis

This thesis is structured as follows. Chapter 2 discusses requirements on and our vision of a compliance checking framework and describes the research methodology employed in our work. The state-of-the-art is discussed with respect to the requirements in Chapter 3. Chapter 4 gives

[1]Webpage: http://www.seaflows.de
[2]Funding reference: RI 1882/1-1

a short introduction to the different components of the SeaFlows compliance checking framework proposed in this thesis before they are described in detail in the subsequent chapters. Chapter 5 describes the necessary fundamentals of the proposed compliance checking framework. It specifically introduces notions and formal backgrounds that serve as footing for the framework. Chapter 6 introduces the *compliance rule graph* (CRG) language, our approach for modeling compliance requirements. Using the CRG language, compliance requirements can be expressed such that they can be verified automatically at both process design and process runtime. The concepts for conducting these compliance checks are provided in Chapter 7 where we operationalize the CRG language. This enables the execution of modeled CRGs, which provides the basis for compliance checks. Chapter 8 describes how a compliance checking framework as envisioned in Chapter 2 can be realized based on the introduced concepts. Chapter 9 complements the proposed compliance checking framework by addressing issues evolving around the application of the proposed framework in practice. Amongst other issues, Chapter 9 specifically addresses the reduction of the complexity of compliance verification. Our prototypical implementation, referred to as SeaFlows Toolset, is presented in Chapter 10. Our efforts to evaluate the concepts proposed in this thesis are described in Chapter 11. Chapter 12 summarizes the thesis and provides an outlook on future research challenges.

2 Requirements analysis, vision, and research methodology

A viable way to tame the expenses for compliance-awareness is to increase the degree of automation in business process compliance activities [SGN07] and the utilization of tools that facilitate compliance monitoring activities [ASI10]. Adequate mechanisms are necessary in order to seize the potential of automatic compliance verification. As stated in Section 1.2, the objective of our work is to compass a domain-independent compliance checking framework that equips process management systems (PrMSs) with automated compliance checking functionality. In particular, it should become possible to capture process-related compliance rules such that process models and process instances can be automatically verified against them. This chapter discusses requirements on such a compliance checking framework and is based on meta-analyses of related literature and on our previous work [RMLD08, LGRMD08, LRMD10, LRMKD11, LRMGD12]. It specifically compiles requirements on a compliance checking framework for PrMSs and describes our vision of such a framework and the research methodology employed.

This chapter is structured as follows. Requirements on the compliance checking framework are discussed in Section 2.1. Section 2.2 summarizes the requirements, describes the research methodology, and sketches our vision of a compliance checking framework.

2.1. Requirements on a compliance checking framework

Clearly, a framework enabling PrMSs to support compliance checks against imposed compliance rules must provide the means to first integrate such compliance rules with the business processes implemented in the PrMSs. Apparently, for each compliance rule, specific hard-coded checks can be implemented. However, such hard-coded checks are unfavorable as they lack sustainability [SGN07]. In particular, they will have to be revised for each emerging or revised compliance rule, for example, as a result of new regulations. This can lead to high costs for compliance-awareness. Hence, a more generic solution to the integration of compliance rules and their verification is desirable. We envisage that a compliance-aware PrMS enables the modeling of compliance rules in an easy manner and allows for assigning them to business processes. Based on the modeled compliance rules, the PrMS "generates" the requisite checks without further implementation becoming necessary. In order to enable its smooth integration into an existing PrMS such as ADEPT [Rei00, DRR+08], the designated compliance checking framework has to support the process lifecycle (cf. Section 1.2). A framework as described would be a valuable tool for dealing with compliance requirements that can be covered by compliance rules as described in Section 1.1. It can further serve as a building block of an overall *compliance management framework* where it addresses PrMS as target systems. In the following, we discuss the requirements on a compliance checking framework with respect to compliance rule specification in Section 2.1.1. The requirements with respect to support along the process lifecycle are detailed in Section 2.1.2.

2.1.1. Compliance rules

Several aspects have to be considered in order to devise a suitable approach for modeling and integrating automatically checkable compliance rules. Expressiveness and formalization requirements are described in Section 2.1.1.1 and 2.1.1.2, respectively. Ease of use requirements are detailed in Section 2.1.1.3. Section 2.1.1.4 addresses issues concerning the management of compliance rules.

2.1.1.1. Expressiveness

As previously stated in Section 1.1, we focus on compliance rules imposing constraints on the occurrence, absence, and temporal ordering of activity executions within a process execution. In retrospect, each process execution (referred to as process instance) can be reflected by a *sequence of events* associated with the activities carried out in the process. Therefore, more abstractly, we address occurrence and ordering constraints on sequences of events. Conceivably, the designated framework has to provide a compliance rule modeling language exhibiting the adequate expressiveness in order to capture the relevant compliance rules.

Meta-analyses revealed that many related approaches (e.g., [AW09, NS08, LMX07, YMHJ06, The09b, The10c]) use the property specification patterns collected by Dwyer and Corbett in a case study [DAC99, DAC98] as the fundament for compliance rule specification. Dwyer and Corbett analyzed over 500 property specifications from different domains (in particular, hardware protocols, communication protocols, GUIs, control systems, abstract data types, avionics, operating systems, distibuted object systems, and databases). The patterns express requirements related to states/events during well-defined regions of system execution. Similar to process instance, the system execution can be interpreted as a sequence of events. Regions in which requirements become effective are specified by means of the *scopes* depicted in Fig. 2.1. Most of the scopes are self-explanatory. The *after Q until R* scope identifies the region after the first occurrence of Q that ends with the first occurrence of R.

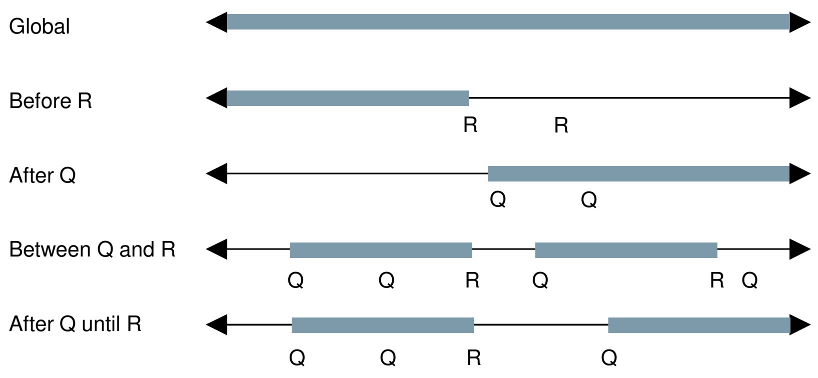


Figure 2.1.: Scopes for property specification patterns [DAC99, DAC98]

As depicted in Fig. 2.2, the patterns collected by Dwyer and Corbett are classified into two major groups. There are four *occurrence* patterns, namely *absence*, *universality*, *existence*, and *bounded existence*, and four *order* patterns where *precedence chain* and *response chain* are variations of *precedence* and *response*.

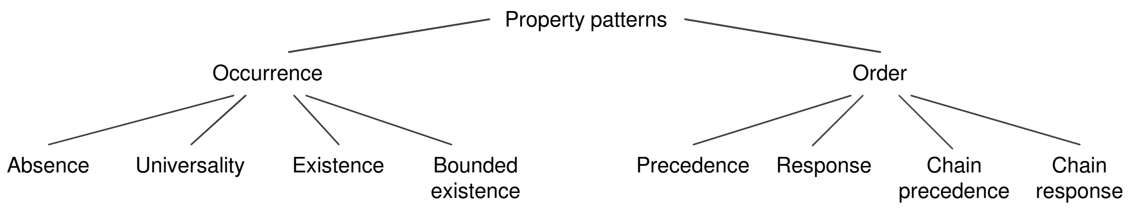


Figure 2.2.: Property specification patterns [DAC99, DAC98]

Below, we give an interpretation of the patterns based on viewing a process execution as a sequence of events attesting the execution of process activities[1]. The patterns refer to a region defined by means of scopes.

Absence Requires that the region does not contain a certain activity.

Existence Requires that the region contains a certain activity.

Bounded existence Requires that the region contains at most a specified number of a certain activity.

Precedence Requires the occurrence of a certain activity prior to the occurrence of another activity (i.e., the first activity is premise to the later activity).

Response Requires the occurrence of a certain activity in response to the occurrence of a prior activity (i.e., stimulus-response)

Chain precedence Variation of *precedence* where the occurrence of a certain activity sequence must be preceded by a particular activity within the scope of the pattern.

[1]Note that the *universality* pattern is applicable to a rather state-oriented than event-oriented environment and, thus, is not considered in the following.

Chain response Variation of *response* where a sequence of activities occurs as response to the occurrence of a certain activity within the scope of the pattern.

Note that the properties may also be interpreted not only for process activities but also for events attesting certain operations on process activities (e.g., execution).

Even though the property specification patterns stem from rather other sources than compliance requirements, the patterns are still relevant to business process compliance due to their application-independent nature as indicated by literature (e.g., [AW09, NS08, LMX07, YMHJ06, The09b, The10c]). As shown in our work [LRMD10] and related literature (e.g., [The09b, The10c, AW09, LMX07]), many compliance rules can be associated with these patterns. Though the question for the requisite expressiveness to cover real world compliance rules cannot be answered entirely, supporting expressiveness of these patterns is certainly a major requirement on a compliance rule specification approach. However, as shown in [LRMD10], we also collected compliance rules that cannot be captured adequately using these patterns. Consider, for example, the requirement from a software development project that the *feature freeze* must be followed by an *unit test* for the individual components. Then, an *integration test* for the components must be conducted. However, if changes are made to the implementation after the unit tests (e.g., due to incoming change requests), a new *unit test* becomes necessary before conducting the final *integration test*. Altogether, it should be ensured that by the end of a development cycle, the *feature freeze* is followed by an *unit test* and a subsequent *integration test* without changes to the implementations between the two tests. It is notable that the described requirement cannot be captured by a conjunction of a *chain response* and a *between scope absence* pattern. The reason is that the requirement does not prohibit changes between the tests in general but rather imposes conditions on the presence of a pair of an unit and an integration test. The support of such compliance rules requires the definition of constraints that refer to specific occurrences of activities. In the example, it should be possible to define absence constraints for a designated pair of unit and integration test (i.e., absence of changes in between these two activities). Our objective is to provide a compositional language that enables this.

Besides the expressiveness with respect to the fundamental structure of compliance rules, clearly the compliance rule specification approach must allow for the integration of context information in order to meet the demands of practical applications (cf. Section 1.1). By context information, the context of process activities is meant, for example, the data context, the temporal context, and the organizational context of the activity (e.g., which role is associated with an activity).

In this thesis, we abstain from dealing with quantitative time constraints (for example, deadlines of activities) as this requires fundamental research on the integration of time constraints with PrMSs, which is subject of other research efforts [Lan08, LWR09, LWR10]. However, we will discuss how quantitative time constraints can be integrated with our approach where appropriate. Nevertheless, the compliance rule language should be extendable such that quantitative time constraints can be smoothly integrated.

2.1.1.2. Formal semantics

In order to enable the verification of compliance rules, it must be possible to unambiguously interpret modeled compliance rules. For that purpose, the compliance rule language should exhibit clear declarative semantics. The compliance rule language should further provide formal means to reason about compliance rules. This becomes necessary, for example, for detecting conflicts among a set of compliance rules imposed on the same business process (i.e., a compliance rule or a conjunction of imposed compliance rules is not satisfiable) [LGRMD08].

2.1.1.3. Ease of use

The modeling of compliance rules for automated compliance checks typically requires the collaboration of different roles of an organization: Compliance experts (in case of external compliance regulations), who are not necessarily familiar with the process implementation details [NS08], practitioners or process users (in case of internal guidelines), who often possess low technical skills, and process experts (e.g., process designers), who are familiar with the process implementation [NS08]. In order to facilitate their collaboration, it is desirable to employ an intuitive compliance rule language. In any case, it is desirable that modeled compliance rules can be easily understood even by practitioners. This not only facilitates the communication between the different stakeholders (for example, to validate the adequacy of modeled compliance rules) but also enables the further involvement of practitioners when compliance rules are applied.

2.1.1.4. Management of compliance rules

Besides the actual compliance rule modeling capabilities, a compliance checking framework must also provide support with respect to the management of these rules. As discussed in Section 1.1, compliance rules integrated with a PrMS for automatic verification can be considered instances of automated controls. They are typically the result of a top-down process starting with abstract and informal compliance requirements as described in Section 1.1 (cf. Fig. 1.1). To also provide support for the top-down control development process depicted in Fig. 1.1, the compliance checking framework has to provide tools to document the relations between these artifacts. For example, it must be possible to identify the set of compliance rules resulting from a particular compliance objective or compliance requirement.

In addition, compliance rules may be associated with different *enforcement levels*. An enforcement level refers to how rigorously a compliance rule should be adhered to. The business rules community proposed enforcement levels for business rules (e.g., in [OMG08]), which can be adopted for compliance rules. The enforcement levels are summarized in the appendix in Section A.4 and range from *strictly enforce* for rules whose enforcement is mandatory to *guideline* for rules whose violation is tolerable. For a modeled compiance rule, it must be possible to also assign an associated enforcement level and additional information on the rule, such as an informal description.

In fact, different *compliance management frameworks* that address these issues were proposed in literature (cf. Section 1.1). For example, Kharbili et al. provide an ontology [KSP08] to capture the relations between process and regulatory artifacts. Even commercial tools providing certain management functions (for example capturing compliance requirements and assigning them to controls) are in place [AJKL06]. Hence, there is no urgent need for further research in this area, which is why we rely on existing work on compliance management. However, a compliance checking framework clearly must enable the integration of the requisite functionality or the integration within an overall compliance management framework.

The modeling and maintenance of compliance rules can further be facilitated by making use of domain models that provide a semantic view at process artifacts and help to align the terminology employed. For example, in the purchase-to-delivery scenario from Example 1.2, different payment methods can be offered to a customer (e.g., payment on account, PayPal or credit card). If, for example, a compliance rule would require actions to be taken after a payment (e.g., confirming the payment), a semantic concept *payment* can be helpful to capture compliance rules at a higher semantic level instead of modeling compliance rules for each particular type of payment. The use of domain models not only can improve the efficiency of compliance rule modeling but can also be helpful in maintaining compliance rules (e.g., when a new payment method becomes available). It further helps to separate concerns with regard to process artifacts (e.g., new or changed process artifacts and their integration) from concerns with regard to compliance rules (e.g., new compliance rules or changed compliance rules). Therefore, a comprehensive compliance checking framework must allow for the integration of domain models.

2.1.2. Compliance support along the process lifecycle

Capturing compliance requirements as checkable compliance rules is premise to automatic compliance verification. In addition to this, verification mechanisms have to be provided. Fig. 2.3 illustrates the requisite compliance checks along the process lifecycle. Generally, compliance can be verified by checking whether the implemented process model enforces the imposed compliance rules (i.e., whether it enables only the execution of compliant process instances). This is also referred to as *a priori*, *forward*, and *model-driven* compliance checking in literature [LGRMD08, KSMP08a, KMSA08, LRMGD12]. By conducting compliance checks at the process model level, not only can violations be detected but also compliance can be enforced by devising the process model accordingly (*compliance by design*) [LSG07, LGRMD08, LRMGD12]. Moreover, compliance rules can also be monitored at runtime (*runtime checking*), for example by

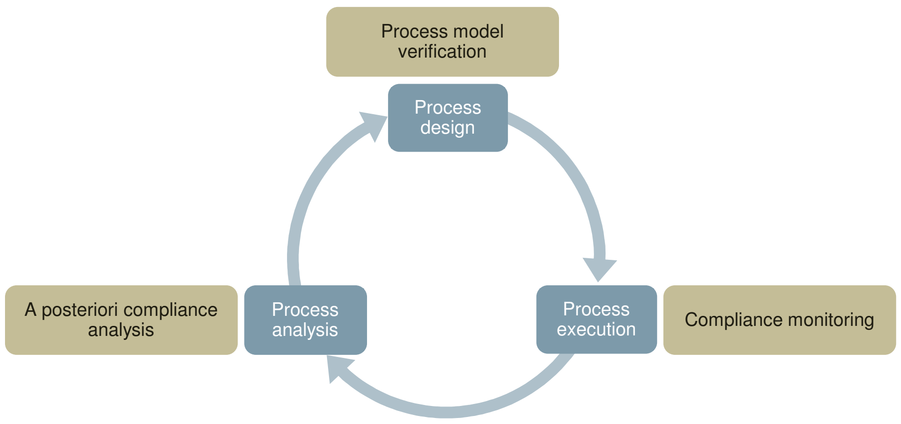


Figure 2.3.: Compliance checks along the process lifecycle

querying the process space [KSMP08a]. Obviously, at runtime, the focus is rather on detection and prediction than on enforcement when a compliance checking framework is not integrated into the process execution but just observes the latter. Finally, compliance can also be checked after process execution has terminated by analyzing the process logs (referred to as *a posteriori* or *backward* compliance checking [LGRMD08, KSMP08a, KMSA08, LRMGD12]). As this is after the fact, the analysis aims at the mere detection of happened violations. We will discuss use cases and corresponding requirements in more detail in the following.

2.1.2.1. Process design time

The use case diagram in Fig. 2.4 depicts typical use cases at process design time. In the process design phase, compliance rules must be assigned to process models. This can happen when a process model is initially created or even during the modeling process. To enable process designers to assess the process model with regard to compliance with the imposed rules, mechanisms are required to verify the process model. In particular, it must be possible to detect whether the process model enables the execution of process instances that do not comply with imposed rules. Based on the compliance report provided by the system, the process designer should be able to pinpoint and analyze the sources of noncompliance in the process model and to apply adequate measures to either enforce compliance or override the compliance rule. In order to assist the process designer in the described use cases, a compliance checking framework clearly must be able to provide support beyond pure *violation detection.* Obviously, a compliance checking framework that is only capable of answering with *compliant* or *not compliant* is not sufficient to effectively support process designers.

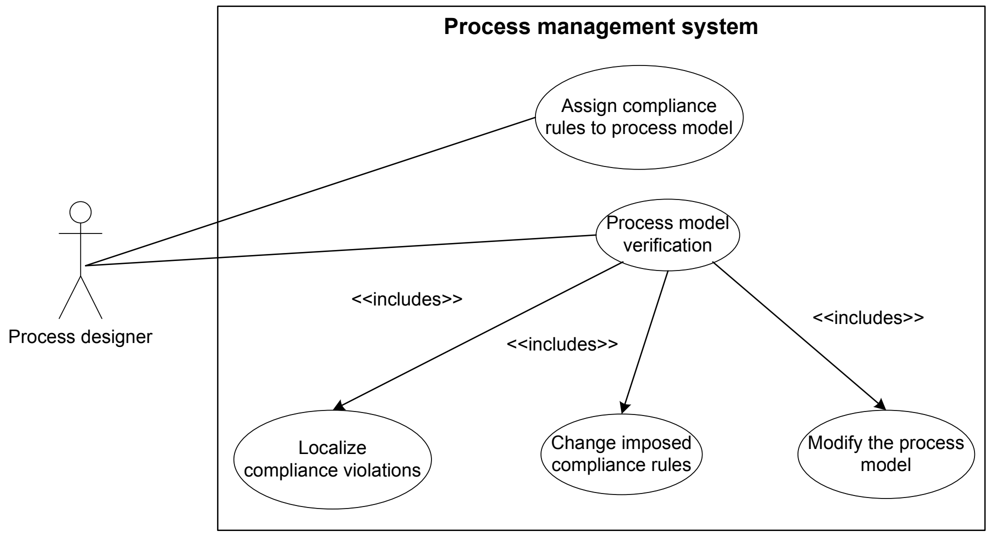

Process management system
Assign compliance rules to process model
Process model verification
<<includes>>
<<includes>>
<<includes>>
Process designer
Localize compliance violations
Change imposed compliance rules
Modify the process model

Figure 2.4.: Use cases at process design time with regard to compliance

Compliance rules typically capture normative statements that apply to particular process situations. As activities can occur multiple times in a process model, such process situations can also occur multiple times in a process model [LRMD10]. We refer to this as *multiple activations* of the compliance rule. It is notable that when compliance rules are modeled using concepts of a domain model (cf. Section 2.1.1.4), the respective compliance rules are even more likely to have multiple activations within a process as multiple activities can be assigned to an abstract concept (e.g., different payment activities such as *payment on account* can be assigned the abstract concept *payment*). Conceivably, different rule activations can be in different compliance states. While some activations can be a source of noncompliance, other may be compliant. Against this background, the compliance checking framework should not only be able to provide feedback on the overall enforcement of the compliance rule, but also should be able to identify the activations of the compliance rule and assess their compliance states. We refer to this as *per activation compliance diagnosis*. Being provided with compliance reports at this level of granularity, the process designer will be able to localize the sources of noncompliance more easily. This also enables dealing with compliance violations in a flexible manner as the compliance rule can, for example, be overridden for only selected activations. Ideally, the detected violations can be visualized in the process model (*violation tracing*). In addition, *explanations* indicating the root-cause of detected violations are desirable. This would further facilitate the application of adequate measures to resolve noncompliance (e.g., by applying suitable process changes).

As compliance verification of large and complex process models involving data can be a costly operation due to state explosion [KLRM+10], a compliance checking framework must further accommodate strategies for dealing with that.

In sum, we identified the following requirements for a compliance checking framework with respect to design time support:

D1 detection of compliance violations within process models,

D2 provision of per activation feedback,

D3 support for advanced compliance diagnoses (explanation and visualization compliance violations), and

D4 accommodation of strategies for dealing with state explosion.

2.1.2.2. Process runtime

In order to ensure that process instances are compliant with imposed rules, compliance can be enforced already at the process model level. However, compiling particularly data-aware compliance rules that refer to runtime data into the process model often leads to overly complex process models. This is due to the fact that such rules often apply only in specific cases (process situations) that have to be accounted for by, for example, introducing suitable splits and branches in the process model. Hence, for compliance rules with lower enforcement levels such as *guideline* (cf. Section 2.1.1.4), it is doubtful whether introducing additional complexity to the process model is justifiable. In other cases, compliance rules may refer to process data that is not encoded in the process model but is only available at runtime making verification at the process model level infeasible. In consequence, a compliance-aware PrMS has to allow for creating and deploying process instances from a process model for which not all imposed compliance rules are enforced. In order to maintain control over imposed compliance rules, it is desirable to monitor compliance with rules not enforced in the process model.

While process instances are usually created based on a predefined process model in the context of PrMS, there are also scenarios in which processes are executed in a rather ad-hoc manner. To be broadly applicable (e.g., to processes scattered over different information systems), a compliance checking framework should also address such scenarios in which processes are not executed based on a known process model or are created and executed in an ad-hoc manner [LRMKD11].

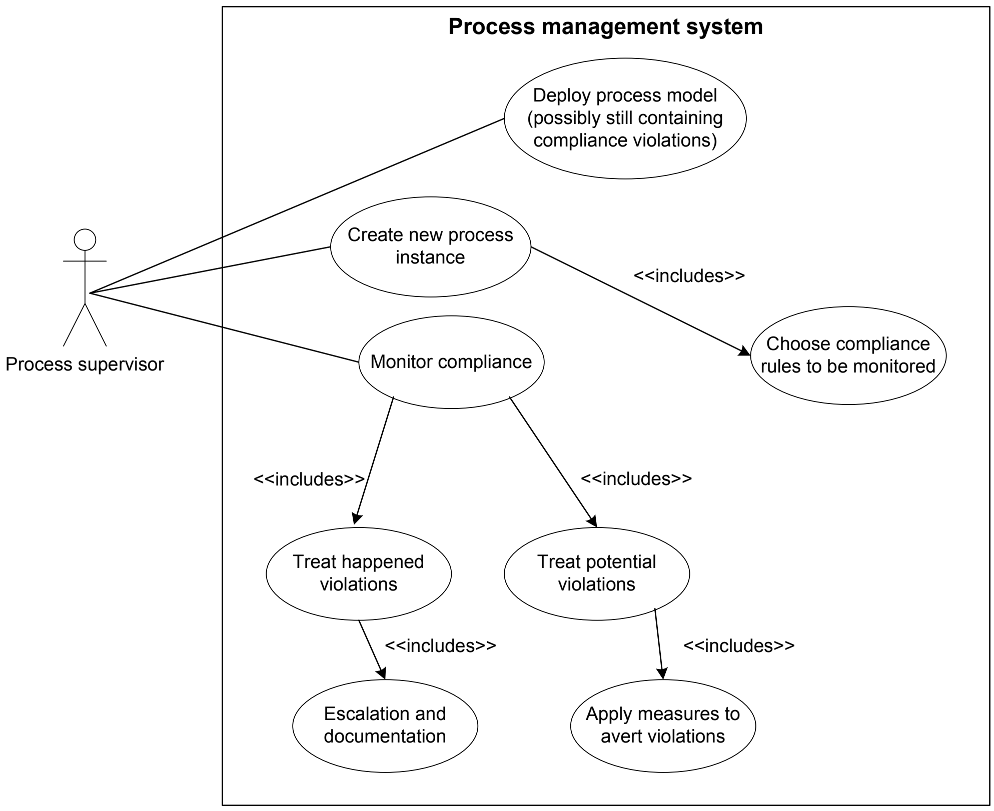

Process management system
Deploy process model (possibly still containing compliance violations)
Create new process instance
<<includes>>
Choose compliance rules to be monitored
Process supervisor
Monitor compliance
<<includes>>
<<includes>>
Treat happened violations
Treat potential violations
<<includes>>
<<includes>>
Escalation and documentation
Apply measures to avert violations

Figure 2.5.: Use cases at process runtime with regard to compliance

As illustrated in the use case diagram in Fig. 2.5, the process supervisor may decide which compliance rules shall be monitored when starting a process instance (e.g., only selected compliance rules or all imposed compliance rules that are not yet enforced at the process model level). The monitoring tier of the compliance checking framework then monitors the events occurring during process execution (e.g., completion of a process activity) and analyzes them with respect to effects on compliance.

A major objective of compliance monitoring is to timely *detect happened compliance violations*. This enables immediate actions, such as application of remedies, controlled overriding of violated compliance rules, or documentation of compliance violations. Timely documentation of happened violations (e.g., by adding an explanation for the violation by the actor in charge) ensures meaningful input for a posteriori analysis of the executed business process and can also avoid the costs that come with an extensive a posteriori root-cause analysis. In the clinical practice, for example, documentation of deviations from predefined pathways can constitute a significant requirement [BSB$^+$07].

Generally, each activation of a compliance rule can be in one of three compliance states in a stage of process execution: satisfied, violated, or violable [LRMKD11, MMWA11]. Satisfied and violated are permanent states. A violable rule activation, however, can become both violated or satisfied depending on the events observed in the future. The state violable can have very different semantics depending on the particular rule activations. For example, a violable rule activation may become violated when a particular event is observed (e.g., a particular activity is executed). In contrast, another violable rule activation may become violated when a particular event will not be observed. Obviously, to prevent the actual violation of these rule activations, different measures are necessary (i.e., preventing an event in the first and scheduling an event in the second case). In order to provide assistance in proactively preventing violations, the monitoring tier has to make the state violable transparent to process supervisors. In particular, support with regard to how to satisfy a rule activation is desirable. Being aware of what is necessary to satisfy a rule activation, the process supervisor may be able to arrange the suitable measures (e.g., ensure that a particular activity is carried out by placing it in an actor's worklist). This fosters compliant behavior and *prevents compliance violations*. In case of violations, the compliance checking framework should be able to *provide explanations* for the reached compliance state in oder to assist process supervisors in identifying the root-cause of violations.

If a process instance is executed based on a predefined process model, the possible behavior of the instance is already encoded in the process model. Thus, this information should be utilized to *anticipate potential violations* in the future (cf. Fig. 2.5).

In sum, we identified the following requirements for a compliance checking framework with respect to runtime support:

R1 timely detection of happened violation,

R2 provision of per activation feedback,

R3 support for advanced compliance diagnoses (explanations for and prevention of compliance violations), and

R4 anticipation of potential future violations based on behavior predefined in the process model.

2.2. Vision, contributions, and research methodology

As described in Section 1.1, compliance requirements impose constraints on the process space. Against the background of PrMSs, to implement compliance controls, therefore, means to device measures to control the enforcement of these constraints. In this thesis, we address constraints focusing on the process flows, so-called *compliance rules*. The compliance rules impose constraints on the occurrence, absence, and temporal ordering of activities or process-related events. We further argued that hard-coded solutions to check the enforcement of the compliance rules are not a viable option due to lack of sustainability. Our objective is, therefore, to provide a generic framework that supports the modeling of compliance rules, their integration with process models, and automated compliance checks against imposed rules at process design and runtime. In the following, we sketch the “big picture” of such a framework in Section 2.2.1. In Section 2.2.2, we detail the contributions of this thesis. Our research methodology is described in Section 2.2.3.

2.2.1. Vision

Fig. 2.6 summarizes essential features of a comprehensive compliance checking framework to support the process lifecycle. In the following, we describe a walkthrough of the different phases of the process lifecycle.

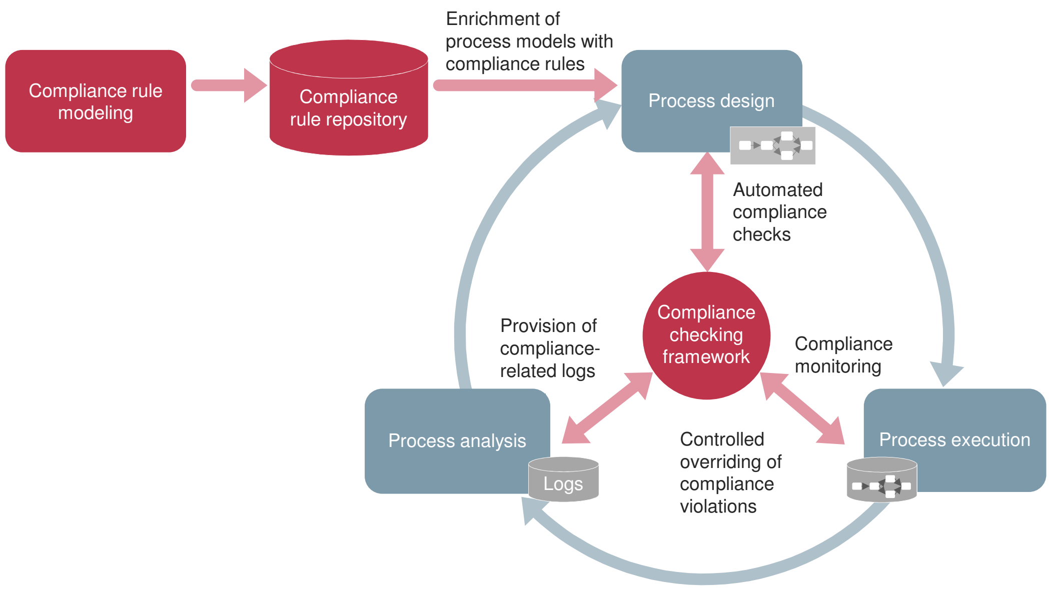

Compliance rule modeling
Compliance rule repository
Enrichment of process models with compliance rules
Process design
Automated compliance checks
Compliance checking framework
Provision of compliance-related logs
Compliance monitoring
Process analysis
Logs
Controlled overriding of compliance violations
Process execution

Figure 2.6.: Essential features of a compliance checking framework

Rule modeling and integration Compliance requirements are modeled using a formal yet easy to use compliance rule language. The integration of compliance rules with process models and process instances is achieved by using process artifacts for compliance rule modeling. Thus, compliance rules are imposed on the artifacts of the process space. A domain model can further be used as interface between compliance rules and process artifacts. A designated *compliance rule management system* (RMS) is used for managing compliance requirements, compliance objectives, and compliance rules. The RMS is an important component in the big picture but it is not in the focus of this thesis. Here, we rely on existing work.

Process design At process design time, the process designer can select compliance rules (or complete rule sets belonging to a compliance objective) from the RMS and assign them to the process model. In order to facilitate the verification of process models against imposed rules, the verification functionality is integrated into the process modeling environment. Thus, there are no gaps between the tools. In order to enable efficient compliance checks, mechanisms for dealing with the complexity arising from intricate process models are provided by the framework.

Based on the compliance checks, comprehensible compliance reports are provided. For each compliance rule, not only its general enforcement is assessed but also details on its activations can be provided to the process designer. Ideally, the process situations in which a compliance rule becomes activated can be visualized to the process designer to pinpoint compliance violations. Explanations can be generated for detected compliance violations. Based on the compliance with imposed rules, compliance reports for compliance objectives and compliance regulations can be generated. This facilitates auditing the process with regard to implemented compliance controls.

Depending on the enforcement level of the compliance rules, the process designer can decide whether their enforcement at the process model level makes sense. He may also take back imposed rules or override only selected activations of a compliance rule. The latter is particularly relevant when multiple activations occur due to defining rules on abstract concepts of the domain model and tailoring becomes necessary.

Process execution After completing a process model, it is then deployed and process instances are created based on the model. For each process instance, the process supervisor may select the compliance rules to be monitored. Alternatively, the system may automatically decide on the rules to be monitored at runtime. Additionally, the process supervisor may select additional rules to be imposed on a process instance (for example, rules that apply only to selected customers in an order-to-delivery process).

For the rules to be monitored, the events occurring during process execution are analyzed with regard to their effect on the rule enforcement. In a compliance cockpit equipped with, for example, dashboards, a process supervisor can control the compliance state of rules imposed on process instances. The compliance cockpit is supposed to visualize all rule activations and their compliance state. For rule activations that are neither satisfied nor violated, the compliance checking framework can provide details on the activations' particular states and, thus, can assist the process supervisor in preventing violations. Moreover, the framework is able to provide explanations for compliance violations making it easier to evaluate the extent of violations and to decide on adequate remedies.

In case a scheduled event leads to the violation of a rule activation, the actor in charge will be notified. Depending on the compliance rule's enforcement level and the actor's authorization, the rule activation may, for example, be overridden with the actor providing an explanation for doing so. In cases where strict enforcement in required, the PrMS could, for example, decline the scheduled event and notify the process supervisor. All such incidents are logged by the system, thus, ensuring that later audits can be provided with meaningful input.

Process analysis Even though our compliance checking framework does not specifically address process analysis, the data collected during process execution can be helpful to assess the executed processes. Particularly the documentation on rule activations and deviations taken by authorized users are valuable to the process analysis.

A framework as envisioned would not only be a valuable complement to existing PrMSs. It would also address the need for "affordable IT and IS tools that facilitate compliance management self-audits and compliance monitoring activities in general" identified by the case study conducted by Abdullah et al. [ASI10].

2.2.2. Contributions

In this thesis, we provide the fundamental building blocks for a compliance checking framework as envisioned in Section 2.2.1:

Compliance rule modeling: We introduce a formal language for compliance rule specification. The language uses a graphical notation consisting of different primitives (e.g., nodes and edges) resembling the ones known from process modeling. The modeling primitives represent the occurrence, absence, or ordering of activity executions. Using our approach, compliance rules expressing constraints on the occurrence, absence, and temporal ordering of activity executions can be modeled by means of graphs, so-called *compliance rule graphs* (CRG). The CRG language does not only cover the property specification patterns [DAC99] but also gives the flexibility of composing more sophisticated compliance rules. Moreover, the language is extensible, thus, enabling the easy incorporation of future research results.

Compliance verification: In order to enable automated compliance checks in the different phases of the process lifecycle, we equipped the CRG language with operational semantics. The latter enables the operational execution of CRGs such that process models as well as running process instances can be verified against them. Following the requirements with respect to the granularity and the comprehensiveness of compliance diagnoses, our approach enables valuable insights into detected compliance states. Thus, it becomes possible to derive meaningful and intelligible compliance reports at the granularity of individual compliance rule activations contained in a process model or a process instance. This is not only beneficial for identifying the root-cause of compliance violations or for generating explanations for violations but also for deriving suitable measures to avert potential but not yet manifest violations. The proposed verification mechanisms are fully-automated, thus, leveraging the potential of automatic compliance verification.

Altogether, the concepts proposed in this thesis can be used to equip PrMSs and other kinds of process-aware information systems with compliance checking functionality. The proposed compliance checking framework can be integrated into an overall compliance management framework (as, for example, proposed in [GLM$^+$05, The11]) where its role is to address PrMSs as target systems. For leveraging the practical application, the core building blocks of the compliance checking framework are complemented by concepts and considerations on questions arising around the actual compliance verification. In particular, we describe considerations on reducing the complexity of compliance verification in general and on the interaction with end-users. This thesis further describes the evaluation of our concepts. Specifically, it introduces our proof-of-concept implementation and case studies conducted in our work. A more detailed overview of the contributions of this thesis will be provided in Chapter 4.

2.2.3. Research methodology

Design research is concerned with the development of technology-based solutions to important and relevant business problems and must produce a viable artifact in the form of a construct (vocabulary and symbols), a method (algorithms and practices), a model (abstractions and representations), or an instantiation (implemented and prototype systems) [HMPR04, CPRS05]. Aiming at developing a concrete solution in the form of a viable compliance checking framework, our research adopts central ideas from design research. According to Andriessen [And07], design research aims at providing answers to design problems formulated as explorative questions (e.g., "How can we improve situation X?") or questions aimed at testing a hypothesis. In design research, the production of an viable artifact as solution to relevant problems constitutes a search process using available means to reach desired ends [CPRS05].

The specific iterative research process applied in the context of our work is illustrated in Fig. 2.7. Initially, we started with requirements analysis based on case studies and meta-analyses. For that, we focused on data from the clinical domain and from meta-analyses. Based on that, we formulated an initial vision of a compliance checking framework [LRD06, LRD08]. Based on the vision, we developed concepts that were also implemented in prototypes. The concepts do not necessarily address all aspects of the vision at once but may also focus on certain aspects. Using the developed prototypes, we were able to analyze and evaluate developed concepts using data from practical applications or from meta-analyses. This may lead to further development and implementation iterations (e.g., in case the developed concepts do not yet cover all relevant aspects or do not yet yield adequate solutions). The evaluation of developed concepts may also result in a completely new iteration leading to modifications or refinements of the vision when studies reveal additional requirements (e.g., [LGRMD08, LRMGD12]).

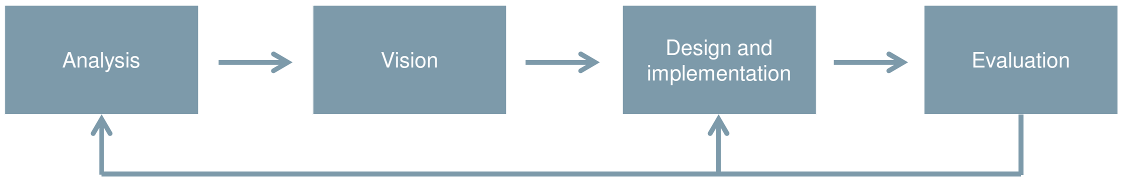


Figure 2.7.: Research process

This iterative process is reflected in our publications and in the prototypes that constitute the SeaFlows Toolset [LKRM$^+$10], our proof-of-concept implementation. For example, in an earlier iteration, we developed an approach for fundamental but not yet very expressive compliance rules [LRD06, LRD08]. For this, we developed a prototype, which is described in [LKRM$^+$10]. As evaluation revealed limitations (e.g., w.r.t. expressiveness), we then explored more expressive compliance rules and the application of linear temporal logic and model checking techniques [Knu08, KLRM$^+$10]. The results of this research is also incorporated into a prototype implementation [LKRM$^+$10]. Evaluation of this revealed that the employed formalism can constitute a major obstacle to the practical application. Our research on an alternative graph-based formalism was more promising and fruitful [LRMD10] and also resulted in a prototype implementation [LKRM$^+$10]. The process of exploring different strategies for compliance verification based on this approach is reflected by several publications [LRMD10, LKRM$^+$10, LRMKD11, Mer10, LRMGD12].

3 State-of-the-art

In Chapter 2, we compiled requirements on a compliance checking framework that addresses both process design and process runtime. In this chapter, we provide an overview of the state-of-the-art and analyze existing approaches with respect to the requirements and our vision of a comprehensive compliance checking framework. Hence, this chapter discusses existing related work from the functional perspective. Subject-specific related work will be discussed in each of the following chapters describing our technical solutions whenever this is appropriate.

As described in Section 2.2.1, a compliance checking framework must enable the specification of compliance rules and accommodate mechanisms for verifying process models and process instances against them. However, the bulk of existing related approaches does not address all scenarios identified in Section 2.2.1 but focuses on specific use cases in a rather isolated manner. Section 3.1 describes approaches focusing on design time compliance verification while approaches focusing on runtime compliance monitoring are described in Section 3.2. Integrated approaches that address both process design and runtime are discussed in Section 3.3. Finally, Section 3.4 concludes this chapter.

3.1. Approaches with emphasis on design time

Enforcing compliance by verifying process models against compliance requirements is a strategy advocated by a multitude of related approaches (e.g., [SGN07]). In the following, we distinguish between these approaches based on the techniques employed for compliance verification.

Model checking approaches Model checking provides techniques for the verification of a system specification (i.e., a model) against certain properties (e.g., deadlocks or fairness) [BBF$^+$01, CGP99]. As illustrated in Fig. 3.1, the model and the property are given as input to a model checker. The latter explores the model w.r.t. the property to be checked. In case the property does not apply, the model checker typically provides a counterexample.

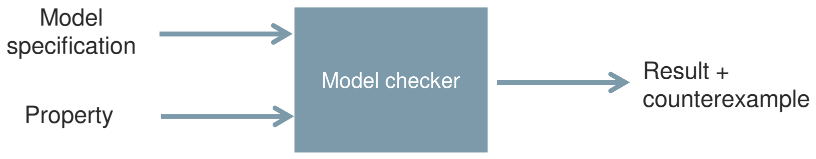


Figure 3.1.: Compliance checking by means of model checkers

As model checking is well-researched and hence provides a variety of languages, techniques, and tools, it is not surprising that model checking is adopted by a variety of related approaches for business process model verification. In this context, model checkers are employed as verification tool in order to find out whether or not a process model complies with a rule. In order to utilize existing model checkers for verifying process models against compliance rules, the latter have to be specified such that they can be evaluated by model checkers. Different languages are available for the specification of properties, for example, temporal logic (e.g., *linear temporal logic* or *computation tree logic*) or automatons may be used. As the linear time semantics of *linear temporal logic* (LTL) is closer to the mental model of a business process, LTL is preferred over *computation tree logic* (CTL), which has branching time semantics [The08, The09b, ETHP10a]. Both languages are well-researched and can be seen as decidable notational variants of “modal” fragments of first-order logic [HSG04]. The property specification patterns described in Section 2.1.1.1 can be mapped to both LTL and CTL. The downside of temporal logics is, however, that they are less suitable for practitioners due to their complexity [DAC99]. Specifically, LTL and CTL employ a navigational mental model (i.e., one navigates through time) that can make compliance rule specification using these languages intricate for non-experts.

The compliance verification approaches proposed in literature (e.g., [FS10, ADW08, FPR06, FUMK06]) employ model checkers such as SAL or NuSMV for verification. Some approaches further seek to overcome shortcomings of LTL or CTL with respect to understandability and user-friendliness by providing graphical notations (e.g., [FS10, LMX07]) or predefined rule patterns (e.g., [YMHJ06, ADW08, ASW09, AWW09, FESS07, KGE11]). This aims at facilitating compliance rule specification. Specifically, Feja et al. introduce a graphical notation for CTL (referred to as G-CTL) for modeling compliance requirements [FS10, FWS11]. The model checker NuSMV is utilized for verifying EPC process models against such G-CTL compliance rules. In [LMX07], Liu et al. propose a framework for verifying process models against compliance requirements based on LTL model checking. In particular, they propose the *business property specification language* (BPSL) that is directly related to LTL properties. Förster et al. [FESS07, FESS06, FES05] present an approach for validating process models against quality constraints. Quality constraints are specified in terms of process patterns in *process pattern specification language* (PPSL). PPSL patterns, in turn, can be transformed into specifications in LTL. An extension of the approach is presented in [KGE11] where PPSL patterns can be mapped to CTL specifications. In the context of web service composition and coordination, the question arises whether or not a choreography complies with certain constraints [FPR06]. In [YMHJ06], Yu et al. introduce an approach for the specification of properties and for the property-based validation of BPEL processes. The properties are based on property patterns [DAC99]. For process validation, a model checking approach is employed. Foster et al. [FUMK06] introduce an approach for validating the interactions of web service compositions against obligation policies specified using message sequence charts. In Section 6.4.1, we discuss related compliance rule specification approaches, such as LTL, BPSL, or PPSL, in more detail when introducing our approach.

In order to enable compliance diagnoses beyond binary feedback (i.e., compliant or noncompliant), the output of the model checker, in many cases a single counterexample, has to be transformed back in order to derive a diagnosis [KLRM^{+}10, AW09]. This can be a challenging task depending on the internal representation of the model. Hence, some compliance verification approaches based on model checking provide advanced feedback by either interpreting the feedback provided by the model checker or by introducing additional mechanisms for diagnosis [AW09, AWW09, ETHP10b]. Awad et al. [AW09] introduce an approach for visualizing and explaining compliance violations that is applied when model checking reveals at least one compliance violation. The approach focuses on the explanation of violations of a specific set of compliance rule patterns. For each such rule pattern, so-called anti-patterns capturing process structure indicating a violation are defined. These anti-patterns are formulated by means of BPMN-Q queries [Awa07] that can be applied to process models specified in BPMN [OMG11]. Subgraphs of the process model matching the BPMN-Q queries are returned by the query processing engine and can be used for visualizing the compliance violation directly in the process model. In [AWW09], this approach is extended for data-aware compliance rule patterns. In order to extract the data conditions under which a violation occurs, *temporal logic querying* [Cha00] is applied while BPMN-Q queries are used for visualizing detected violations in the process model. Elgammal et al. seek to compensate the limitations of model checkers w.r.t. compliance diagnosis by introducing an approach based on current reality trees [Det97]. Specifically, in [ETHP10b], Elgammal et al. provide current reality trees for property specification patterns introduced by Dwyer et al. [DAC99] and some additional patterns (such as *exclusive choice*). For that purpose, Elgammal et al. identified all potential causes of a violation for each pattern. These potential causes are then compiled into current reality trees of the corresponding patterns. By traversing the tree answering questions associated with a detected violation, the user is guided to the root-cause of the violation and is further provided with guidelines and suggestions that help to resolve the compliance violation.

Approaches based on detecting logical inconsistencies A further type of approaches seeks to detect compliance violations in process models by identifying inconsistencies in specifications. Specifically, this is achieved by representing the process model using some logic formalism such that compliance violations can be related to logical inconsistencies between the process model specification and the rule formulas.

In [DKRR98], an approach for compliance validation based on *concurrent transaction logic* (CTR) is introduced. For validating a process model against constraints specified in CTR, the process graph is transformed into a CTR formula. This allows for reasoning about conflicts between the formulas. In [GMS06], compliance validation is addressed from the business contract perspective using *formal contract language* (FCL) for specifying contracts. The compliance of a process model with a given contract is validated by transforming the process model into a representation similar to the contract notation. This allows for reasoning about contract violations.

Graph analysis approaches One way to verify compliance of process models with imposed compliance rules is to search the process model, specifically the graph representation of the model, for presence or absence of certain structures.

In [BBB$^+$11], Becker et al. propose an approach that relates compliance checking of process models to subgraph isomorphism and adopts existing subgraph isomorphism algorithms. Premise to this is that compliance rules are represented as graphs (e.g., a sequence). The approach then basically searches for occurrences of the subgraph in the process graph. Thus, this approach is designed to seek confirmation of a certain pattern in the process model.

In our early work, we investigated graph analysis as a method to verify compliance [LRD06, LRD08, LKRM$^+$10]. The basic idea of this approach is to automatically derive structural conditions from imposed compliance rules. Then, process models can be verified against compliance rules by checking whether associated structural conditions apply. Depending on the process description language employed, structural compliance checking can be conducted in a more efficient manner than exploring the behavior encoded by the process model (e.g., for a block-structured language such as ADEPT [Rei00]). However, the approach is restricted to a simple compliance rule language and does not support data-awareness.

Further approaches The Comcert approach introduced by Accorsi et al. [ALS11] relates compliance verification of process models to checking the conformance between a petri net representing the process model and a Petri net representing the compliance rule. Thus, noncompliance can be detected by simulating both nets. Since rule specification based on petri nets is rather cumbersome, Comcert provides a textual notation for specifying rules. Such rules are then mapped to the corresponding Comcert petri net rule pattern.

In [GHSW08], an approach based on annotating activities with their effects represented by logical propositions is introduced. Induced state changes of artifacts involved in process can be considered effects of an activity. FCL is utilized to formulate constraints based on the activity annotations (i.e., FCL rules specifying under which state conditions certain obligations arise). By propagating the effects of activities throughout the process model, obligations arising during process execution can be detected and evaluated. A similar propagation approach is used for compliance checking in [WGH08]. This approach aims at detecting states of the process execution in which certain constraint clauses are violated.

In [WKH08a], Wörzberger et al. introduce three types of compliance rules, namely *inclusion*, *existence*, and *precedence*. For their implementation, Wörzberger et al. model rules as OCL constraints. These OCL constraints can be integrated into and evaluated by the process modeling environment described in [WKH08b].

Discussion As described in Section 2.1.2, compliance verification at design time constitutes one phase of the process lifecycle. In fact, most of the techniques applied by the approaches discussed in this section are not directly suitable for compliance monitoring (particularly when the process model is unknown) when incremental compliance checks become necessary. This necessitates the accommodation of separate approaches for design and runtime support, which may require separate compliance rule specifications and result in different kinds of diagnoses. It seems to be more promising to provide an integrated approach that supports both design and runtime for the same rule language, which is a major objective of this thesis.

We investigated the application of graph analysis to process verification in our previous work [LRD06, LRD08, LKRM$^+$10]. Specifically, we were able to provide per activation compliance diagnoses that explain the root-cause of the compliance violation (requirements D2 and D3). However, this approach provides only limited support for data-awareness as only the process structure is taken into account.

Relating compliance verification of process models to the detection of logical inconsistencies among specifications is only proposed by a few approaches (e.g., [DKRR98]). The requirements with respect to granularity and comprehensiveness of compliance diagnoses are not sufficiently addressed (requirements D2 and D3) by these approaches as feedback is typically provided in the form of a single counterexample. In [DKRR98], for example, the proof procedure returns a Horn goal that can be interpreted as the smallest subpart of the process consistent with imposed constraints.

Conceivably, model checking was adopted by many approaches due to the powerful yet decidable property specification languages and the availability of existing model checkers such as SAL. However, the complexity of temporal logics can become an obstacle to the practical utilization of model checkers for compliance verification. To overcome this, several graphical notations or pattern-based approaches were proposed. They will be described in more detail when discussing compliance rule specification approaches in Section 6.4.1. The requirements with respect to granularity and comprehensiveness of compliance diagnoses (i.e., requirements D2 and D3 in Section 2.1.2.1) have not yet been sufficiently addressed. While approaches such as [AWW09, ETHP10b] exist that seek to provide advanced compliance diagnoses by introducing additional mechanisms, these approaches are still restricted to specific predefined compliance rules and cannot be generalized. Feedback at the level of particular rule activations (requirement D2) as envisioned in Section 2.1.2.1 is not directly supported and, hence, necessitates additional mechanisms. As model checking is typically applied to verify complex reactive systems, the state explosion problem is well-known and well-researched by the model checking community. Thus, strategies for dealing with state explosion exist that can be adopted for process verification (requirement D4). In this thesis, we adopt the idea of model checkers to explore the process model and provide an approach that aims at more advanced support with respect to granularity and comprehensiveness of compliance diagnoses.

3.2. Approaches with emphasis on runtime

Compliance monitoring at process runtime basically means to observe events stemming from process execution and to evaluate the process behavior exhibited with respect to compliance.

Depending on the availability of process model information, potential future compliance violations can be predicted. In the following, we describe the different compliance monitoring approaches proposed in related work based on the techniques they employ.

Approaches using monitoring automatons As illustrated in Fig. 3.2, one approach to monitor compliance with imposed rules is to employ an observer automaton that will reach an accepting state if the rule to be checked is satisfied. If the automaton is not in accepting state at the termination of a process instance, the compliance rule is violated. Using automatons, it is further possible to determine whether a compliance rule can still become satisfied during process execution. This can be done by checking whether an accepting state is still reachable from the current state of the automaton. If no accepting states are reachable, the compliance rule is violated and cannot be rendered satisfied in the further course of process execution. This can happen, for example, if a compliance rule imposes the absence of a certain activity and the very activity is observed during the process execution. Such information can be used to prevent compliance violations. However, the derivation of measures to satisfy an imposed compliance rule (cf. requirement R3 in Section 2.1.2.2) requires an analysis of the automaton. As compliance rules are typically not modeled as automatons, they first have to be modeled using a formalism from which an automaton can be generated, such as *linear temporal logic* (LTL), as illustrated in Fig. 3.2. A major weakness of the observer automaton approach lies in the lack of support for providing compliance diagnoses for individual compliance rule activations (cf. requirement R2 in Section 2.1.2.2). For that, additional mechanisms become necessary.

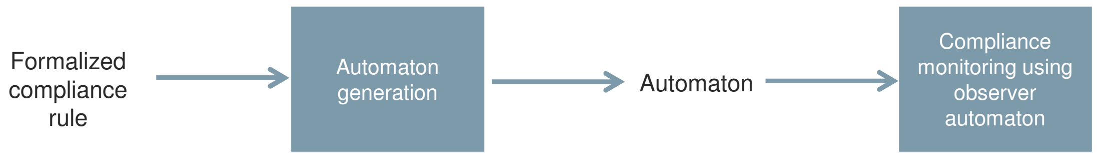

Formalized compliance rule
Automaton generation
Automaton
Compliance monitoring using observer automaton

Figure 3.2.: Automaton-based compliance monitoring

In [MMWA11, MWMA11], Maggi et al. suggest an approach that expresses compliance rules as LTL constraints. In this approach, each LTL constraint is translated to a local automaton and the entire set of rules is monitored through a colored automaton. A colored automaton is obtained as the synchronous product of local automatons. As such, this global automaton is able to identify violations caused by the interplay of two or more constraints. In addition, the colored automaton includes information about the accepting states of the original local automatons. Therefore, when a violation occurs it is possible to identify which constraints from the original set have been violated. To hide the complexity of LTL from the modeler, graph notations for frequently used constraint patterns based on the work of Dwyer and Corbett [DAC99], such as DECLARE [PA06], were suggested. Santos et al. [SFV$^+$12] propose a set of rules to check control flow and resource distribution in a business process. These rules are represented through a set of automatons. These automatons are used to build a controller that can be exploited for guiding the execution of the process by enabling and disabling tasks. This adopts ideas from declarative process modeling and execution as proposed by Pesic et al. [Pes08, PA06].

Approaches based on querying for violation patterns As illustrated in Fig. 3.3, the basic idea of violation-based approaches is to query the evolving execution trace (i.e., the execution history) of a running process instance for patterns of compliance violations. If at least one violation pattern of a compliance rule is present in the execution trace, the compliance rule is violated. Obviously, in order to query the execution trace, the possible ways to violate a compliance rule have to be identified. This can be done automatically or manually. For evaluating queries, existing frameworks and technologies such as complex event processing (CEP) [JML09] can be applied.

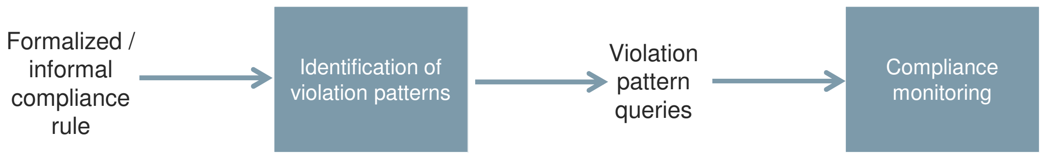

Formalized / informal compliance rule
Identification of violation patterns
Violation pattern queries
Compliance monitoring

Figure 3.3.: Violation-based compliance monitoring

A major weakness of violation pattern based approaches lies in the derivation of violation patterns from a given compliance rule. This is particularly an issue for rules that are formulated positively (i.e., what must be done instead of what must not be done). For example, if an execution of activity A requires a subsequent execution of activity B, the corresponding violation pattern would be "A is executed without subsequent B“. For simple compliance rules or basic relations (as, for example, introduced in [WZM$^+$11, WPDM10]), all violation patterns can be anticipated. However, for more complex compliance rules that can be violated in multiple ways, automatic computation of violation patterns to identify all possible violations becomes a challenge. Being focused on the detection of violation patterns, this kind of monitoring approach aims at after-the-fact detection. As such, no support is provided with respect to violation prevention.

Beheshti et al. [BBMNS11] present a framework for analyzing event logs based on the concepts of folders and paths. The proposed framework uses folders to group related events in the logs and allows users to identify relevant paths based on a given correlation condition. Event processing technologies are further used by numerous compliance monitoring frameworks to detect violations, e.g., in the COMPAS project [HMZD11, BDL$^+$10]. COMPAS aims at providing integrated compliance support for process design and runtime and will be discussed in more detail in Section 3.3.3. In [AJKL06], Agrawal et al. propose a framework for taming compliance with the Sarbanes-Oxley Act, particularly with the section on installing an effective system of internal controls, by using database technology. The authors suggest checking the conformance of transactions with prescribed workflows during execution and after-the-fact. In case of anomalies, enforcement can be achieved by declining the transaction. Additionally, noncompliant transactions can be admitted while documenting the anomalies at the same time in order to enable later audits.

Further approaches In [MMC$^+$11], Montali et al. introduce an event calculus formalization for ConDec [PA06] constraints that supports the identification of constraint activations. This approach is able to deal with temporal scopes. The formalization is, however, restricted to specific property specification patterns, namely *existence*, *absence*, and *response* constraints (cf.

Section 2.1.1.1). Alberti et al. [ACG^{+}08] report on monitoring contracts expressed as rules using the notion of happened and expected events. At runtime, events are aggregated in a knowledge base and that serves as basis to reason about violations.

Discussion Approaches focusing solely on runtime monitoring neglect the process design time, an important scenario in the business process lifecycle. Thus, the exploitation of information encoded in the process model for predicting future compliance violations (requirement R4 described in Section 2.1.2.2) is not addressed.

Clearly, all approaches discussed in this section are able to detect violations of modeled compliance rules (requirement R1 in Section 2.1.2.2). However, as previously described, a comprehensive compliance checking framework should be able to provide support beyond pure violation detection. Automaton-based approaches exhibit appealing features, such as the possibility to analyze whether a compliance rule can still become satisfied and how to satisfy a compliance rule. However, the exploitation of these features typically necessitates further analysis of the automaton such as analyzing whether the end state is still reachable. Furthermore, the existing approaches do not pay attention to the transparency of reached compliance states of a compliance rule during monitoring. Compliance diagnosis is only addressed in terms of reproducing the event trace that has led to a compliance violation. Explanations beyond that and diagnosis for specific rule activations are not addressed (requirement R3 and R2 described in Section 2.1.2.2). Querying-based approaches are able to detect violations at the level of particular rule activations (requirement R2 in Section 2.1.2.2). However, they focus solely on after-the-fact detection. Advanced issues such as the prevention of violations have been neglected. Thus, it seems that a combination of the strengths of both kinds of approaches would be desirable.

3.3. Integrated approaches

Besides approaches focusing on solely process design or process runtime, several integrated frameworks were proposed in literature. In the following, the most important ones are described.

3.3.1. $\mathcal{S}$CIFF

The work conducted by Alberti and Chesani et al. based on the so-called $\mathcal{S}$CIFF framework addresses both buildtime and runtime issues [ACG^{+}06, ACG^{+}08, ACG^{+}07, CMMS07b]. $\mathcal{S}$CIFF consists of a declarative language based on abductive logic programming and an operational framework. It was originally developed to specify and reason about multi-agent protocols.

Events are the first class entities in $\mathcal{S}$CIFF. The $\mathcal{S}$CIFF language is composed of entities for expressing events and expectations about events as well as relationships between events and expectations [ACG^{+}08]. An event occurring during process execution is referred to as *happened event*. The desired behavior is represented as *expectations*. Expectations are modeled as abductive predicates as they represent events that may or may not occur. A positive expectation models an event that must occur while a negative expectation models an event that is expected not to occur. Using the negation operator, the deontic modalities (i.e., permission, obligation, and prohibition) can be expressed based on positive and negative expectations. The literals of the $\mathcal{S}$CIFF language can be used to define so-called *integrity constraints* (IC) that express behavior constraint and, thus, correspond to compliance rules. ICs are basically defined as forward rules where the rule antecedent can contain a conjunction of all the elements of the language and the rule consequence contains a disjunction of conjunctions of all literals of the language except for happened events [ACG$^+$08]. Further, restrictions can be defined over finite domain variables by integrating logic constraint programming. The $\mathcal{S}$CIFF language does not accommodate a notion for ordering relation between events (such as *after* or *before*). Thus, the ordering of events is determined by a designated parameter indicating the time at which an event is expected or not expected. By defining constraints on the time parameter, ordering relations can be established within ICs. Using the $\mathcal{S}$CIFF language, the patterns collected by Dwyer and Corbett (cf. Section 2.1.1.1) can be captured. However, $\mathcal{S}$CIFF is not restricted to these patterns. The $\mathcal{S}$CIFF language is powerful and enables expressing complex compliance rules constraining the process behavior. Due to its formal notation, however, it may not be suitable for practitioners. A sound and complete proof procedure, defined as a set of transition rules, constitutes the operational specification of $\mathcal{S}$CIFF. The $\mathcal{S}$CIFF proof procedure builds a proof tree by transforming one node to others in a rewriting system manner [ACG$^+$08]. In order to perform runtime checking, the $\mathcal{S}$CIFF framework adopts abduction to dynamically generate the expectations. Being defined as abducibles, the expectations are hypothesized by the abductive proof procedure, i.e., the proof procedure makes hypotheses about the behavior. These hypotheses must be confirmed by the happened events in a confirmation step. If no set of hypotheses can be confirmed, a violation is detected. The proof procedure has been implemented and integrated into a reasoning and verification tool.

Using the $\mathcal{S}$CIFF proof procedure, monitoring of compliance with imposed ICs can be realized. In their initial work [ACG$^+$06], Alberti et al. apply the $\mathcal{S}$CIFF framework to the runtime verification of integrity constraints imposed on web service interactions. In [ACG$^+$06], two kinds of violations are addressed by the approach: events that occur without being explicitly expected and expected events that do not happen. In [CMMS07a], the approach is applied to testing the conformance of careflow processes from the healthcare domain. For that purpose, the process model is translated into $\mathcal{S}$CIFF constraints and a knowledge base. Then, executions of careflow processes can be tested for conformance with the model. The $\mathcal{S}$CIFF proof procedure can be modified and adopted for checking whether a model specification enforces a certain property (g-$\mathcal{S}$CIFF) [ACG$^+$08, ACG$^+$07]. This corresponds to process model verification at design time. The property to be checked is formulated as a conjunction of $\mathcal{S}$CIFF literals. The proof procedure is basically utilized to find a history of events that proves that the property does not hold (i.e., a counterexample). For that purpose, a $\mathcal{S}$CIFF goal is formulated based on the negated property to be verified. Then, the proof procedure tries to yield a history of events complying with both the goal and the model specification. For that purpose, it is necessary that the model is specified using $\mathcal{S}$CIFF. Otherwise, the proof procedure cannot be applied. This may be an obstacle for the practical application.

In their work, Alberti and Chesani et al. do not put emphasis on intelligible feedback (requirements D3 and R3 in Section 2.1.2.1 and 2.1.2.2, respectively). At design time, for example, only a single counterexample is generated by the proof procedure. While the runtime support of the approach is quite advanced and interesting information can be derived from the nodes of the proof tree (e.g., still pending expectations), Alberti and Chesani did not further seize this in their work, for example, for deriving compliance diagnosis. The detection of multiple rule activations and provision of per activation diagnoses was not addressed (requirement D2).

3.3.2. ICCOMP

As a result of the SAP project ICCOMP, Namiri proposes a framework for model-driven management of internal controls for business process compliance in his thesis [Nam08]. The work of Namiri addresses both design time and runtime compliance issues.

For design time compliance verification, Namiri proposes to formally define the process model to be verified by means of a formal ontology (formalized in OWL-DL). Then, compliance requirements are expressed according to the terms and concepts defined in the formal ontology using the *semantic web rule language* (SWRL)[1]. In this approach, compliance requirements are basically defined as production rules [Nam08]. For the verification of the process model, Namiri suggests the application of an inference engine. As the approach for design time is not detailed in [Nam08], the applicability cannot be assessed. However, it seems that the requirements concerning intelligible compliance diagnoses and per activation diagnoses (R3 and R2 in Section 2.1.2.2, respectively) are not addressed.

For runtime compliance monitoring, Namiri introduces a control model [NS07a, NS07b, Nam08, NS08]. Each control basically consists of an event, after whose occurrence during the course of the execution of a business process a set of conditions has to hold (or not hold). Scopes (i.e., *global*, *before*, *after*, and *between* described in Section 2.1.1.1) can be utilized to define events. The control conditions can be composed from basic queries on the process state, process properties, or the state of previous controls. In addition to that, each control further defines recovery actions to be undertaken if the conditions of a control fail. When executing the process, the predefined recovery actions will be scheduled for the process instance when the control conditions fail. This can be considered a process instance change. In order to facilitate control design, Namiri et al. propose a set of control patterns addressing frequent controls such as for accommodating the four-eyes-principle [NS07c].

While the approach proposed by Namiri et al. addresses both design and runtime issues, the two parts of the solution are not truly integrated but constitute rather isolated solutions. Specifically, different approaches are utilized to capture rules to be checked at design and at runtime. Requirements w.r.t. granularity and comprehensiveness of compliance diagnoses as described in Section 2.1.2 were not addressed. However, Namiri et al. present an interesting framework for managing compliance requirements and respective controls that can be adopted by our approach.

3.3.3. COMPAS

The COMPAS project[2] aims at devising a compliance technology framework that can be utilized to ensure compliance of the composition of business processes and services. It proposes an overall compliance management framework that enables to define and manage com-

[1]http://www.w3.org/Submission/SWRL/
[2]http://www.compas-ict.eu

pliance requirements from different sources and at different abstraction levels and to associate them to specific controls. The controls are defined as compliance rules (declarative) or process fragments (procedural). COMPAS introduces a view-based modeling framework that accommodates a distinct view for integrating compliance concerns and compliance metadata [THO$^+$10, HTZD10]. Verification and monitoring features are accommodated in the overall COMPAS architecture, which is described in [The11]. Thus, COMPAS addressed both process design and process runtime. The results of the COMPAS project were summarized in the project's deliverables [The08, The09b, The09a, The10a, The10c, The10b, The11].

The compliance specification language (referred to as CRL for Compliance Request Language) is described in [The09b, The09a, The10a, The10b]. COMPAS provides a compliance rule specification approach that integrates property specification patterns [DAC99] (cf. Section 2.1.1.1) and higher-level patterns. From compliance rules composed from such patterns, *linear temporal logic* (LTL) formulas can be generated. With the CRL tool, COMPAS delivered a prototype for defining compliance requirements and manage them together with compliance risks, sources, targets, and controls [The10b]. In addition to compliance rules, process fragments that are designed to enforce certain compliance requirements constitute compliance controls that can be integrated into process model design [SAL$^+$10, SLM$^+$10].

As the compliance rules are ultimately formalized in LTL, existing model checkers such as Spin may be applied to verify compliance of process models [TEHP12]. According to the architecture proposed in [The11], different verification tools may be utilized. The importance of intelligible compliance diagnoses was emphasized in [The10b] (requirement D3 in Section 2.1.2.1). The approach for generating compliance diagnoses described in [ETHP10b] is based on the definition of current reality trees for compliance rule patterns that can be traversed in order to derive compliance diagnosis and remedy strategies (cf. also Section 3.1). This approach is, however, restricted to specific compliance rule patterns. More general solutions and particularly per activation compliance diagnosis (requirement D2 in Section 2.1.2.1) were not addressed.

COMPAS relies on complex event processing (CEP) and business protocol monitoring for the detection of compliance violations during process execution [The11, THO$^+$10, HMZD11] (requirement R1 in Section 2.1.2.2). In [HMZD11], the compliance monitoring part of COMPAS is described detail. In the framework, low-level events from the process execution engine are aggregated to meaningful business events by a CEP engine. The business events are then passed to a business intelligence (BI) component that checks compliance and decides on actions to take. How compliance checks are conducted, is not described in detail. In [THO$^+$10], CEP queries are employed to detect violations of quality of service (QoS) policies. How the queries are derived from compliance requirements is not detailed in the paper. Compliance prediction is addressed by Rodriguez et al. [RSDC10]. Specifically, this approach focuses on the definition of key compliance indicators based on process data (in analogy to key performance indicators).

Process models constitute only one of the compliance targets that are addressed in the COMPAS project. It seems that the major focus of COMPAS was on devising an overall comprehensive and mature compliance management framework. Thus, no emphasis has been put on the specific techniques for compliance verification. In contrast, we focus on providing techniques for compliance rule specification and verification at design and runtime in this thesis. The approach proposed in this thesis can be integrated into the overall COMPAS architecture [The11] in order to support compliance rule definition and both design and runtime compliance verification.

As we abstain from addressing management and governance issues evolving around business process compliance, our work can benefit from the management and integration functionalities developed in the COMPAS project when being integrated into such an overall framework.

3.3.4. Further approaches

REALM In [GLM$^+$05, GMP06], Giblin et al. propose REALM, a meta-model for modeling compliance requirements and for managing them in a systematic lifecycle in an enterprise. REALM stands for *regulations expressed as rule models.* The vision of this approach is to provide a general meta-model that enables capturing compliance requirements. REALM models are supposed to provide the basis for subsequent model transformations, for example, into concrete process models, access or monitoring rules, or data retention policies. The latter are deployed into the business and IT infrastructure of the enterprise while still preserving the association with respective passages of compliance regulations. This systematic compliance management is supposed to enable traceability from the regulations to concrete controls implemented in the systems. This, in turn, enables enterprises to demonstrate to auditors how compliance is achieved [GLM$^+$05].

A REALM model consists of a *concept model*, describing the objects of the domain, a *compliance rule set* for capturing the actual rules, and *meta-data* for capturing information about the structure of the legal source as well as lifecycle data. In particular, the objects and relationships occurring in a regulation are formalized in a concept model (i.e., a domain ontology). The compliance rules on these concepts are formalized using *real-time temporal object logic*, a combination of *timed propositional temporal logic* and concept models in UML. The temporal logic used for REALM compliance rules is equipped with the usual temporal operators (i.e., *globally*, *eventually* and *once*) as known from LTL with past operators. In order to enable time constraints, a *freeze* quantifier is introduced, a time variable that enables to refer to the point in time in which a formula holds. Thus, it becomes possible to define constraints on the time variables.

While the transformation of REALM models into concrete artifacts for compliance enforcement and control at both design and runtime is a major part of the vision underlying this approach, the technical transformation part has been neglected. The transformation is only addressed in [GMP06] where Giblin et al. describe an approach for generating event correlation rules from REALM models. This approach automatically derives correlation rules for a set of REALM rule templates. The correlation rules can be utilized for detecting compliance violations at runtime. Even though REALM does not provide a compliance checking framework as targeted by this thesis, the systematic approach of REALM for managing and translating compliance requirements to concrete controls can be adopted to complement our approach.

SUPER The work conducted by Kharbili et al. in the SUPER project[3] aims at developing a compliance management framework. In [KSMP08b], Kharbili et al. propose modeling compliance regulations by using policies as a formal tool for declarative implementation of compliance management. This is supposed to enable the splitting of compliance management into policy

[3]http://www.ip-super.org

management and policy implementation and enforcement. Kharbili et al. envision a policy framework consisting of an upper level policy ontology, an intermediary level for domain-specific policies, and a bottom level for rules as concrete implementations of policies. The bottom level policy ontology is supposed to support a variety of rule languages in order to address different target systems. A discussion of different policy frameworks can be found in [TBJ$^+$03].

In [KSMP08a, KSP08, KS08], Kharbili et al. propose a high-level architecture for a compliance checking framework that also accommodates the enforcement of rules in business process models and rule monitoring. Concrete approaches for realizing the enforcement of rules or for monitoring policies were not addressed. Similar to Giblin et al., Kharbili et al. do not provide a compliance checking framework but rather introduce a compliance architecture. With respect to the categories to check and enforce business process compliance discussed in [KSMP08a], the approach proposed in this thesis can be integrated into the architecture proposed by Kharbili et al. in order to realize *policy-aware process monitoring* and to *enforce semantic operative rules in business process models.*

3.4. Summary

As became apparent in the state-of-the-art discussion, the bulk of compliance checking approaches proposed in literature focuses on either design time or runtime compliance verification. In Section 3.1 and Section 3.2, we described approaches addressing compliance checks at process design and process runtime, respectively. For verifying process models, the bulk of proposed approaches utilizes existing model checkers. For runtime compliance monitoring, the bulk of existing approaches proposes the use of observer automatons or the detection of compliance violations by querying for violation patterns (e.g., using CEP). As pointed out in the discussion, the proposed approaches still lack support for requirements identified in Section 2.1. Specifically, design time compliance verification approaches either rely on complex formal languages or on a set of predefined compliance rule patterns. While formal languages such as LTL or CTL can become an obstacle for the practical application due to their complexity, predefined patterns based on the property specification patterns may be too restrictive (cf. Section 2.1.1.1). Intelligible compliance diagnoses at the granularity of particular compliance rule activations are not sufficiently supported. Existing approaches addressing the explanation of compliance violations, such as [AW09, AWW09, ETHP10a], are restricted to specific rule patterns and rely on additional mechanisms beyond the actual compliance checking approach. The proposed runtime compliance verification approaches lack support for easily interpretable of compliance states. While automaton-based approaches enable the derivation of actions to render a compliance rule satisfied in general, this necessitates further analysis of the automaton. In addition, a compliance rule is typically monitored as a whole while individual compliance rule activations that may undergo different compliance states during process execution (cf. Section 2.1.2.2) are not distinguished. Approaches that detect noncompliance by querying for presence of compliance violation patterns in the execution trace focus on after-the-fact detection. Thus, they lack support for the prevention of compliance violations.

Only a few approaches address compliance in the context of the process lifecycle. The most prominent ones among these approaches, however, aim at devising a comprehensive compliance management framework and do not put much emphasis on compliance verification techniques. As previously discussed, the “default” techniques employed by these approaches, such as model checking, do not essentially differ from the isolated approaches addressing solely process design or process runtime. $\mathcal{S}$CIFF is most related to the work proposed in this thesis as it constitutes a compliance checking approach addressing both process design and process runtime. However, $\mathcal{S}$CIFF is not suitable for an integrated approach as it has major shortcomings w.r.t. compliance diagnoses particularly concerning design time support (cf. Section 3.3.1).

In this thesis, we focus on providing the technology for modeling compliance rules and for verifying process models and running process instances against imposed rules. Thus, our *compliance checking framework* can be embedded into *compliance management frameworks* such as COMPAS where it can benefit from management functionalities while the compliance management framework is enhanced with a compliance checking approach that supports both design and runtime. Thus, the enforcement of the same compliance rules can be checked at the process model as well as at the process instance level. We believe that this synergy can leverage existing compliance management frameworks that now still rely on rather isolated compliance checking mechanisms.

4 The SeaFlows compliance checking framework in a nutshell

In Chapter 2, we described requirements on a compliance checking framework. Specifically, a compliance checking framework has to support the modeling of compliance requirements and has to accommodate mechanisms to verify compliance at both process design and process runtime. Key to the practical application of a compliance checking framework will further be its ability to provide meaningful compliance diagnoses. As discussed in Chapter 3, existing compliance checking approaches predominantly focus on specific scenarios rather than on the integrated support of process design and process runtime. The requirements with respect to granularity and comprehensiveness of compliance diagnoses are not yet addressed in an adequate manner. Existing compliance frameworks that employ a holistic view on business process compliance issues such as proposed in the COMPAS project, in turn, focus on compliance management in the large rather than on the actual compliance checks. While these frameworks are valuable for establishing an overall and systematic approach to compliance management in general, they particularly rely on existing techniques, such as model checking, for compliance verification. These techniques, again, are applicable only to specific scenarios (e.g., process design time) and suffer from known limitations as discussed in Chapter 3. Thus, there is still need for a compliance checking framework that supports process design and process runtime and that is capable of providing comprehensive compliance diagnoses at the granularity of particular rule activations. In this thesis, we describe such a compliance checking framework. As our work was conducted in a research project called SeaFlows, we refer to the framework as the SeaFlows compliance checking framework in the following. The proposed framework can become part of an overall integrated compliance management framework as, for example, envisioned in the COMPAS project (cf. Section 3.3.3). Before it is introduced in detail, we first provide an overview on the components of the framework. In the following, Section 4.1 introduces the components and Section 4.2 summarizes in which chapters these components will be described in detail.

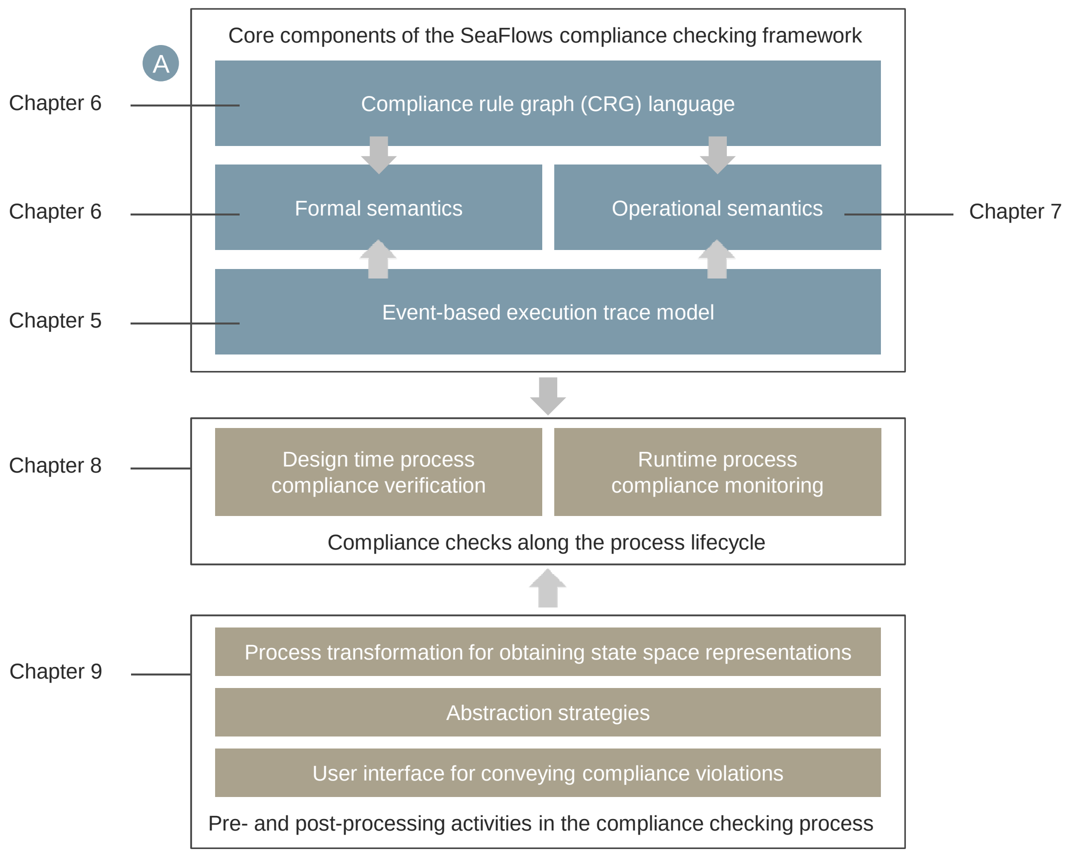

Core components of the SeaFlows compliance checking framework
A
Chapter 6
Compliance rule graph (CRG) language
Chapter 6
Formal semantics
Operational semantics
Chapter 7
Chapter 5
Event-based execution trace model
Chapter 8
Design time process compliance verification
Runtime process compliance monitoring
Compliance checks along the process lifecycle
Chapter 9
Process transformation for obtaining state space representations
Abstraction strategies
User interface for conveying compliance violations
Pre- and post-processing activities in the compliance checking process

Figure 4.1.: Building blocks of the SeaFlows compliance checking framework

4.1. Building blocks of the compliance checking framework

The building blocks of the proposed compliance checking framework are illustrated in Fig. 4.1. The framework's core components are shown in Fig. 4.1 A: a language to specify checkable compliance rules, its formal semantics, and the operational semantics of the language for conducting compliance checks. These are footed on an event-based execution trace model that ensures interoperability. In the following, we provide a brief introduction of each building block.

4.1.1. Event-based execution trace model

As a major goal of our work is to provide a general framework that is not tied to a specific process description language, the footing of the proposed compliance checking framework is constituted by an event-based execution trace model. The general and extensible event notion is applicable to events in PrMSs and other business applications. It is specifically compatible to the event notion often employed for *process mining* [VBDA10]. Execution traces consisting of such events serve as language-independent representation of process executions in our framework.

4.1.2. The compliance rule graph language

Our goal is to provide a simple yet powerful and extensible compliance rule language. Based on our experience from experimenting with LTL, we aimed at hiding the complexity of a formal language from the rule modeler. Following approaches in literature (e.g., [Awa07, ADW08, AP06, PSSA07]), we adopt the assumption underlying graph-based process description languages that a graph notation is suitable to represent constraints on the occurrence and ordering of activities. Thus, we propose an approach for modeling compliance rules by means of graphs. The rules modeled using this approach are referred to as *compliance rule graphs* (CRGs). In contrast to many approaches in literature, the CRG language is not a collection of patterns but a compositional language that consists of modeling primitives such as nodes and edges expressing occurrence, absence, or ordering of activity executions. These primitives can be assembled to compose complex compliance rules. The property specification patterns described in Section 2.1.1.1 can be modeled using the CRG language. However, being compositional, the CRG language also enables more complex compliance rules that cannot be captured directly using the property specification patterns (cf. Section 2.1.1.1). The CRG language can be parsed in order to generate natural language descriptions. Due to its extensibility, further features can be easily integrated.

Following the requirement for formalization (cf. Section 2.1.1.2), the CRG language is associated with clear formal semantics. Specifically, each compliance rule modeled using the CRG language can be mapped to a rule formula specified in first-order predicate logic (PL1) that can be interpreted over execution traces. The rule formulas constitute a fragment of PL1. Hence, existing algorithms can be exploited for analyzing rule formulas. Being based on the event-based execution trace model, the formal semantics of CRGs is not restricted to a specific process description language. Thus, CRGs can be utilized to constrain business processes specified using a multitude of description languages.

4.1.3. Operational semantics of compliance rule graphs

The rationale behind equipping CRGs with operational semantics is to enable checking process models and process instances against imposed CRGs using the same mechanisms. The basic idea is to exploit the graph structure of CRGs for compliance checking. In order to encode compliance states such that they can be interpreted for generating compliance diagnoses, we introduce state markings for CRG nodes. This way, each compliance state with respect to an imposed CRG that a process execution may yield can be represented using the very CRG and suitable state markings. In contrast to states of automatons generated from temporal logic formulas as used for explicit model checking, the compliance states thus encoded can be interpreted easily as they refer directly to the CRG structure.

A major objective of our work is to enable meaningful compliance reports as this requirement still constitutes a major limitation of most related approaches in literature. One benefit of the proposed approach is specifically that explanations for compliance violations can be derived from the compliance states encoded using CRGs and state markings (cf. requirements D3 and R3 described in Section 2.1.2). Even if no compliance violation has occurred (yet), the meaningful compliance states can provide valuable insights into the compliance situation. In particular, it is easy to derive measures to avert potential violations from the information encoded in a reached compliance state.

The operationalization of CRGs is inspired by pattern matching mechanisms and is conducted by applying rules that alter compliance states according to observed events in a process to be verified. Thus, the operational semantics can be applied to both design and runtime compliance checks as it supports an incremental procedure (as required for runtime compliance monitoring). It further enables to "instantiate" a CRG for each new activation of the compliance rule observed in the process. Thus, the framework is able to provide compliance reports not only on the general enforcement of the CRG but also on the particular rule activations in a process and their individual compliance (cf. requirements D2 and R2 described in Section 2.1.2).

Similar to the formal semantics of CRGs, the operational semantics is defined over event-based execution traces. Thus, the proposed compliance checking framework can be applied to business processes specified using a multitude of description languages. This will leverage the practical application of the framework.

4.1.4. Application of the compliance checking framework in the process lifecycle

The core components of the proposed compliance checking framework as depicted in Fig. 4.1 A provide the fundament for realizing compliance checks along the process lifecycle within a PrMS and other process-aware information systems. At process design time, a process model is verified against an imposed CRG by exploring the process model with regard to compliance with the CRG. In this process, the CRG operational semantics is applied in order to detect compliance states that can be yielded by the possible executions of the process model. Apparently, a process model complies with a rule if it only enables the execution of process instances that comply with the rule. In this thesis, we describe different strategies for exploring a process model that may be applied depending on the desired level of granularity with respect to the compliance report. Based on the detected compliance states, meaningful compliance reports for process designers can be generated that can help process designers to resolve compliance violations. At process runtime, events observed during process execution are processed in order to provide detailed feedback on the compliance state of a running process instance. In addition to that, the information encoded by the corresponding process model can be exploited to account for the predefined future behavior.

4.1.5. Pre- and post-processing activities in the compliance checking process

Beyond the mechanisms for modeling compliance requirements in a checkable manner and for compliance verification, further aspects are vital for a comprehensive compliance checking framework. Specifically, questions on necessary activities before and after conducting the actual compliance checks arise. In this thesis, we outline how processes can be transformed into state space representations for compliance checking for the example of the process description language ADEPT [Rei00]. We further address the state explosion problem that may arise due to complex processes to be verified (cf. Section 2.1.2.1) by pointing out a variety of abstraction strategies adopted from literature and developed in the SeaFlows project. After conducting the actual compliance checks, detected compliance violations need to be conveyed to users in an intelligible manner. Considerations on this challenge are also introduced in the thesis.

4.2. Structure of the thesis

The remainder of this thesis is structured as follows:

Chapter 5 first provides process fundamentals to clarify basic notions such as process model or process instances used in the process world. Moreover, it introduces the logical model that serves as footing of the proposed compliance checking framework. Specifically, the notion of events and execution traces will be described in detail.

Chapter 6 introduces the CRG language and its formal semantics. In particular, the modeling primitives of the CRG language and their composition are described. Furthermore, the translation of CRGs into rule formulas specified in first-order predicate logic is detailed. Then, the formal semantics of rule formulas is defined over execution traces by stipulating how such formulas are formally interpreted over an execution trace.

Chapter 7 defines the operational semantics of CRGs. Specifically, the markings for CRGs to represent compliance states are introduced. Rules that define the transitions of compliance states are further provided. This is accomplished by providing so-called execution and marking rules that alter markings in an adequate manner. Chapter 7 also describes how the operational semantics can be applied at different levels of granularity affecting the granularity of compliance checks.

Chapter 8 describes how the operational semantics of CRGs is applied to realize process design and process runtime compliance checks. In particular, it describes the exploration of a process model with respect to compliance with imposed CRGs. In addition, it describes how execution trace based process instance monitoring is realized using the compliance checking framework. Chapter 8 further introduces ideas on compliance predictions at the process instance level by exploiting the future behavior predefined in the corresponding process model.

Chapter 9 addresses issues evolving around the actual compliance checks. Specifically, it complements the proposed compliance checking framework with concepts on the transformation of process models into state space representations for the example of WSM nets [Rei00]. In addition, Chapter 9 contains concepts collected from literature and own ideas on taming the state explosion problem that such verification tasks often suffer from. Finally, considerations on how to convey compliance violations are presented in Chapter 9.

Chapter 10 presents our proof-of-concept implementation. The research procedure underlying this thesis (cf. Section 2.2.3) resulted in a variety of different tools. These tools are summarized in the SeaFlows Toolset. The SeaFlows Toolset comprises tools for modeling compliance rules using the CRG language and for conducting compliance checks using the operational semantics of CRGs. Moreover, the SeaFlows Toolset also features prototype implementations that showcase the application of abstraction strategies for taming the state explosion problem.

Chapter 11 describes our efforts to evaluate the proposed compliance checking framework. Specifically, it presents a pattern-based evaluation of the CRG approach by modeling the property specification patterns described in Section 2.1.1.1 using the CRG language. These patterns have been adopted by a multitude of related approaches for compliance verification. Moreover, Chapter 11 summarizes the findings and lessons learned from two cases studies where we applied the proposed compliance checking framework to real world data / processes. It not only showcases the feasibility of our framework but also points out highly interesting future extensions.

Chapter 12 summarizes the contributions of the thesis and outlines future research challenges.

5 Fundamentals

A broadly applicable compliance checking framework requires a sound and generic footing. The challenge is to provide a general process representation that does not tie us to a particular process description language. In the following, we introduce a logical model that will serve as fundament for defining the formal and operational semantics of our compliance rule language. As compliance rules impose constraints on the process behavior, we opted for *execution traces* to represent process executions. The independence from particular process description languages is achieved by using a generic event model, on the one hand, and by making only few assumptions on process models and process instances in the logical model, on the other hand. To represent the behavior encoded by process models and process instances from different process description languages, we use finite state automatons that represent a set of execution traces (referred to as *process event graphs*). In Section 5.1, fundamentals on process models and process instances are provided. Section 5.2 then introduces the logical model.

5.1. Process fundamentals

Before going into detail on the logical model that serves as fundament to our compliance checking framework, we first introduce some process fundamentals that will be used throughout this thesis.

5.1.1. Process models

According to Weske [Wes07], "business processes consist of a set of related activities whose coordinated execution contributes to the realization of a business function in a technical and organizational environment". *Process models* capture business processes. In particular, a process model describes the workflows of a particular business process. This is done by specifying the flow of process artifacts such as activities or events within the process and the data flows between them.

Process models can be defined using different formalisms. Though there exists approaches in literature that suggest to model processes using other means (for example, using logic formulas [MDK+03, FW99]), the bulk of approaches in literature and applied in commercial tools prefers graph-based process models. Following this, we stick to graph-based process models basically consisting of nodes and edges. Directed edges are used to describe the relationships between the nodes of the process model. A process model node can be associated with an *activity*, an *event*, or a *gateway* (a routing construct) [Wes07]. The *control flow* of the process model is established via directed edges that specify the ordering relations between these process artifacts.

Activities An activity is a unit of work conducted in the business process. Activities, whose implementation can be provided through a piece of software or by manual tasks not involving the system, are the building blocks of operational business processes [Wes07]. An activity may be associated with multiple process nodes as a task may be carried out multiple times within a business process[1].

Events Events represent specific states that are relevant to the process such as the reception of a message.

Gateways Gateways are constructs that route the control flow. Each node with multiple incoming or outgoing edges constitutes a gateway node. For simplicity, we assume that, like normal nodes, gateway nodes are assigned an activity (if no activity is assigned, then a designated dummy activity can be used).

For the operational execution of a business process, it is further necessary to specify who is responsible for performing which tasks. At the process model level, this can be done by means of user assignment rules specifying which agents are eligible or obliged to perform a certain task (e.g., executing an activity, sending an event) [LR00, Wes07]. Thus, it becomes possible to express, for example, which activities shall be performed by which agents.

Beyond the pure control flows, a process model can further contain information on *data flows*. In particular, process artifacts can be associated with data objects, which can be produced, manipulated, and consumed by the former.

Standardization attempts have led to languages such as WSBPEL [WSB07] or BPMN [OMG11]. However, still a multitude of further process description languages to specify process models are available, e.g., WSM nets [Rei00, Rin04] or Event Process Chains (EPC) [Aal99] as used by ARIS. Example 5.1 illustrates a process model specified as a WSM net and in BPMN.

[1] To be more precise, we could further distinguish between activity types and activities (i.e., instances of a certain type). Then, the activity type is defined only once in a particular domain while multiple instances of that type can be utilized in a process model. This distinction is made for example by ADEPT [DRR+08, AAD+04, Rei00]. However, as this distinction does not affect the considerations presented in this thesis, we will use only the concept of activities.

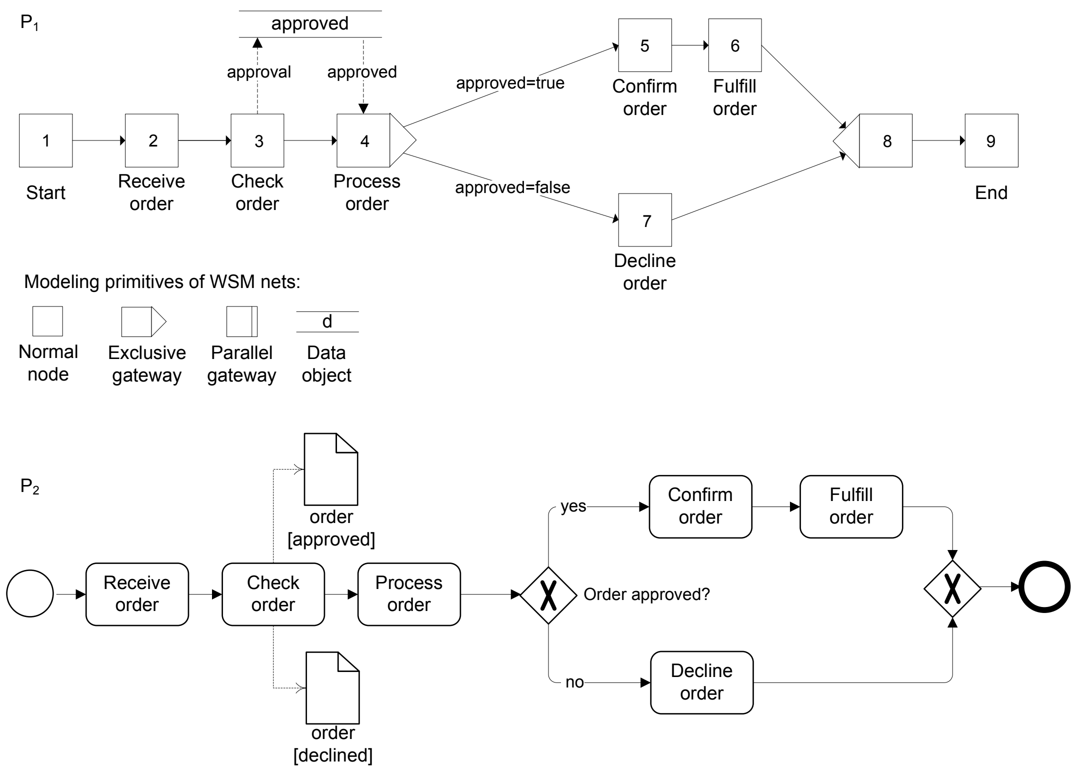


Figure 5.1.: A simple order process modeled as WSM net (P_1) and as BPMN model (P_2)

Example 5.1 (Process models):
In Fig. 5.1, a simple order process is modeled as WSM net [Rei00, Rin04] (P_1) and by means of the *business process modeling and notation*, BPMN in brief (P_2). Basically, the order is checked after its reception. The result of this task is recorded in a data object (*approved* in P_1, *order* in P_2). For example, one can think of a boolean flag to indicate whether or not the order is approved. The order is further processed and depending on the approval, the order is either confirmed and fulfilled or declined. In P_1, the data edges are annotated with the parameters as which the data object *approved* is written / read by the respective activities. Note that for P_1, the activity *Process order* constitutes a data-based exclusive gateway that takes the data object *approved* as input and chooses an outgoing path after its completion depending on the conditions associated with the outgoing edges ([Rei00], p.60).

In this thesis, we assume that process models under investigation comply with structural soundness criteria [Wes07] of the underlying description language.

5.1.2. Process instances

Process models describe how a business process is conducted. In order to enable the execution of the modeled process within a process management system (PrMS), the process description language employed has to provide operational semantics that defines how the model is processed. A running process started from a process model is referred to as *process instance*. During the process execution, the nodes of the process model are processed according to the defined order and associated activities and events are sent to the worklists of the corresponding agents or automatically processed by the system [LR00]. When activities are carried out, the data parameters and corresponding data objects are assigned concrete values. Depending on the paths within the process model chosen during the execution of the process instance and the instance-specific process data, only a subset of the possible future process behavior encoded by the process model may still be reachable for the process instance (for example due to data-based gateways). This is illustrated by Example 5.2.

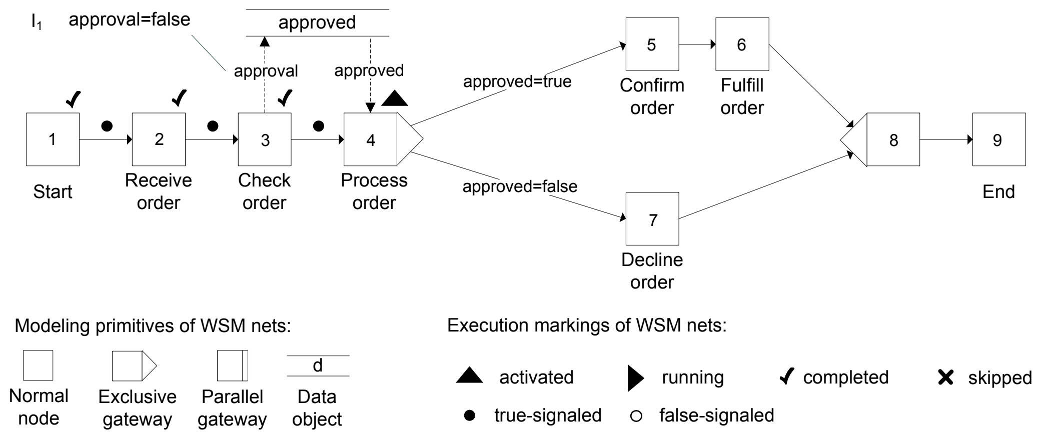


Figure 5.2.: A running process instance of P_1 from Fig. 5.1

Example 5.2 (Process instances):
Fig. 5.2 depicts a running process instance of P_1 from Fig. 5.1. Three activities have been executed so far and activity *Process order* is activated. Informally, the execution history of I_1 would comprise the following:

- execution of the start activity
- execution of *Receive order*
- execution of *Check order* with *approval* set to `false`

It is notable that the lower branch will be selected after completion of *Process order* due to the data-based branching conditions.

Depending on the particular process description language, the current execution state of the process instance can be represented in different ways. WSM nets, for example, introduce execution state markings for process nodes and edges indicating whether or not they have already been processed (cf. Fig. 5.3 and Example 5.2).

Generally, a process instance can be associated with an execution history capturing all activities executed so far. By “replaying” the execution history (also referred to as *execution trace*)

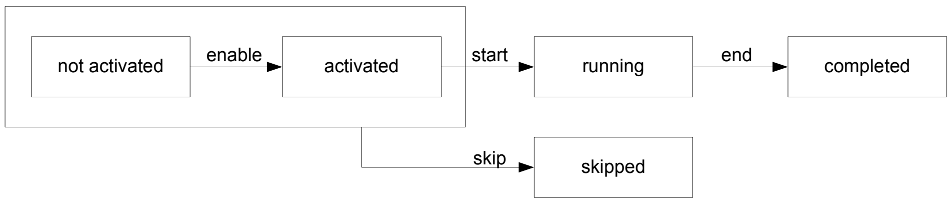


Figure 5.3.: A subset of the execution states of activities of WSM nets [Rei00]

over the associated process model, the current execution state of the process instance can be reconstructed, provided that all data values and decisions with respect to gateways are also recorded in the execution history.

5.1.3. Implications for the compliance checking framework

In literature, a few approaches were proposed that define compliance constraints directly on the process structure (e.g., fork constraints [SOS05]). However, as illustrated in Fig. 5.4, our observations from literature and the requirements analysis indicate that compliance rules are rather defined from the linear execution perspective [DAC99, Nam08, AP06, YMHJ06, LMX07]. In particular, compliance rules typically do not directly constrain the process structure but rather impose constraints on the process behavior (i.e., on the particular process executions). Then, a process model or a process instance does not comply with an imposed rule if it enables a process execution violating the compliance rule.

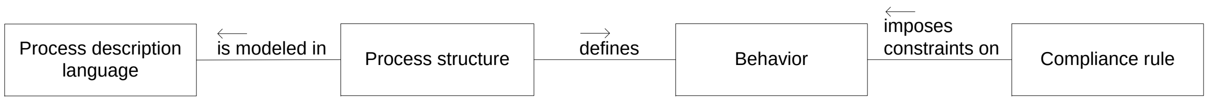


Figure 5.4.: Relations between process description languages, process structures, behavior, and compliance rules

As also illustrated in Fig. 5.4, the process structure may be modeled in different process description languages. Due to the multitude of existing process description languages, key to the practical application of a compliance checking framework will be its ability to operate with a broad variety of languages, such as BPEL or BPMN, that are used in practice. In order to achieve language-wise independence, it becomes necessary to abstract from particular process description languages and to introduce a more general layer as fundament of the compliance checking framework. This layer has to be designed such that a multitude of process description languages can be mapped to it while still capturing all information relevant to the verification with respect to compliance with imposed rules. In the following, we introduce a logical model that will enable the language-independent representation of process behavior and, thus, can serve as interface between the compliance rule and the process perspective.

5.2. Logical model

The rationale behind employing a logical model as interface between processes and compliance rules is twofold. Firstly, the logical model shall enable compliance rule specification on the process behavior without having to consider the specifics of the process structure. Secondly, it enables us to abstract from particular process description languages and constitutes a sound footing for the formal and operational semantics of compliance rules. For that purpose, the logical model utilizes *events*, *execution traces*, and *process event graphs*. Fig. 5.5 illustrates the concepts of the logical model and their relations.

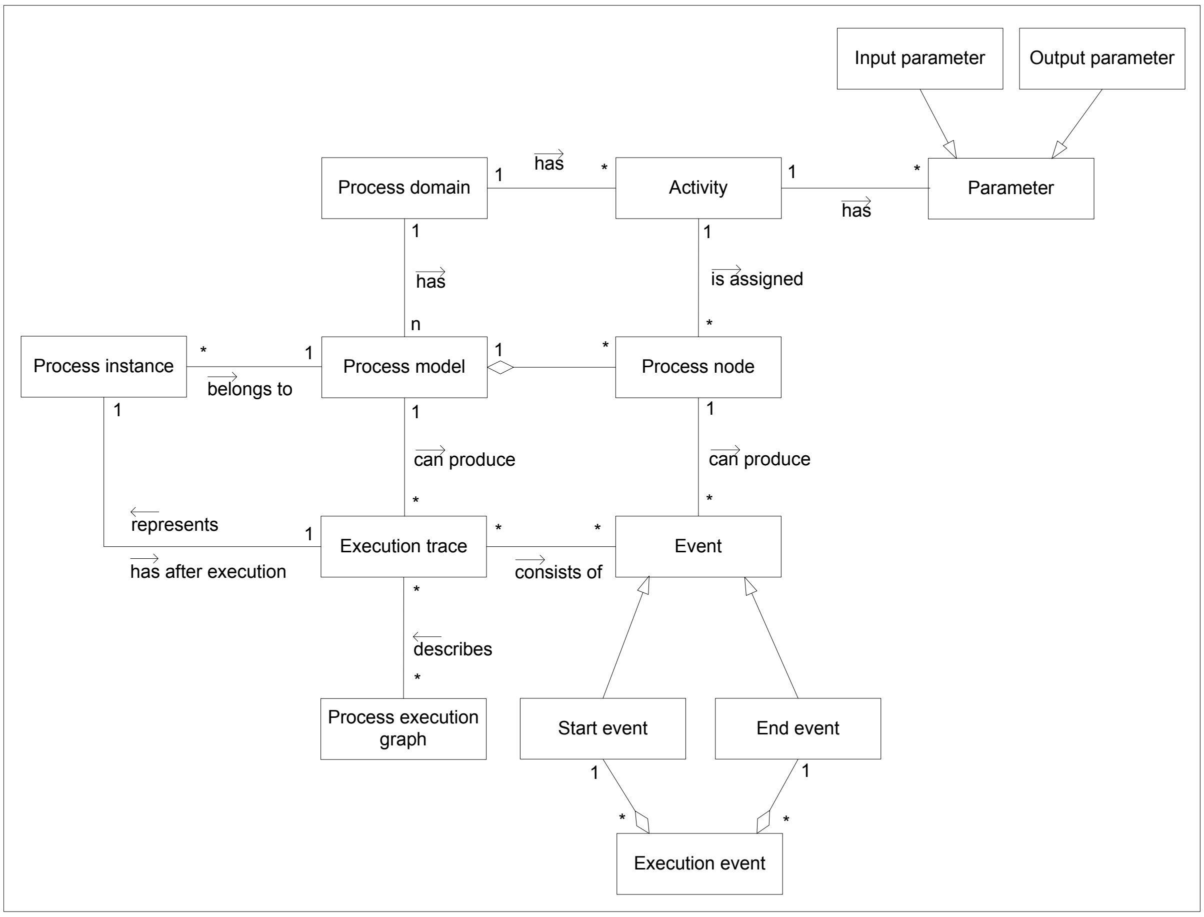


Figure 5.5.: Relations between process models, process nodes, activities, and events

Events represent the execution of process artifacts. An *execution trace* is a sequel of events and corresponds to the execution history of a process instance. Therefore, an execution trace can be used to represent a process execution (i.e., the execution of a particular process instance). Execution traces are a known concept in the business process management area. They are used for example to define the equivalence of business processes [MW06], to formalize criteria for process schema evolution [Rin04], or as input for process mining [LRDR06, ABD05, AHW^{+}11]. Typically, process models do not only enable a single process execution. For example, due to exclusive gateways and process instance-specific process data allocation, a process model may enable a multitude of different executions. Hence, a process model is usually able to generate a whole set of different execution traces. To capture the behavior encoded by a process model, the logical model provides *process event graphs*, a data structure based on finite automatons. The logical model is designed such that it reflects all properties relevant to the compliance rules addressed in this thesis. However, the concepts can be extended to provide for further aspects if required by the compliance rules.

In Section 5.2.1, we introduce the process domain and process activities before basic assumptions with respect to process models are introduced in Section 5.2.2. Section 5.2.3 then introduces events in more detail before formalizing the notion of execution traces in Section 5.2.4. Based on that, we provide a general notion of process instances in Section 5.2.5. Finally, Section 5.2.6 introduces process event graphs, a data structure for representing the behavior encoded by process models and process instances based on finite automatons.

5.2.1. Process domain and activities

In the remainder of this thesis, we assume a defined *process domain* with a given set of *activities* from which process models are composed. We focus on activities as process artifacts. However, our concepts can be applied to other process artifacts, such as events as known from BPMN [OMG11], as well. In the following, we will use T to denote the set of activities of the process domain.

An activity constitutes a business function of the process domain, such as confirming or checking an order in an order-to-delivery scenario (cf. Section 5.1.1). To capture the data perspective of processes, we assume that activities possess data parameters. In an order-to-delivery scenario, for example, the result of the approval decision (e.g., `approved`) is a likely data parameter of the activity *Check order* (cf. Fig. 5.1 in Section 5.1.1). Following a common paradigm among process description languages, we distinguish between *input* and *output parameters.* Input parameters typically provide an activity with data context necessary to carry out the business function, while output data typically capture the data context set by the business function. In the following, we denote as $Input_{at}$ / $Output_{at}$ the input / output parameters of an activity at, respectively. We assume that these parameters capture all data that can be subject to compliance rules concerning the corresponding activity. For example, the actor who executed an activity can be reflected in an output parameter named *agent*. We further assume that the domains of the input and output parameters are finite. In consequence, the set of possible executions of a particular activity is finite. In case of infinite domains, abstraction strategies such as proposed in [KLRM$^+$10] can be applied to yield finite domains of interest (cf. Chapter 9).

5.2.2. Process models

For the sake of the generality of the logical model, we only make minimal assumptions with regard to process models. Basically, process models consist of process nodes, which, in turn, are assigned process activities from the process domain. Instead of operating directly on the process models, we will later introduce *process event graphs* (`PEG`s) (cf. Section 5.2.6) that will serve as notation-independent representation of the behavior encoded by process models. Therefore, we abstain from formally defining process models.

5.2.3. Events

An execution trace reflects the activities conducted within a process execution. The events in an execution trace, therefore, capture the behavior of process nodes when being executed. As we address compliance rules on the occurrence, absence, and ordering of activity executions, we focus on events attesting activity executions. We distinguish between three event types. START and END events represent the start and completion of an activity, respectively. Sometimes, it will be more convenient to use one event to represent the execution of an activity instead of using a pair of START and END events. For that reason, we introduce EX events, each of which corresponds to the aggreation of a START and a corresponding END event.

In addition to the event types, events should have properties reflecting the source of the event in the process model or the process instance and associated context information (e.g., data and user assignment) [VBDA10]. For that reason, we assume that an event is associated with a process node[2]. In order to also reflect the data context of a process execution, events may be assigned data properties. In this thesis, we assume that the data properties refer to the parameters of the activity associated with the event (via the event's node). Conceivably, one can also think of further properties of events. In this work, we assume that all relevant properties are related to the activity associated with the event[3]. Equipped with these properties, events and, thus, execution traces enable the definition of the compliance rules addressed in our work (cf. Section 1.1). In fact, the event notion introduced in this thesis corresponds to XES [VBDA10], an event format advocated by the process mining community.

As process nodes are associated with activities, the execution of a process node produces a START and a corresponding END event. Due to the parameter signature of the associated activity, a process node is capable of producing a multitude of concrete events as shown in Example 5.3. Based on the above considerations, Def. 5.1 formalizes the notion of events.

Example 5.3 (Events):
Consider node 3 of process model P_1 in Fig. 5.1. Node 3 is associated with the activity *Check order*, which, in turn, outputs the parameter *approval*. Thus, in any execution of P_1, the activity *Check order* becomes executed. Assuming that *approval* is a boolean parameter, there can be one START event and two different END events of node 3 (agent assignment neglected) as illustrated in Fig. 5.6. In the event notation employed, the first argument denotes the event type, the second argument the source of the event in the process (i.e., the node), and the third argument the activity associated with the event. The fourth argument records the data allocations. Note that the fourth argument of the event signature will be left out if the event has no relevant data assignment (e.g., for e_1).

[2]The rationale behind an event being associated with a particular process node in addition to an activity is to enable compliance rules to refer to specific process nodes (i.e., local compliance rules). This can, for example, be applied to "hook" a compliance rule into a process model (e.g., a compliance rule that becomes activated between two predefined process nodes [RMM11]). As an activity can occur multiple times within a process model / process instance, the granularity of activities may not be sufficient in such cases.

[3]However, our concepts are also applicable to events associated with further properties.

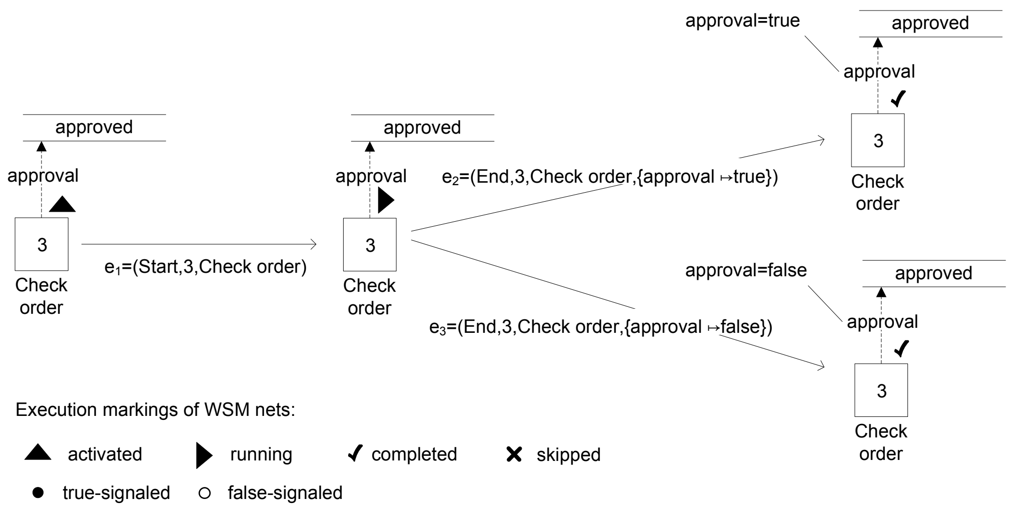


Figure 5.6.: Events occurring during the execution of node 3 of process model P_1 (cf. Fig. 5.1)

Definition 5.1 (Event)
Let P be a process model with N_P being the nodes of P. Then, an event e over P is a 4-tuple $e = (type_e, n_e, at_e, data_e)$ with

- $type_e \in \{\text{START}, \text{END}, \text{EX}\}$ denoting the type of e,
- $n_e \in N_P$ is a node of P associated with e,
- at_e denotes the activity associated with n_e, and
- $data_e$ is a function assigning a value of the corresponding domain to each input ($Input_{at_e}$) / output ($Output_{at_e}$) / input and output parameter of the activity at_e for event e of type START/ END/ EX, respectively.

We denote as E_P^{Start}, E_P^{End}, and E_P^{Ex} the set of all START, END, and EX events, as E_P^* the set of all events over P, as E^{Start}, E^{End}, and E^{Ex} as the set of all START, END, and EX events, and as E^* the set of all events over the process domain, respectively.

Note that EX events "combine" a START and a corresponding END event. Therefore, they are associated with data assignments for both input ($Input_{at}$) and output ($Output_{at}$) parameters as described in Def. 5.1.

Assumptions In order to be able to replay execution traces in a process model, for END events associated with a split gateway node, we assume that the data function from Def. 5.1 also records which outgoing branch is selected after processing the node. This can, for example, be done using a designated parameter for gateway decisions (e.g., *dec*) whose domain is the set of the outgoing edges of the respective node.

Notation Following the notation used in Fig. 5.6 and Example 5.3, we will omit the data function in illustrations if no assignments are made for relevant data parameters. If the activity associated with an event is not relevant (e.g., for illustration), we will omit the activity in the event notation. For illustration purposes, we will simplify the notation of events when only the activity type is relevant (e.g., event $= A$).

As mentioned, sometimes it will be more convenient to deal with EX events attesting the atomic execution of an activity instead of separate START and END events. Therefore, we introduce an auxiliary function to aggregate a pair of associated START and END events to an EX event in Def. 5.2.

Definition 5.2 (Aggregation of START and END events to EX events)
Let P be a process model and e_s and e_e be two events of P, where $e_s = (\text{START}, n, at, data_s)$, $e_e = (\text{END}, n, at, data_e)$ with at being the activity type of n and $Input_{at}$ / $Output_{at}$ being the input / output parameters of at. Then:

- $actExEvent(e_s, e_e) := (\text{EX}, n, at, data)$ where $\forall p \in Input_{at} \cup Output_{at}$:

$$data(p) := \begin{cases} data_s(p), & if\ p \in Input_{at} \\ data_e(p), & otherwise\ (i.e., p \in Output_{at}). \end{cases}$$

For an EX event $e = (\text{EX}, n, at, data)$

- $startEvent(e) := (\text{START}, n, at, data_s)$ returns the START event corresponding to e with $\forall p \in Input_{at} : data_s(p) := data(p)$ and
- $endEvent(e) := (\text{END}, n, at, data_e)$ returns the END event corresponding to e with $\forall p \in Output_{at} : data_e(p) := data(p)$.

Note that Def. 5.2 does not check whether a pair of START and END events really do belong together for the aggregation to an EX event. This has to be ensured when applying the aggregation. Example 5.4 illustrates the application of the aggregation.

Example 5.4 (Aggregation of START and END events to EX events):
Consider again the events depicted in Fig. 5.6. Then, the aggregation of e_1 and e_2 is as follows: $actExEvent(e_1, e_2) = (\text{EX}, 3, Check\ order, \{approval \mapsto \texttt{false}\})$.

5.2.4. Execution traces

Generally, an *execution trace* is a finite ordered sequence of events[4]. Our execution trace notion corresponds to the ones from literature as, for example, used in process mining research [VBDA10, AHW+11, Pes08]. As we use execution traces to capture the behavior of process models and process instances, we are particularly interested in execution traces that can be produced by process models. Therefore, we introduce the notion of execution trace with respect to a process model in Def. 5.3. Considering how activities within processes are executed, a resulting execution trace typically consists of START and END events.

Definition 5.3 (Execution trace)
Let P be a process model and σ be a finite ordered sequence of events of type START and END with: $\sigma = <e_1, \ldots, e_n>$ with $e_i \in E_P^{Start} \cup E_P^{End}, i = 1, \ldots, n$.

Then, σ is considered an *execution trace of* P iff σ can be produced by P by applying the operational semantics of the corresponding process description language.

For σ, we define

- $\sigma[i] := e_i$ as the ith element of σ,
- $\sigma \backslash \sigma[i]$ as σ with the ith element being removed, and
- $\sigma|\sigma'$ as the concatenation of σ with another execution trace σ'.

Sometimes, it will be more convenient to use a more compact representation of an execution trace, in which START and END events are aggregated to EX events. In Def. 5.4, we, therefore, introduce execution traces containing EX events.

Definition 5.4 (Execution traces containing EX events)
Let P be a process model and σ be a finite ordered sequence of events of type START, END, and EX with: $\sigma = <e_1, \ldots, e_n>$ with $e_i \in E_P^*, i = 1, \ldots, n$. Let further σ' be the event sequence obtained when replacing each event $e = (\text{EX}, n, at, data)$ in σ with the sequence $<startEvent(e), endEvent(e)>$, respectively.

Then, σ is considered *an execution trace of* P iff σ' is an execution trace of P.

5.2.5. Process instances

At runtime, process instances are typically created and executed based on a process model. The current execution state of a process instance can be yielded by "replaying" its execution

[4]In theory, as process models can contain loops, a process instance may be executed without ever terminating (non-terminating loops). However, for practical scenarios, fairness assumptions can be made. That is why we assume that finite traces are sufficient to describe the behavior of a process execution.

history in the process model. This is sometimes referred to as the *token game* (in the context of process models denoted as Petri nets) [RA08]. The execution history of a process instance can be captured by an execution trace (cf. Section 5.2.4). Therefore, altogether, a process instance can be represented by a combination of its process model and an execution trace to represent its execution history [Rin04]. This intuition is adopted in Def. 5.5.

Definition 5.5 (Process instance)
Let P be a process model. Then, a process instance I of P can be represented through a tuple (σ, P) where $\sigma = <e_1, \ldots, e_k>$ with $e_i \in E_P^*, i = 1, \ldots, k$, is an execution trace representing the execution history of I.

Example 5.5 (Process instance):
Consider again process instance I_1 from Fig. 5.2. Then, I_1 can be represented through the tuple (P_1, σ) with P_1 being the process model from Fig. 5.1 and the execution history σ being defined as follows:
$<(\textsc{start}, 1), (\textsc{end}, 1), (\textsc{start}, 2), (\textsc{end}, 2), (\textsc{start}, 3), (\textsc{end}, 3, \{\mathit{approval} \mapsto \texttt{false}\})>$.

5.2.6. Process event graphs

For compliance verification, process models and process instances have to be explored (i.e., unfolded) in order to check whether they enable behavior violating compliance rules to be checked. A process model typically enables multiple different process executions and, thus, can generate multiple different execution traces. Process instances whose execution is not yet completed often also enable multiple different possible futures, depending on the options encoded in the corresponding process model. To represent the behavior encoded by process models and process instances independently from the process description languages employed, we utilize a structure that is based on finite state automatons where each node is associated with an event[5]. Thus, a path from a start to an end node constitutes an execution trace. We refer to these structures as *process event graphs* (`PEG` for short). The structure of a `PEG` is introduced in Def. 5.6. Note that Def. 5.6 does not imply how `PEG`s are derived from a process model but rather defines a general data structure based on which we will illustrate our compliance checking concepts. As illustrated in Fig. 5.5, a `PEG` basically describes a set of execution traces and can represent an unfolded process model.

[5]Similar structures are used by a multitude of approaches to capture process behavior (e.g., [LMX07, KLRM$^+$10]). Specifically, the structure employed in this thesis resembles a Kripke structure [Kri63] that is not necessarily associated with infinite paths. As we focus on the compliance checking mechanisms rather than on the representation of behavior encoded in process models, we opted for a simple representation that is sufficient for illustrating the concepts presented in this thesis.

Definition 5.6 (Process event graph)
Let P be a process model. A process event graph (PEG) X over P is a 5-tuple $X = (S, s_0, S_E, T, el)$ where:

- S is a set of nodes,
- $s_0 \in S$ is a start node,
- $S_E \subset S$ is a set of end nodes,
- $T \subset S \times S$ is a set of precedence relations, and
- $el : S \rightarrow E_P^{Start} \cup E_P^{End} \cup E_P^{Ex}$ is a function assigning an event of type START, END, or EX to each node of X.

We assume that each node of S is reachable via the precedence relations.

Note that Def. 5.6 allows for only a single start node for PEGs. This is not an actual restriction as a virtual start node can be inserted if a process has multiple start nodes.

In Chapter 8, PEGs will be used to illustrate the application of the compliance checking framework based on the compliance rule graphs and their operational semantics proposed in this thesis. However, it should be noted that the proposed compliance checking framework is not restricted to the PEG data structure introduced here. In fact, our approach is applicable to data structures reflecting the execution traces of a process to be verified. In addition, it is not necessary to explicitly generate a PEG from a process model in order to verify compliance but compliance may also be checked on-the-fly while exploring the model. This will be addressed in Chapter 8.

Clearly, Def. 5.6 only formalizes the basic structure to capture the behavior of process models and process instances. In Def. 5.6, we did not make restrictions with respect to the *equivalence* of a process model and a PEG. Some fundamental equivalence notions, however, become necessary when PEGs are used as input to verify the compliance with imposed rules for process models and process instances. Def. 5.7 provides a very fundamental notion of equivalence between PEGs and process models / process instances based on trace-equivalence. Note that the set of execution traces described by a process instance is a subset of execution traces described by the corresponding process model. In particular, the subset is constituted by those traces starting with exactly the execution history of the process instance.

Definition 5.7 (Equivalence between PEGs and process models / process instances)
Let P be a process model and X be a PEG over P. Then, X is considered equivalent to P iff P and X describe the same set of execution traces.

Let further $I = (\sigma, P)$ be a process instance of P. Then, X is considered equivalent to I iff I and X describe the same set of execution traces.

Note that it may become necessary to transform EX events associated with PEG nodes into pairs of START and corresponding END events following Def. 5.2 in order to apply the equivalence notion provided in Def. 5.7. As the equivalence notion based on traces is quite intuitive, we abstain from providing a more formal notion.

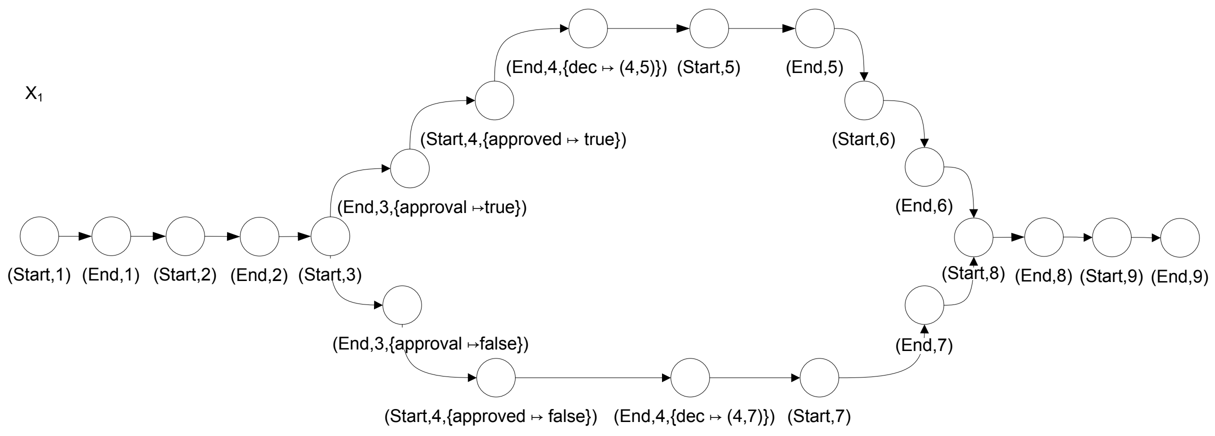


Figure 5.7.: A PEG that is considered equivalent to process model P_1 from Fig. 5.1

Example 5.6 (Process event graph):
Fig. 5.7 depicts a PEG that is equivalent to process model P_1 from Fig. 5.1 as it describes the same set of execution traces as can be produced from P_1.

We do not imply how PEGs are derived from a process model. A process model can, for example, be transformed into a PEG (e.g., by applying its operational semantics), which, in turn, serves as input for compliance verification. It is notable, however, that equivalence between the process model (or process instance) and the PEG to be verified as specified by Def. 5.7 is not necessarily required for compliance verification. In order to verify the compliance of processes with specific compliance rules, PEGs can be used that do not fully capture the complete but only the relevant process behavior. For the sake of verification efficiency, it clearly makes sense to use compact PEGs that capture only the process behavior relevant to the compliance rules to be checked. To support this, it is desirable to relax the equivalence notion specified in Def. 5.7. Such considerations to optimize the compliance verification do not affect the compliance checking framework introduced in this thesis as they have impact only on the size of the input. In Chapter 9, we will discuss such abstraction strategies to yield more compact PEGs in more detail. For now, we assume given PEGs that are equivalent to the processes to be verified.

6 Compliance rule graph fundamentals

Premise to automatic compliance checking is the specification of the compliance rules to be checked. Providing the means to transform compliance requirements into checkable properties, the compliance rule language constitutes an important building block of a compliance checking framework. As discussed in Section 2.1, a compliance rule language has to be formal and sufficiently expressive to capture compliance requirements while still remain easy to use in order to not become an obstacle to compliance automation. In this chapter, we introduce the *compliance rule graph* language for modeling declarative compliance rules developed based on the requirements discussed in Section 2.1. For the sake of ease-of-use, formal details are hidden from the modeler. This is achieved by providing a visual notation that is based on graphs, a metaphor well-known from process modeling. The compliance rule graph language is equipped with declarative formal semantics, which enables the formal analysis of modeled compliance rules. In Chapter 7, we will provide operational semantics for compliance rule graphs that will enable to execute them over execution traces in order to verify compliance at process design and runtime.

In the following, the goals of the compliance rule graph language are described in Section 6.1. Then, the modeling primitives of compliance rule graphs, their syntax, and informal semantics are introduced in Section 6.2. Section 6.3 addresses the formal semantics of compliance rule graphs by mapping them to rule formulas that can be formally interpreted over execution traces. Finally, Section 6.4 discusses alternatives to compliance rule graphs, describes ideas on extending compliance rule graphs in different respects such as support of time constraints, and provides ideas to round up our approach with respect to comprehensive compliance support.

6.1. Introduction

With respect to the requirements and the vision of an overall compliance checking framework introduced in Chapter 2, we compiled goals that guided the development of the compliance rule graph language. These goals are described in the following in Section 6.1.1. Then, Section 6.1.2 briefly describes the basic ideas of compliance rule graphs.

6.1.1. Requirements and goals

In Section 2.1.1, we elaborated on requirements on the compliance rule language. As identified in Section 2.1.1.2, the compliance rule language must have clearly defined formal semantics. However, for ease-of-use, particularly for intelligibility of modeled rules, it is desirable to hide formal details of the language from the user. That is why we envisioned a graphical modeling language inspired by business process modeling.

As discussed in Section 2.1.1.1, the property specification patterns suggested by Dwyer and Corbett and applied by a multitude of related approaches must be supported by the compliance rule language as they have been proven to be of practical relevance. However, the limitations of pattern-based approaches should be avoided. In [LRMD10], we showed that the property specification patterns still lack expressiveness for certain types of compliance rules (cf. Section 2.1.1.1). Therefore, a goal of our work is to provide a truly compositional language that enables composing rules using fine-grained modeling primitives, thus, giving the flexibility to go beyond predefined patterns. In this context, it is vital that the language is extensible in order to incorporate further aspects not yet addressed in this thesis (such as time constraints).

In Section 2.1.1.4, we described important requirements with respect to the organization and management of compliance rules. They are not directly tied to a specific compliance rule language and are not within the scope of this thesis. However, it must be possible to integrate existing solutions for these requirements with the compliance rule language.

The primary goal of compliance rule specification is to enable automated compliance checking. Hence, the compliance rule language should lay the fundament for compliance checking mechanisms providing compliance diagnoses that meet the requirements with respect to granularity and comprehensiveness (cf. Section 2.1.2).

6.1.2. The compliance rule graph approach

Based on the goals described in Section 6.1.1, we developed the compliance rule graph approach proposed in this thesis. The components of the approach are depicted in Fig. 6.1.

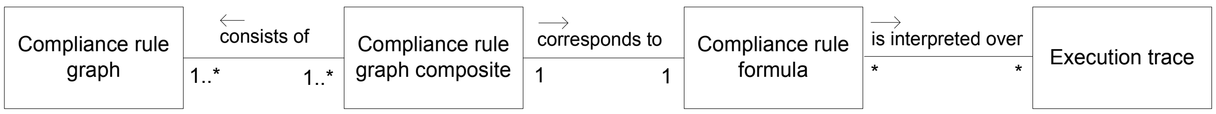


Figure 6.1.: Relations between execution traces, compliance rule formulas, compliance rule graph composites, and compliance rule graphs modeled in UML class diagram notation

The compliance rule graph (CRG) language constitutes the fundamental building block of our approach. CRGs are a graphical representation for compliance rules and can be composed from modeling primitives with a notation based on nodes and edges. The graph structure of CRGs constitutes the basis for operationalization, which enables compliance checks providing diagnoses directly based on the rule structure. The operationalization of CRGs further enables to verify compliance of process models and running process instances using the same mechanisms. This ensures integrated support for the process lifecycle as envisioned in Section 2.2.1. CRG composites are represented by a set of CRGs that form a compliance rule. Each CRG composite, in turn, can be transformed into a compliance rule formula (RF), a formula specified in first-order predicate logic. The formal semantics of RFs is defined over execution traces, which enables the formal interpretation of CRGs and CRG composites. After this brief overview, Section 6.2 now introduces the modeling primitives, syntax, and informal semantics of CRGs.

6.2. Structure of compliance rule graphs

As clarified in Section 1.1, we focus on compliance rules imposing constraints on the occurrence, absence, and ordering of activities that are carried out during a process execution. Typically, a compliance rule implies a certain behavior if certain conditions are met. In the context of business process activities, a compliance rule implies some event patterns (with respect to the activities) if some other event patterns occur [YMHJ06, SGN07, CMMS07a, FESS07, ADW08, AWW09]. This intuition is adopted for CRGs. A CRG captures a compliance rule and is specified by explicitly modeling

- a pattern of activity executions whose occurrence in a process execution activates the compliance rule (e.g., necessitates other activities) and
- a pattern of activity executions, which has to apply in the process execution when the rule becomes activated (rule consequence).

While a CRG only captures a single consequence pattern, multiple disjunctive rule consequences are enabled by CRG composites as we will later describe in Section 6.2.6. The patterns associated with a CRG's antecedent and consequence are composed from modeling primitives of the CRG language, which are introduced in Section 6.2.1. The informal semantics of CRGs is described in Section 6.2.2 before CRGs are formalized in Section 6.2.3.

6.2.1. Modeling primitives

Based on the assumption adopted from graph-based process description languages that a graph is a suitable representation for expressing occurrence and ordering relations of activities, the patterns associated with the rule antecedent and the rule consequence are modeled by means of graphs. In order to distinguish between the antecedent and the consequence in the graph representation, they are modeled using designated node types (condition and consequence node types). Thus, despite the graphical notation, the underlying explicit rule structure is rigorously enforced in the CRG approach. This can facilitate the understanding of modeled compliance rules. Altogether, from their looks, CRGs are graphs with different node types with a graph fragment describing the rule antecedent pattern and a graph fragment describing the rule consequence pattern.

The individual patterns of activity executions, in turn, are modeled using nodes representing the occurrence or absence of activity executions of certain properties and edges that constrain their ordering as well as whether two nodes can be associated with the same activity execution. Based on their formal semantics, which will be introduced in Section 6.3, CRGs can be tested over an execution trace in a pattern matching manner by testing whether CRG nodes can be matched with activity executions contained in the execution trace.

In the following, we describe the modeling primitives and their semantics in more detail. Section 6.2.1.1 describes the CRG node types and Section 6.2.1.2 describes the properties of CRG nodes. The relations between CRG nodes are detailed in Section 6.2.1.3.

6.2.1.1. Compliance rule graph node types

As mentioned, we distinguish between occurrence and absence nodes. While the former represent the occurrence of a particular activity execution, absence nodes represent the absence of particular activity executions. To distinguish between nodes that are used to model a compliance rule's antecedent pattern and nodes that are used to model a compliance rule's consequence pattern, we further differentiate between antecedent and consequence nodes. Altogether, this results in four different CRG node types: AnteOcc, AnteAbs, ConsOcc, and ConsAbs nodes. The graphical notation of CRG nodes is depicted in Fig. 6.2.

- An AnteOcc node represents the occurrence of an activity execution as part of the pattern activating the compliance rule.
- An AnteAbs node represents the absence of activity executions that as part of the pattern activating the compliance rule.
- A ConsOcc node represents the occurrence of an activity execution requested by a consequence pattern of the compliance rule.
- A ConsAbs node represents the absence of activity executions requested by a consequence pattern of the compliance rule.

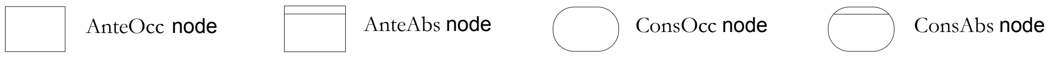


Figure 6.2.: Notation for CRG node types

Note that a CRG node corresponds to an activity execution (instead of a single START or a single END event). Clearly, it would also be possible to define constraints directly on START and END events of activity executions. However, we opted to provide semantically more high-level support for defining compliance rules. This is the rationale behind defining constraints using the granularity of activity executions. We would like to stress, however, that CRG nodes may be also be associated with more fine-grained events.

6.2.1.2. Compliance rule graph node properties

CRG nodes, regardless of their type, can be assigned conditions on properties of activity executions. These are used to filter the activity executions that are relevant to the antecedent or the consequence pattern of a CRG. A CRG node will *match* with an activity execution of an execution trace if all conditions apply[1]. Thus, the patterns modeled in CRGs can be tested over execution traces in a pattern matching manner by pairing CRG nodes and matching activity executions. Against the background of the information exhibited by events in an execution trace as defined in Section 5.2.3, a CRG node n can be assigned the following conditions:

- An **activity** from the process domain (denoted by at) to express that only activity executions of that type can match with n.
- A **process node identifier** to refer to specific nodes of an existing process model. Thus, it is possible to virtually "plug" a process-specific compliance rule into an existing process model[2]. The node identifier will be denoted by nid.
- A set of **data conditions** referring to data parameters of the associated activity (e.g., of the form $(p \odot v)$ where p is a parameter, $\odot$ is a comparison operator, and v is a value from p's domain). The assigned data conditions are considered a conjunction. Thus, in order for an activity execution to match with a CRG node, all data conditions have to apply[3].

We assume that each CRG node is assigned exactly one activity. The other properties are optional. Beyond the above conditions, we can think of further conditions associated with CRG nodes, for example, time conditions such as deadlines or conditions on the duration of activity executions. Prerequisite to the assignment of further conditions is that the logical model (cf. Section 5.2) provides the suitable fundament (e.g., a notion of time for execution traces). In this work, we confine our considerations to the information provided by the events as defined in Section 5.2.3. Considerations on extending the concepts are described in Section 6.4.2. The notation for CRG node properties is shown in Fig. 6.3.

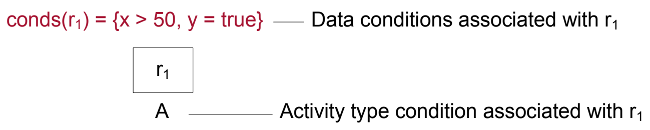

conds(r₁) = {x > 50, y = true}
Data conditions associated with r₁
r₁
A
Activity type condition associated with r₁

Figure 6.3.: Notation for node properties

[1]The matching of CRG nodes and activity executions will be detailed in Chapter 7 (cf. Section 7.3.1.1).

[2]For example, it becomes possible to specify rules requesting that a certain activity is not allowed between two particular process nodes (e.g., two milestones of the process). This enables to "hook" a compliance rule into a process model [RMM11].

[3]For our concepts, the particular form of the data conditions is not important as long as the data conditions can be evaluated based on the data assignment of the parameters of the associated activity. Therefore, we do not go into details on the particular structure of data conditions. In compliance rules from practical applications, we often find disjunctive data conditions (e.g., "if a patient has a confirmed allergy or is taking drugs that might cause interactions, then ..."). Supporting such disjunctive data conditions is possible in our framework. Even more complex data conditions (e.g., nested conditions) can be supported, a service for matching conditions with events provided.

Example 6.1 (CRG nodes):
Fig. 6.3 depicts AnteOcc node r_1, which is associated with different conditions (i.e., $at = A$, $x > 50$, $y = \texttt{true}$). Now consider, for example, the following execution trace:
$<e_1 = (\textsc{start}, 1, A, \{x \mapsto 70\}), e_2 = (\textsc{end}, 1, A, \{y \mapsto \texttt{true}\}), e_3 = (\textsc{ex}, 2, A, \{x \mapsto 80, y \mapsto \texttt{true}\})>$.

Then, the activity execution described by e_1 and e_2 (i.e., the execution of process node 1) corresponds to the profile defined by AnteOcc node r_1. In addition, r_1 also matches with the activity execution represented by EX event e_3 (i.e., the execution of process node 2).

6.2.1.3. Compliance rule graph relations

So far, we described how CRG nodes can be used to identify and match with activity executions contained in an execution trace (represented through a pair of START and corresponding END event or a single EX event). In order to model complex patterns of activity executions in compliance rules, it must be possible to relate CRG nodes to each other. For that purpose, we introduce relations. Their graphical notation as edges is depicted in Fig. 6.4.

ORDER edge
DIFF edge

Figure 6.4.: Notation for CRG relations

Ordering relations Clearly, essential to the definition of a pattern of activity executions is the ordering of the latter. Therefore, we introduce ORDER relations to define the ordering of activity executions. An ORDER relation between two CRG nodes s and t expresses, that s and t will only match with two activity executions X and Y (provided that the individual conditions assigned to s and t apply) if the execution of X ends before the execution of Y starts. Thus, an ORDER relation corresponds to a precedence relation commonly known from process modeling languages such as ADEPT [Rei00] or BPMN [OMG11][4]. In the CRG notation, an ORDER relation is represented by a directed edge. Apparently, ORDER relations among activity executions are transitive (i.e., if (s,t) and (t,u) holds, (s,u) will also hold), irreflexive, and asymmetric.

Example 6.2 (Ordering relations):
Consider the CRG fragment depicted in Fig. 6.5 a). We have a match for this fragment in an execution trace, if i) two activity executions satisfying the conditions associated with the CRG nodes r_1 and r_2 are contained in the trace and ii) these activity executions are also ordered such that the activity matching with node r_1 is completed before the activity matching with node r_2 is started. Consider the following traces:
1) $<(\textsc{start}, 1, A), (\textsc{end}, 1, A), (\textsc{start}, 2, B), (\textsc{end}, 2, B)>$

[4]Note that further ordering relations, such as start-start relation, can be used in compliance rules modeled as CRGs when interpreting CRG nodes not as activity executions but as single events.

2) <(START, 2, B), (END, 2, B), (START, 1, A), (END, 1, A)>
Then, trace 1) contains a match for the CRG fragment shown in Fig. 6.5 a) while the trace 2) does not.

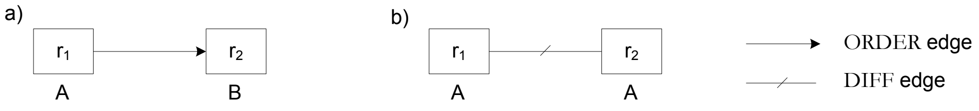


Figure 6.5.: Ordering and inequality relations

Inequality relations Sometimes, it can become necessary to ensure that two CRG nodes n_1, n_2 are not paired with the same activity execution in the pattern matching process despite the nodes having the same or overlapping node profiles (i.e., informally: activity execution paired with $n_1 \neq$ activity execution paired with n_2). For example, if one wants to express that two activity executions with the same attributes shall occur in a process execution (regardless of their ordering), two ConsOcc nodes with the same attributes can be used to model this pattern. However, consider the case that a single activity execution is contained in a trace. Then, the two ConsOcc nodes may match and be paired with the same activity execution, which is not intended. To prevent this, we introduce inequality relations, referred to as DIFF relations in the remainder of the thesis. A DIFF relation between two CRG nodes s and t expresses that they must not be paired with the same activity execution (i.e., activity execution of $s \neq$ activity execution of t). DIFF relations are non-transitive, irreflexive, and symmetric. Due to the symmetry, DIFF relations are represented via undirected edges in a CRG.

Example 6.3 (Inequality relation):
Consider the CRG fragment depicted in Fig. 6.5 b) and the following execution trace:
<(START, 1, A), (END, 1, A)>
Despite containing an activity execution matching with both nodes of the fragment, the trace does not exhibit a pattern that matches the CRG fragment. This is due to the DIFF relation requesting that there must be two different executions of A matching with the nodes of the CRG. This CRG fragment can, for example, be utilized to express the compliance rule requesting that if two lab tests are done within the same process, an invoice has to be created when releasing the patient as the patient will have to pay one of these lab test.

By combining CRG nodes with ORDER and DIFF relations, complex patterns of activity executions can be modeled. In particular, we can capture the property specification patterns [DAC99] and more complex compliance rules on the occurrence, absence, and ordering of activity executions [LRMD10] using these primitives. However, we can think of further relevant relations, for example, data relations (e.g., the 4-eyes-principle as in "the agent responsible for developing a component must not be the agent to test this component") or time relations (e.g., minimal or maximal distance constraints). A discussion on extending CRGs with further relations is provided in Section 6.4.2.

6.2.2. Informal semantics

As described, a CRG node, regardless of the node type, will only match with an activity execution of an execution trace, if all conditions associated with this node apply. For a pattern consisting of nodes and relations to match with a set of activity executions, the conditions imposed by relations associated with the node must also apply. The node type (i.e., whether a CRG node is an occurrence or an absence node), in turn, determines whether a matching activity execution is required or prohibited for the matching of the overall pattern. For occurrence nodes (i.e., AnteOcc and ConsOcc nodes), the respective CRG pattern only applies if matching activity executions can be found for all nodes. By contrast, the respective CRG pattern will not apply, if a matching activity execution can be found for at least one absence node (i.e., AnteAbs and ConsAbs nodes).

Example 6.4 (Modeling and evaluation of CRGs):
Fig. 6.6 depicts three CRGs where CRG R_1 is gradually refined to CRG R_2 and then to CRG R_3. Initially, the CRG visualized by R_1 is activated by the sequence $<A, B>$. In case of activation, it is required that activity C is executed between A and B and activity D is executed after B. In CRG R_2, we refined the pattern activating the compliance rule. In particular, R_2 is activated by the sequence $<A, B>$ only if activity E is not executed between A and B. The consequence CRG of R_1 is further refined in CRG R_3. Here, we included ConsAbs node r_6. As a result, R_3 expresses that there must be an execution of activity D after activity B, such that there is no execution of activity F between B and D.

As example, consider the following execution trace and R_3:
$<A_1, C_1, B_1, E_1, D_1, A_2, B_2, F_1, D_2>$

Then, Fig. 6.7 summarizes the matching of R_3 with the activity executions contained in the trace:

a) A_1 and B_1 activate R_3 as there is no execution of E between them. Moreover, for this activation of R_3, we can also find a match of the consequence pattern as C_1 occurs between A_1 and B_1 and D_1 occurs after B_1.

b) In contrast, A_1 and B_2 do not yield a match of the antecedent pattern due to E_1 occurring between them.

c) Finally, A_2 and B_2 yield a match of the antecedent pattern of R_3. However, no match for the consequence pattern can be found for A_2 and B_2 despite D_2 as F_1 occurs between B_2 and D_2. In addition, no match for r_3 can be found.

Altogether, R_3 becomes activated twice in the above execution trace. While the activation is satisfied in case a), the activation in case c) is not satisfied. Thus, R_3 is not enforced in the trace.

As Example 6.4 shows, a CRG's antecedent pattern may occur multiple times within an execution trace. Consequently, a CRG may become activated multiple times within a process model / process instance. We refer to this as *multiple rule activation* (cf. Section 2.1.2.1). Based on the

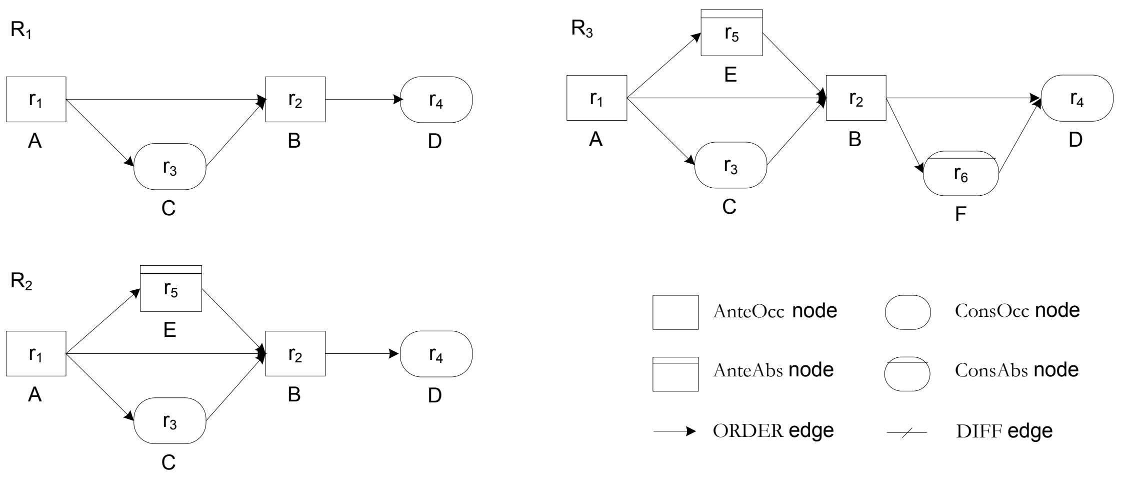


Figure 6.6.: Incremental modeling of a CRG

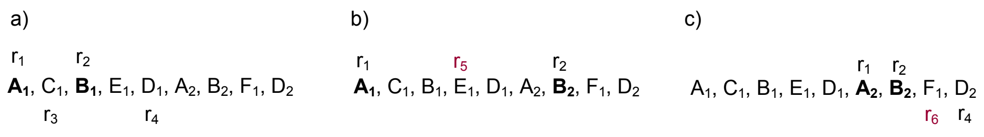

a)
r1 r2
A1, C1, B1, E1, D1, A2, B2, F1, D2
r3 r4
b)
r1 r5 r2
A1, C1, B1, E1, D1, A2, B2, F1, D2
c)
r1 r2
A1, C1, B1, E1, D1, A2, B2, F1, D2
r6 r4

Figure 6.7.: Matching CRGs with activity executions

manual pattern matching conducted in Example 6.4, we are already able to state whether or not a CRG becomes activated in an execution trace and whether these activations are satisfied (i.e., have a corresponding match of the consequence pattern).

The incremental composition of compliance rules as shown in Fig. 6.6 is a benefit of the CRG language. Specifically, it is possible to enrich a CRG with absence and occurrence constraints as shown in Fig. 6.6 without having to restructure the CRG. This is possible due to the pattern-oriented mental model of the CRG language (as opposed to the navigational mental model of LTL). Example 6.5 illustrates the semantics of absence nodes.

Example 6.5 (Absence nodes in CRGs):
Examples of CRGs containing ConsAbs nodes are provided in Fig. 6.8. While CRG R_4 expresses that there shall not be any execution of activity D that is both between A and B and between A and C at the same time, CRG R_5 expresses a slightly different constraint. In particular, R_5 expresses that there shall neither be any execution of D between A and B nor between A and C. Thus, the sequence $<A, B, D, C>$, for example, is compliant with CRG R_4 since D is executed between A and C but not between A and B. In contrast, this sequence is not compliant with R_5. This example shows that similar to occurrence nodes, an absence node will only match with an activity execution, if all conditions (e.g., ORDER relations) associated with that absence node apply.

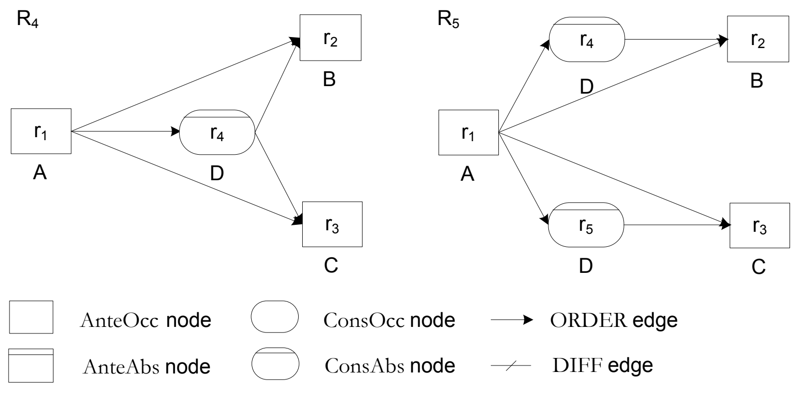


Figure 6.8.: CRGs containing absence nodes

6.2.3. Formalization

After introducing the structure and the primitives of CRGs, Def. 6.1 now formalizes the structure of CRGs. As illustrated by Example 6.4, a CRG consists of an antecedent and a consequence pattern.

Definition 6.1 (Compliance rule graph)
A compliance rule graph R is defined as $R := (A, C)$ where A denotes the subgraph representing the antecedent and C denotes the subgraph representing the consequence pattern of R with:

- $A := (N_A, OrderE_A, DiffE_A, nt_A, conds_A)$ where
  - N_A denotes the set of nodes of A,
  - $OrderE_A \subset N_A \times N_A$ is a set of directed edges representing ORDER relations between the nodes of A,
  - $DiffE_A \subset N_A \times N_A$ is a set of undirected edges representing DIFF relations between the nodes of A,
  - $nt_A : N_A \rightarrow \{\texttt{AnteOcc}, \texttt{AnteAbs}\}$ is a function assigning a node type to each node of A, and
  - $conds_A$ is a function assigning a set of conditions to each node of A.
- $C := (N_C, OrderE_C, DiffE_C, nt_C, conds_C)$ where
  - N_C denotes the set of nodes of C,
  - $OrderE_C \subset N_C \times N_C \cup N_A \times N_C \cup N_C \times N_A$ is a set of directed edges representing ORDER relations among nodes of N_C as well as between nodes of N_C and N_A,
  - $DiffE_C \subset N_C \times N_C \cup N_A \times N_C \cup N_C \times N_A$ is a set of undirected edges representing DIFF relations among nodes of N_C as well as between nodes of N_C and N_A,

- $nt_C : N_C \mapsto \{\texttt{ConsOcc}, \texttt{ConsAbs}\}$ is a function assigning a node type to each node of C, and
- $conds_C$ is a function assigning a set of conditions to each node of C.

We assume that for each node $n \in N_A \cup N_C$, $conds_A$ and $conds_C$ each returns at least a condition on the expected activity type of n, respectively. As discussed in Section 6.2.1.2, further conditions such as on the parameters of the activity are optional.

The antecedent pattern (captured by A) specifies the behavior that activates the compliance rule. In practice, we often encounter compliance rules that are always activated in a process (for example, to express that a security check is mandatory in each process without any specific pre-conditions). To model such compliance rules, the antecedent pattern can be left empty (i.e., the CRG consists of only `ConsOcc` or/and `ConsAbs` nodes).

Example 6.6 (Compliance rule graphs):
Consider again CRG R_3 depicted in Fig. 6.6. Then, R_3 is defined as $R_3 = (A, C)$ with

- $A = (N_A, OrderE_A, DiffE_A, nt_A, conds_A)$ where
  - $N_A = \{r_1, r_2, r_5\}$,
  - $OrderE_A = \{(r_1, r_2), (r_1, r_5), (r_5, r_2)\}$,
  - $DiffE_A = \emptyset$,
  - $nt_A = \{(r_1, \texttt{AnteOcc}), (r_2, \texttt{AnteOcc}), (r_5, \texttt{AnteAbs})\}$, and
  - $conds_A(r_1) = \{at = A\}$, $conds_A(r_2) = \{at = B\}$, $conds_A(r_5) = \{at = E\}$.
- $C = (N_C, OrderE_C, DiffE_C, nt_C, conds_C)$ where
  - $N_C = \{r_3, r_4, r_6\}$,
  - $OrderE_C = \{(r_6, r_4), (r_1, r_3), (r_3, r_2), (r_2, r_4), (r_2, r_6)\}$,
  - $DiffE_C = \emptyset$,
  - $nt_C = \{(r_3, \texttt{ConsOcc}), (r_4, \texttt{ConsOcc}), (r_6, \texttt{ConsAbs})\}$, and
  - $conds_C(r_3) = \{at = C\}$, $conds_C(r_4) = \{at = D\}$, $conds_C(r_6) = \{at = F\}$.

Fig. 6.9 summarizes the modeling primitives of CRGs. While the formalization in Def. 6.1 lays the fundament to model CRGs, not all structures following Def. 6.1 are considered *correct* CRGs. In Section 6.2.4, we will introduce constraints on the syntax of CRGs, for example, on the usage of ORDER and DIFF relations.

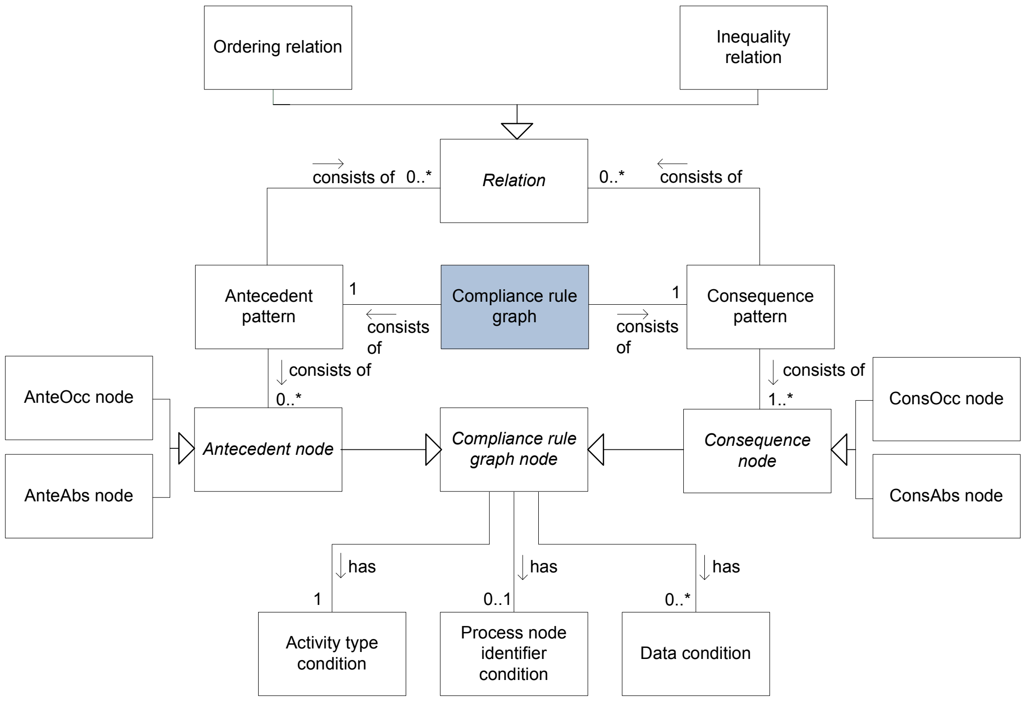


Figure 6.9.: Primitives of the CRG language

Auxiliaries To facilitate the handling of CRGs, we further introduce some auxiliary properties and functions that will enable us to refer to properties of a CRG in a convenient manner.

Definition 6.2 (Auxiliaries for compliance rule graphs)
Let $R = (A, C)$ be a CRG. Then, we define the following functions and properties over R:

- $N_R := N_A \cup N_C$ denotes the set of all nodes of R.
- $OrderE_R := OrderE_A \cup OrderE_C$ denotes the set of all ORDER edges of R.
- $DiffE_R := DiffE_A \cup DiffE_C$ denotes the set of all DIFF edges of R.
- $nt_R : N_R \rightarrow \{\texttt{AnteOcc}, \texttt{AnteAbs}, \texttt{ConsOcc}, \texttt{ConsAbs}\}$ is a function assigning a node type to each node of R with
$$nt_R(n) := \begin{cases} nt_A(n) & \text{for } n \in N_A \\ nt_C(n) & \text{for } n \in N_C. \end{cases}$$
- $conds_R$ is a function assigning a set of conditions (cf. Section 6.2.1.2) to each node of R with
$$conds_R(n) := \begin{cases} conds_A(n) & \text{for } n \in N_A \\ conds_C(n) & \text{for } n \in N_C. \end{cases}$$
- $startConds_R$ is a function returning the set of conditions associated with a node n that can be evaluated over a START event. These are conditions on the activity type of an execution, on input parameters of the respective activity, and on the process node.

- $endConds_R$ is a function returning the set of conditions associated with a node n that can be evaluated over an END event. These are conditions on the activity type of an execution, on output parameters of the respective activity, and on the process node.

We further define auxiliary functions for analyzing CRGs. These will enable us to easily refer to predecessor and successor CRG nodes.

Definition 6.3 (Auxiliaries for compliance rule graph analysis)
Let $R = (A, C)$ be a CRG. Then, we define auxiliary functions over R as follows:

- $succConsAbs : N_R \rightarrow 2^{N_R}$ is a function assigning the set of directly succeeding `ConsAbs` nodes to a given node with:
  $succConsAbs(n) := \{t \in N_R \mid nt_R(t) = \texttt{ConsAbs} \wedge \exists e = (n, t) \in OrderE_R\}$.
- $predAnteOcc : N_R \rightarrow 2^{N_R}$ is a function assigning the set of directly preceding `AnteOcc` nodes to a given node with:
  $predAnteOcc(n) := \{t \in N_R \mid nt_R(t) = \texttt{AnteOcc} \wedge \exists e = (t, n) \in OrderE_R\}$.
- $predAnteAbs : N_R \rightarrow 2^{N_R}$ is a function assigning the set of directly preceding `AnteAbs` nodes to a given node with:
  $predAnteAbs(n) := \{t \in N_R \mid nt_R(t) = \texttt{AnteAbs} \wedge \exists e = (t, n) \in OrderE_R\}$.
- $predConsOcc : N_R \rightarrow 2^{N_R}$ is a function assigning the set of directly preceding `ConsOcc` nodes to a given node with:
  $predConsOcc(n) := \{t \in N_R \mid nt_R(t) = \texttt{ConsOcc} \wedge \exists e = (t, n) \in OrderE_R\}$.
- $predConsAbs : N_R \rightarrow 2^{N_R}$ is a function assigning the set of directly preceding `ConsAbs` nodes to a given node with:
  $predConsAbs(n) := \{t \in N_R \mid nt_R(t) = \texttt{ConsAbs} \wedge \exists e = (t, n) \in OrderE_R\}$.
- $predConsAbs^* : N_R \rightarrow 2^{N_R}$ is a function assigning the set of directly and indirectly preceding `ConsAbs` nodes to a given node with:
  $predConsAbs^*(n) := \{t \in N_R \mid (t \in predConsAbs(n)) \vee (nt_R(t) = \texttt{ConsAbs} \wedge \exists t_1, \ldots, \exists t_k \in N_R \textit{ with } (t, t_1), (t_1, t_2), \ldots, (t_k, n) \in OrderE_R)\}$.
- $predConsOcc^* : N_R \rightarrow 2^{N_R}$ is a function assigning the set of directly and indirectly preceding `ConsOcc` nodes to a given node with:
  $predConsOcc^*(n) := \{t \in N_R \mid (t \in predConsOcc(n)) \vee (\exists t_1, \ldots, \exists t_k \in N_R, nt_R(t_i) = \texttt{ConsOcc}, i = 1, \ldots, k, \text{ with } t_1 \in predConsOcc(n), \ldots, t_k \in predConsOcc(t_{k-1}) \text{ and } t \in predConsOcc(t_k))\}$.

Example 6.7 demonstrates the usage of the auxiliaries from Def. 6.3.

Example 6.7 (Auxiliaries for compliance rule graph analysis):
Consider again CRG R_3 from Fig. 6.6. Then:

- $succConsAbs(r_2) = \{r_6\}$, $succConsAbs(r_1) = \emptyset$
- $predAnteOcc(r_2) = \{r_1\}$, $predAnteOcc(r_6) = predAnteOcc(r_4) = \{r_2\}$
- $predAnteAbs(r_2) = \{r_5\}$
- $predConsOcc(r_2) = \{r_3\}$
- $predConsAbs(r_2) = \emptyset$, $predConsAbs(r_4) = \{r_6\}$
- $predConsOcc^*(r_2) = \{r_3\}$, $predConsOcc^*(r_4) = \emptyset$

6.2.4. Syntactic correctness and conventions

As aforementioned, Def. 6.1 only specifies the basic structure of CRGs and still permits the specification of CRGs containing errors (i.e., structures such that a CRG cannot be interpreted properly or such that parts of the modeled patterns become unsatisfiable). This can be compared to errors in process models that should be prevented such as deadlocks. Such an error is described in Example 6.8. In order to prevent such cases, we introduce *syntactic correctness constraints* on CRGs.

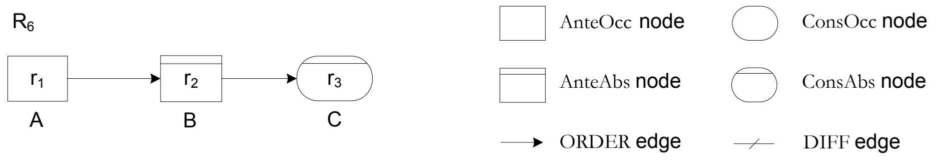


Figure 6.10.: A CRG containing errors

Example 6.8 (CRG containing errors):
Considering only the antecedent nodes, R_6 becomes activated for each execution of A without a subsequent execution of B. Once activated, the absence of C after B is requested by R_6. This, however, does not make sense as the absence of B after A is requested to activate R_6 in the first place. Generally, it does not make sense to relate AnteAbs nodes to consequence nodes at all as the former require the absence of certain activity executions for the rule to become activated. A CRG containing such structures cannot be interpreted properly. This can be prevented by introducing certain syntactic correctness constraints for CRGs.

In addition to constraints to prevent errors, we will further introduce *syntactic conventions* for CRGs to ensure certain properties (which will be useful for defining the operational semantics). In the following, syntactic correctness constraints are discussed in Section 6.2.4.1 and 6.2.4.2. Conventions for ensuring certain properties will be discussed in Section 6.2.4.3. These considerations are then compiled into a set of syntactic correctness constraints and conventions in Section 6.2.4.4.

6.2.4.1. Cyclic ordering relations

Due to the transitivity of ORDER relations (cf. Section 6.2), the cyclic definition of ORDER relations leads to patterns that cannot occur within an execution trace. As example, consider R_7 depicted in Fig. 6.11, which contains a cycle w.r.t. ORDER edges. R_7 expresses that for the activity execution A matching with r_1 after which an activity execution B matching with r_2 follows, an activity execution C matching with r_3 has to occur after B. At the same time, R_7 requests that C occurs before A. Clearly, this cannot be fulfilled. Similarly, R_8 contains a cycle w.r.t. ORDER relations among AnteOcc nodes. Again, this pattern cannot occur in any execution trace as an activity execution cannot be both predecessor and successor of another activity execution. Such errors can be compared to errors in process models such as deadlocks. To avoid the definition of such patterns, cyclic definitions of ORDER edges are prohibited[5].

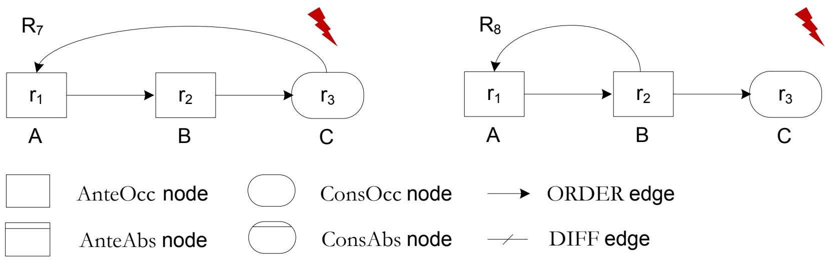


Figure 6.11.: CRGs containing cyclic ordering relations

6.2.4.2. Relations of absence nodes

As illustrated in Example 6.8, relations between absence nodes are not reasonable. Consider CRG R_9 depicted in Fig. 6.12, which contains an ORDER edge between two AnteAbs nodes. Similarly as with Example 6.8, the semantics of this relation is not clear as the associated activity executions are not supposed to occur. The same is true for the edge connecting the two ConsAbs nodes of R_{11} in Fig. 6.12. Therefore, we prohibit direct relations among absence nodes.

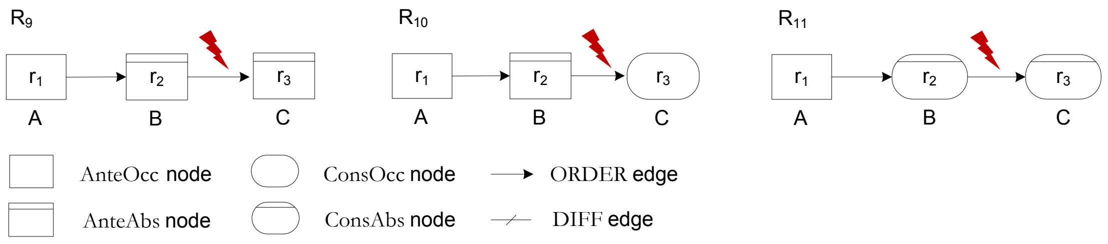


Figure 6.12.: CRGs containing ordering relations among AnteAbs, between AnteAbs and consequence or among ConsAbs nodes

[5]Note that this does not prevent the definition of unsatisfiable CRGs in general as we will discuss in Section 6.3.3.

R_{10} depicted in Fig. 6.12 contains a relation between an AnteAbs and a consequence node. As discussed in Example 6.8, it does not make sense to relate AnteAbs nodes to consequence nodes in general, as AnteAbs nodes request the absence of activity executions. If a rule becomes activated, no matching activity execution was found for the AnteAbs node in the first place. In fact, the role of AnteAbs nodes within CRGs is to enable to refine the pattern specified by AnteOcc nodes. In order to ensure that CRGs can be interpreted properly, we, therefore, further prohibit direct relations between AnteAbs and consequence nodes. Note that AnteAbs nodes may still be indirectly connected to consequence nodes.

6.2.4.3. Implicit ordering

Besides the considerations on restricting the use of relations, we further want to imply the use of explicit ordering relations when nodes are implicitly ordered in certain cases. The rationale behind doing this is to enhance the understandability of CRGs and to facilitate CRG execution (cf. Chapter 7). By introducing these syntactic conventions, subgraphs of CRGs (when removing nodes of certain types) can be analyzed separately. Here, we are considering only ORDER relations. Thus, when speaking of nodes being connected, a connection via ORDER edges is meant. Two nodes are considered ordered when they are ordered w.r.t. transitive ORDER edges.

Implicit ordering among AnteOcc nodes As ORDER relations are transitive, activity executions matching two AnteOcc nodes that are indirectly connected via ORDER edges must also occur in the implied order in order for the overall pattern to apply. For example, consider the nodes r_1 and r_3 of CRG R_{20} depicted in Fig. 6.13. Then, first A and then C has to be executed. If C and A are ordered differently in an execution trace, there cannot be any execution of B occurring after A and before C as requested by r_2. For another example, consider CRG R_{12} shown in Fig. 6.13. Then, B can only occur between A and C in the specified order for an A that occurs before C.

To facilitate the analysis of CRGs and to enforce clarity in CRG modeling, we want compliance rule modelers to make such implicit orderings as with r_1 and r_2 explicit in such cases. In particular, two AnteOcc nodes s and t that are connected (through ORDER edges) via an AnteAbs node or a set of consequence nodes such that an ordering between s and t is implied must be ordered in the subgraph of the CRG that contains only AnteOcc nodes and associated edges.

Example 6.9 (Implicit ordering among AnteOcc nodes):
Consider for example R_{12} from Fig. 6.13. Then, r_1 and r_3 are connected via r_2 and, thus, are implicitly ordered. However, when removing r_2 from R_{12}, r_1 and r_3 will no longer be ordered. Hence, the ordering between them cannot be analyzed without considering nodes of other types (i.e., r_2). In R_{13}, this is corrected.

In R_{16} and R_{20}, the AnteOcc nodes r_1 and r_2 are connected via a node of another type. When dropping the nodes of other types, r_1 and r_2 are not connected and, thus, not ordered at all. This is not desirable. In R_{17} and R_{21}, r_1 and r_2 would still be ordered when removing nodes of other types.

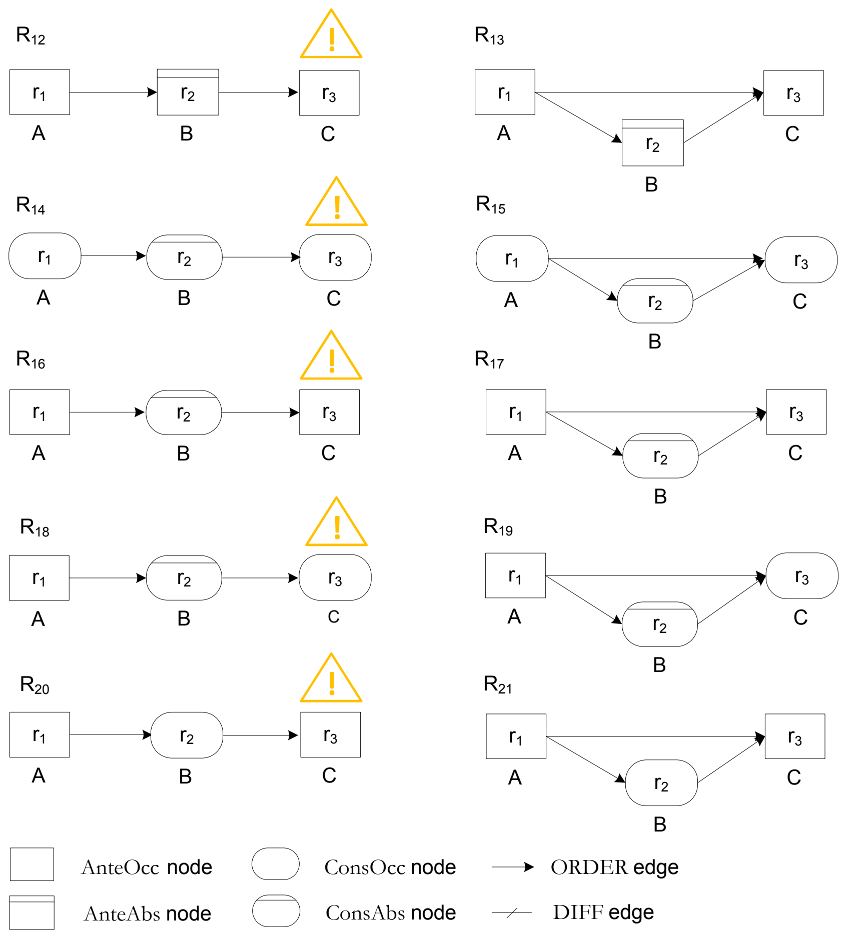


Figure 6.13.: CRGs containing ordering relations among AnteAbs, between AnteAbs and consequence or among ConsAbs nodes

Implicit ordering among ConsOcc nodes Similar as with AnteOcc nodes, two ConsOcc nodes that are connected via a ConsAbs node such that an ordering is implied shall also still be ordered when removing all ConsAbs nodes from the CRG.

Example 6.10 (Implicit ordering among ConsOcc nodes):
In R_{14} from Fig. 6.13, ConsOcc nodes r_1 and r_3 are connected through ConsAbs node r_2. However, they are not connected at all in the subgraph of R_{14} when dropping ConsAbs nodes. In contrast, r_1 and r_3 would still be connected in the respective subgraph of R_{15}.

Implicit ordering between AnteOcc and ConsOcc nodes AnteOcc and ConsOcc nodes may be connected through a ConsAbs node (as for example in R_{18} in Fig. 6.13). For these nodes, the same holds as among AnteOcc or among ConsOcc nodes. In particular, two occurrence nodes (i.e., AnteOcc or ConsOcc) s and t that are connected via a ConsAbs node[6] such that an ordering is implied, must also be ordered in the subgraph of the CRG when removing all ConsAbs nodes and associated edges.

Example 6.11 (Implicit ordering between AnteOcc and ConsOcc nodes):
In CRG R_{18} depicted in Fig. 6.13, AnteOcc node r_1 is connected with ConsOcc node r_3 via ConsAbs node r_2. When removing r_2 from R_{18}, r_1 and r_3 are no longer connected, which is not desirable. In contrast, r_1 and r_3 would still be connected after removing r_2 from R_{19}. Thus, in R_{19} the relation between AnteOcc and ConsOcc nodes can be analyzed without taking ConsAbs nodes into account.

Note that the introduced conventions lead to a small loss of expressiveness. However, we have no indication that this constitutes a limitation. In addition, these conventions are introduced primarily for convenience and for clarity. Our approach is hence able to deal with CRGs that do not adhere to these conventions.

6.2.4.4. Syntactic correctness and conventions

Based on the considerations discussed in Section 6.2.4.1, 6.2.4.2, and 6.2.4.3, we compile a set of constraints and conventions for CRGs. These constraints ensure absence of certain inconsistencies / unreasonable patterns within a modeled CRG (cf. constraints i)- iii)). In addition, they ensure that the antecedent pattern can be analyzed and executed independently from the consequence pattern and that occurrence nodes can be analyzed independently from absence nodes (cf. constraints iv) - vi)).

Definition 6.4 (Syntactic correctness and conventions for compliance rule graphs)
A CRG $R = (A, C)$ is considered syntactically correct iff the following conditions hold:

i) R is an acyclic graph with regard to $OrderE_R$.

ii) $\forall (s,t) \in OrderE_R \cup DiffE_R$ holds:
$nt_R(s) \in \{\texttt{AnteOcc}, \texttt{ConsOcc}\} \vee nt_R(t) \in \{\texttt{AnteOcc}, \texttt{ConsOcc}\}$.

iii) $\forall n \in N_R$ with $nt_R(n) = \texttt{AnteAbs}$ holds:
$\neg(\exists e = (s,t) \in OrderE_R \cup DiffE_R : (s = n \wedge t \in N_C) \ \vee \ (t = n \wedge s \in N_C))$.

iv) $\forall s \forall t \in N_A, nt_R(s) = nt_R(t) = \texttt{AnteOcc}$ with

– $\nexists n \in N_A$ with $nt_R(n) = \texttt{AnteAbs}$ such that $(s,n),(n,t) \in OrderE_A$

one of the following holds:

– $(s,t) \in OrderE_A \vee$

[6]Note that AnteAbs nodes cannot be directly connected to any consequence nodes (cf. Section 6.2.4.2).

- $\exists n_1, \ldots, \exists n_k \in N_A, nt_R(n_i) = \texttt{AnteOcc}, i = 1, \ldots, k$ such that $\exists e_1, \ldots, \exists e_{k+1} \in OrderE_A$ with $e_1 = (s, n_1), \ldots, e_{k+1} = (n_k, t)$.

v) $\forall s \forall t \in N_C, nt_R(s), nt_R(t) = \texttt{ConsOcc}$ with

- $\exists n \in N_C$ with $nt_R(n) = \texttt{ConsAbs}$ such that $(s, n), (n, t) \in OrderE_C$

one of the following holds:

- $(s, t) \in OrderE_C \vee$
- $\exists n_1, \ldots, \exists n_k \in N_C, nt_R(n_i) = \texttt{ConsOcc}, i = 1, \ldots, k$ such that $\exists e_1, \ldots, \exists e_{k+1} \in OrderE_C$ with $e_1 = (s, n_1), \ldots, e_{k+1} = (n_k, t)$.

vi) $\forall s \forall t \in N_R$ where
$(nt_R(s) = \texttt{AnteOcc} \wedge nt_R(t) = \texttt{ConsOcc}) \vee (nt_R(s) = \texttt{ConsOcc} \wedge nt_R(t) = \texttt{AnteOcc})$ with

- $\exists n \in N_R$ with $nt_R(n) \in \{\texttt{ConsAbs}\}$ such that $(s, n), (n, t) \in OrderE_R$

one of the following holds:

- $(s, t) \in OrderE_R \vee$
- $\exists n_1, \ldots, \exists n_k \in N_R$ with $nt_R(n_j) \in \{\texttt{AnteOcc}, \texttt{ConsOcc}\}, j = 1, \ldots, k$ such that $\exists e_1, \ldots, \exists e_{k+1} \in OrderE_R$ with $e_1 = (s, n_1), \ldots, e_{k+1} = (n_k, t)$.

vii) $\forall s \forall t \in N_R$, $nt_R(s), nt_R(t) = \texttt{AnteOcc}$ with

- $\exists n_1, \ldots, \exists n_k \in N_C$ such that $\exists e_1, \ldots, \exists e_{k+1} \in OrderE_R$ with $e_1 = (s, n_1), \ldots, e_{k+1} = (n_k, t)$

one of the following holds:

- $(s, t) \in OrderE_A \vee$
- $\exists n'_1, \ldots, \exists n'_l \in N_A$ with $nt_R(n'_i) = \texttt{AnteOcc}, i = 1, \ldots, l$ such that $\exists e'_1, \ldots, \exists e'_{l+1} \in OrderE_A$ with $e'_1 = (s, n'_1), \ldots, e'_{l+1} = (n'_l, t)$.

Example 6.12:
Def. 6.4 is applied in Fig. 6.14. Apparently, CRG R_{22} does not comply with Def. 6.4. For example, the AnteOcc nodes r_1 and r_3 are connected via a ConsOcc Node r_7. However, there is no ordering between r_1 and r_3 when removing r_7. This violates condition vii). The same applies to the pair of r_1 and r_5. In addition, the AnteOcc nodes r_3 and r_4 induce a violation of constraint iv), since they are connected to each other via AnteAbs node r_6 but would not be ordered when removing r_6.

Two ways of making R_{22} compliant with the constraints from Def. 6.4 are provided by R_{23} and R_{24}. The latter differ in the ordering relations between r_1, r_2 and r_3. It is notable that R_{23} and R_{24} have different semantics. While R_{23} can only be activated by sequences like $<A, B, C>$, R_{24} can also be activated by sequences of type $<A, C, B>$.

As this example shows, the constraints introduced in Def. 6.4 force CRG designers to be more accurate about a CRG's semantics and, thus, can help to avoid semantic errors.

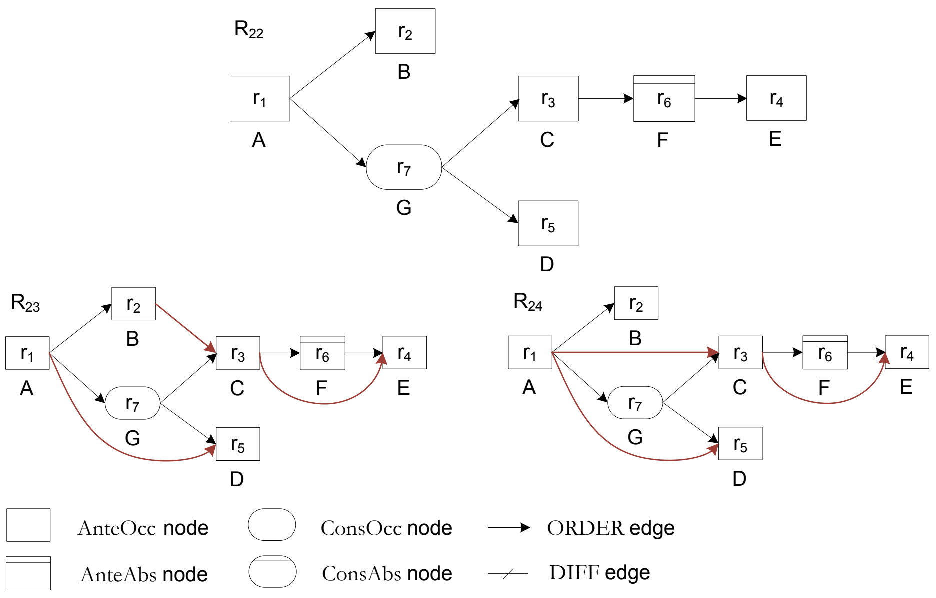


Figure 6.14.: Application of the constraints from Def. 6.4 to a CRG

The constraints from Def. 6.4 are easy to implement. To implement syntactic constraint i) we can apply existing cycle detection algorithms for graphs. The implementation of the constraints ii) and iii) is also fairly straight-forward. In particular, we have to check the source and target nodes of edges within a CRG. In order to implement the constraints iv) to vii), the particular cases that can occur within the antecedent pattern, within the consequence pattern, and within the combination of both antecedent and consequence pattern have to be identified. In the remainder of this thesis, we assume that all CRGs are correct with respect to Def. 6.4.

6.2.5. Redundant edges

A CRG complying with Def. 6.4 may contain redundant edges. An edge is considered redundant if removing it would not alter the semantics of the CRG. This means that the redundant edge does not alter the sequences that activate or satisfy a CRG. Example 6.13 shows a CRG containing a redundant ORDER edge.

Example 6.13 (A CRG containing a redundant edge):
In CRG R_{25} depicted in Fig. 6.15, (r_2, r_4) is redundant. R_1 becomes activated for each sequence of $<A, B, C>$ and will be satisfied if an execution of D follows the execution of C. Obviously, if this applies, D is also executed after completion of B. Therefore, removing (r_2, r_4) would not alter the semantics of R_{25}.

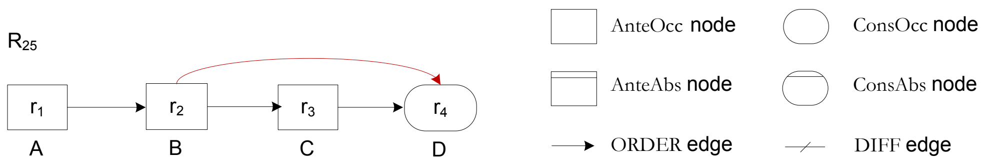


Figure 6.15.: A CRG containing a redundant ORDER edge

While redundant edges are not an issue for the formal (cf. Section 6.3) and the operational semantics (cf. Chapter 7) of CRGs, they add unnecessary complexity to a CRG, thus, making modeled CRGs more difficult to understand. In addition, they can lead to superfluous computing steps when analyzing and executing CRGs. Hence, redundant edges should be avoided. In Appendix A.1, criteria to detect redundant edges are provided. In the following, we briefly describe the underlying ideas.

6.2.5.1. Inequality relations

As previously discussed, DIFF edges correspond to inequality relations and are used to ensure that two CRG nodes are not paired with the same activity execution. The latter case can only occur if the intersection of activity executions matching with the profile of two CRG nodes is not empty. Premise for this is that two CRG nodes are assigned the same activity type from the process domain[7]. In any other case, DIFF relations are superfluous.

Since DIFF relations are not transitive, a set of DIFF relations is always free of redundancy. However, a DIFF relation between two nodes clearly becomes redundant when there is already an ORDER relation between these nodes as shown in Example 6.14. Hence, no DIFF edges shall be defined for such nodes.

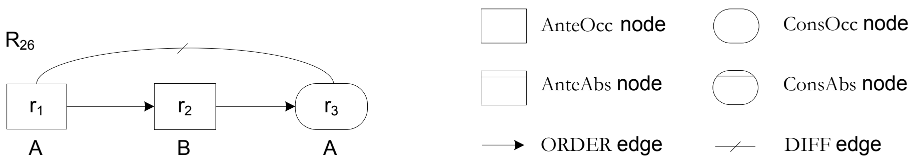


Figure 6.16.: A CRG containing a redundant DIFF edge

Example 6.14 (Redundant DIFF edges):
Consider R_{26} depicted in Fig. 6.16. Then, (r_1, r_3) is redundant as r_1 and r_3 cannot match and be paired with the same activity execution due to the ORDER relations in R_{26}.

[7]Note that even if two nodes have the same activity type, other conditions such as data conditions may exclude them from matching with the same activity execution. Thus, more sophisticated analyses may be conducted to find out whether two nodes can match with the same activity. We abstain from doing that in this thesis.

6.2.5.2. Ordering relations

Intuitively, an ORDER edge (s, t) connecting two CRG nodes s and t may be redundant if there also exists an indirect path from s to t (for example, (r_2, r_4) in Fig. 6.15). However, testing solely this is not sufficient to identify redundant ORDER edges due to the constraints in Def. 6.4, which introduce edges in CRGs (for example, (r_1, r_5) in R_{23} from Fig. 6.14). However, redundant ORDER edges can be identified easily when analyzing different CRG node types separately. By analyzing solely AnteOcc nodes and associated edges, we can easily identify redundant ORDER edges as edges (s, t) connecting two nodes s and t while there also exists an indirect path from s to t. In the same manner, we can identify redundant ORDER edges among ConsOcc nodes when analyzing solely ConsOcc nodes. As a CRG may also contain redundant ORDER edges between AnteOcc and ConsOcc nodes. To detect these, we analyze the CRG while ignoring absence nodes and associated edges.

Example 6.15 (Redundant ORDER edges):
Consider R_{27} depicted in Fig. 6.17 where redundant edges are colored. Then, analysis of solely AnteOcc nodes / ConsOcc nodes enables us to identify (r_1, r_3) / (r_5, r_7) as redundant, respectively. Analysis of edges between AnteOcc and ConsOcc nodes reveals (r_7, r_2) and (r_2, r_8) as redundant. Note that (r_7, r_1) is not a redundant edge as removing it would change the semantics of R_{27}. This edge is introduced by Def. 6.4.

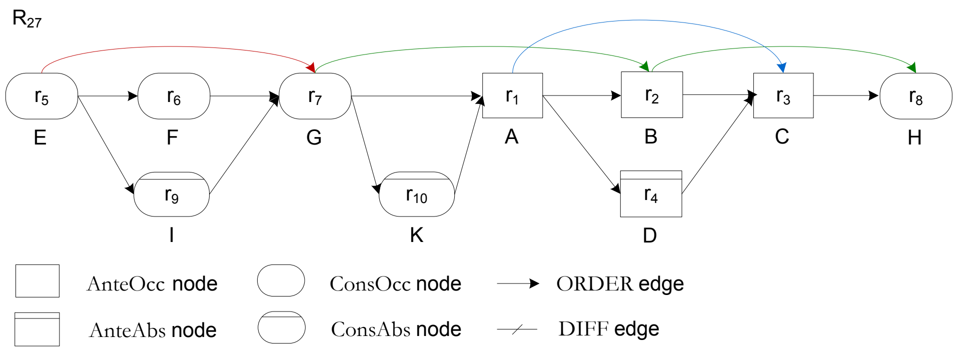


Figure 6.17.: A CRG containing multiple redundant ORDER edges

So far, we focused on redundant edges among occurrence nodes. Redundant ORDER edges associated with absence nodes can still be contained in a CRG. Redundant ORDER edges associated with absence nodes can be detected by applying similar criteria as applied to detect redundant ORDER edges among an individual node type. In particular, an ORDER edge (s, t) with s or t being an absence node is considered redundant if there is a further path from s to t in the CRG.

Example 6.16 (Redundant ORDER edges associated with absence nodes):
Consider CRG R_{28} from Fig. 6.18. R_{28} will only be activated by a sequence $<A, B, C>$ if there is no execution of D that is both in between A and B as well as in between A and C. Apparently,

any execution of D in the sequence $<A, B, C>$ that is located in between A and B will also be located in between A and C. Hence, the latter is the weaker constraint and, thus, redundant. As a result, ORDER edge (r_4, r_3) of R_{28} is redundant. The same applies to (r_6, r_5).

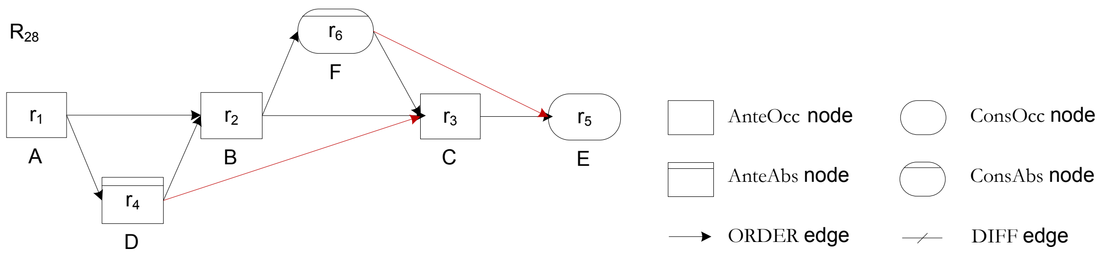


Figure 6.18.: A CRG containing redundant ORDER edges associated with absence nodes

6.2.6. Compliance rule graph composites

A CRG as formalized in Def. 6.1 consists of an antecedent pattern and *one* consequence pattern. In real-world scenarios, however, we often observe alternative options to fulfill a compliance rule [GMS06, SGN07]. For example, prior to a MRT examination, the patient should be administered a contrast agent to enhance the results. However, this is not always possible or wanted (e.g., due to intolerance against contrast agents). Hence, omitting the administration of the contrast agent with the patient's approval is also legitimate. In order to support multiple consequence patterns, we introduce *compliance rule graph composites*, which are formalized in Def. 6.5. A CRG composite is constituted by a set of CRGs with the same antecedent pattern where each CRG's consequence pattern represents one option to satisfy the CRG composite.

Definition 6.5 (Compliance rule graph composite)
A compliance rule graph composite is a tuple $RC := (A, \{C_1, \ldots, C_n\})$ where for each $i = 1, \ldots, n$ it holds that $R_i := (A, C_i)$ is a CRG.

Note that a CRG can be considered a special case of a CRG composite, namely a composite with only a single consequence. We will refer to CRG composites as CRGs for short if no distinction is necessary.

Notation-wise, it is intricate to integrate all consequence patterns into a single graph representation as it is necessary to indicate the different consequence parts. For this reason, we opted for capturing alternative consequence patterns in separate CRGs. As a result, a compliance rule consisting of multiple alternative consequence patterns from which one has to apply, is represented by a set of CRGs (having the same antecedent). This is exemplified in Fig. 6.19.

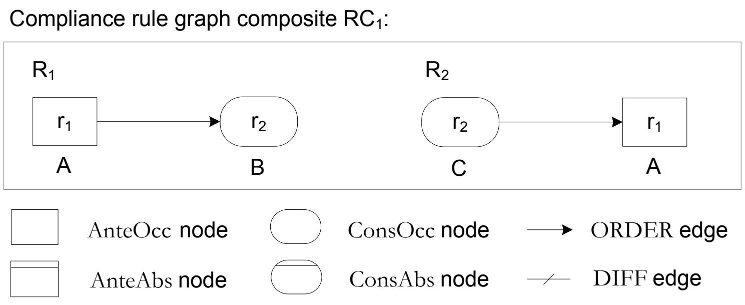


Figure 6.19.: A CRG composite consisting of two CRGs

Example 6.17 (Compliance rule graph composite):
Fig. 6.19 depicts two CRGs R_1 and R_2 constituting the CRG composite RC_1. RC_1 expresses that the execution of activity A has to be followed by an execution of B or there has to be a prior execution of C.

In the SeaFlows Toolset (cf. Chapter 10), we provide a graphical user interface to model CRG composites with multiple alternative consequence patterns by modeling the antecedent and the consequence patterns separately and relating them to each other.

6.3. Formal semantics of compliance rule graphs

In Section 6.2.2, we informally described the semantics of CRGs. To avoid ambiguity, however, the semantics of CRGs has to be formalized. While the operational semantics, which will be introduced in Chapter 7, enables the operational execution of CRGs, the formal semantics lays the foundation for the formal analysis of CRGs. Recall Fig. 6.1 from Section 6.1.2 that illustrates that each CRG composite (and CRG) can be expressed as a *compliance rule formula* (RF). A RF, in turn, can be interpreted over execution traces. This constitutes a naive process meta-model independent way to verify the compliance of processes with imposed CRG composites. We will later show, how the CRG operational semantics can be applied to verify compliance.

To specify RFs, we opted for first-order predicate logic (PL1). Due to its expressiveness[8], PL1 is suitable to capture not only the semantics of CRG composites but also of many possible future extensions thereof. In contrast to linear temporal logic, advocated by related approaches such as [ASW09], PL1 enables us to define the formal semantics of CRGs in a straight-forward manner as the explicit if-then rule structure of CRGs can be maintained in the RF.

In the following, we show how CRG composites are mapped to RFs in Section 6.3.1. First, the syntax of RFs is introduced. Then, we show how RFs are interpreted over execution traces in Section 6.3.2, which defines the semantics of RFs and, thus, the semantics of CRG composites.

[8]Note that LTL and CTL can be considered decidable notational variants of modal fragments of first-order logic [HSG04].

6.3.1. Compliance rule formulas

As explained in Section 6.2.2, CRG nodes, regardless of the type, serve as placeholders for activity executions in the overall antecedent or consequence pattern. Therefore, the basic idea to create a RF from a given CRG composite is to use *variables* that correspond to the CRG nodes in the RF. Thus, the conditions associated with CRG nodes (such as the activity type condition) and relations between nodes can be captured in the formula through adequate *functions* and *predicates* over these variables. Altogether, the primitives of a CRG composite can be reflected in a RF through respective variables, functions, and predicates.

CRG element	**RF counterpart**
CRG node n	Variable v_n
Activity type condition $at = t \in conds_R(n)$, t denotes an activity type	Predicate $=(at(v_n), t)$ (or $at(v_n) = t$ for short) where t is a constant and at is a designated function.
Node identifier condition $nid = i \in conds_R(n)$, i denotes a process node	Predicate $=(nid(v_n), i)$ (or $nid(v_n) = i$ for short) where i is a constant and nid is a designated function.
Data condition $p \odot d \in conds_R(n)$, $\odot$ is a comparison operator and d is a value	Predicate $\odot(p(v_n), d)$ (or $p(v_n) \odot d$ for short) where d is a constant, p is a designated function of the respective data parameter, and $\odot$ is a comparison operator. Note that we are double-using p as a parameter and a function name.
ORDER edge (n_1, n_2)	Predicate $Pred(v_{n_1}, v_{n_2})$
DIFF edge (n_1, n_2)	Predicate $\neq(v_{n_1}, v_{n_2})$ (or $v_{n_1} \neq v_{n_2}$ for short)

Table 6.1.: Mapping of CRG elements to RF counterparts

As shown in Table 6.1, each node of a CRG has a corresponding variable in the RF and each relation and condition of a CRG corresponds to a predicate in the RF. Hence, for a CRG composite, the corresponding RF contains a set of ANTEOCC, ANTEABS, CONSOCC, and CONSABS variables. As CRG nodes are placeholders for activity executions, RF variables also refer to activity executions[9]. The functions at, nid, and p are supposed to return the activity, the process node, and the data allocation of parameter p of the associated activity execution, respectively. As $\odot$ and $=$ are common comparison operators, their semantics should be clear. $Pred$ is supposed to be evaluated to `true` if the variables v_{n_1}, v_{n_2} are allocated with two activity executions such that the activity execution associated with v_{n_1} is completed before the other activity execution is started. Finally, predicate $\neq$ is supposed to be evaluated to `true` if the associated variables are not allocated with the same activity execution. The semantics of the predicates will be formally defined in Section 6.3.2. Based on Table 6.1, we introduce some auxiliaries that will facilitate the mapping of CRG composites to RFs:

[9] The domain for the formal interpretation of a RF is populated by activity executions contained in the execution trace, over which the RF is evaluated. We will describe this in more detail in Section 6.3.2.

Definition 6.6 (Auxiliaries for mapping CRG composites to RFs)
Let $RC = (A, \{C_1, \ldots, C_n\})$ be a CRG composite with $R_i = (A, C_i), i = 1, \ldots, n$ being CRGs. For R_i, let V_{R_i} be a set containing a distinct variable for each node of N_{R_i} and let $n(v), v \in V_{R_i}$, return the CRG node $n \in N_{R_i}$ that corresponds to variable v. Let further $V, V_1, V_2 \subseteq V_{R_i}$ be sets of RF variables. Then, based on Table 6.1, we define the following for R_i:

- $OConds_V^{R_i}$ denotes the set of $Pred$ predicates among the variables of V with:
  $OConds_V^{R_i} := \{Pred(v_1, v_2) \mid v_1, v_2 \in V \textit{ where } (n(v_1), n(v_2)) \in OrderE_{R_i}\}$.
- $OConds_{V_1|V_2}^{R_i}$ denotes the set of $Pred$ predicates between the variables of V_1 and V_2 with:
  $OConds_{V_1|V_2}^{R_i} := \{Pred(v_1, v_2) \mid v_1 \in V_1, v_2 \in V_2 \textit{ where } (n(v_1), n(v_2)) \in OrderE_{R_i}\} \cup \{Pred(v_2, v_1) \mid v_1 \in V_1, v_2 \in V_2 \textit{ where } (n(v_2), n(v_1)) \in OrderE_{R_i}\}$.
- $DConds_V^{R_i}$ denotes the set of $\neq$ predicates among the variables of V with:
  $DConds_V^{R_i} := \{v_1 \neq v_2 \mid v_1, v_2 \in V \textit{ where } ((n(v_1), n(v_2)) \in DiffE_{R_i} \vee (n(v_2), n(v_1)) \in DiffE_{R_i})\}$.
- $DConds_{V_1|V_2}^{R_i}$ denotes the set of $\neq$ predicates between the variables of V_1 and V_2 with:
  $DConds_{V_1|V_2}^{R_i} := \{v_1 \neq v_2 \mid v_1 \in V_1, v_2 \in V_2 \textit{ where } ((n(v_1), n(v_2)) \in DiffE_{R_i} \vee (n(v_2), n(v_1)) \in DiffE_{R_i})\}$.
- $NConds_V^{R_i}$ denotes the set of $\odot$, at, and nid predicates over the variables of V with:
  $NConds_V^{R_i} := \{at(v) = t \mid v \in V \textit{ where } (at = t) \in conds_{R_i}(n(v))\} \cup$
  $\{nid(v) = i \mid v \in V \textit{ where } (nid = i) \in conds_{R_i}(n(v))\} \cup$
  $\{p(v) \odot d \mid v \in V \textit{ where } (p \odot d) \in conds_{R_i}(n(v))\}$.
- $Conds_V^{R_i}$ denotes the set of all predicates among the variables of V with:
  $Conds_V^{R_i} := OConds_V^{R_i} \cup DConds_V^{R_i} \cup NConds_V^{R_i}$.
- $Conds_{V_1,V_2}^{R_i}$ denotes the set of all predicates between the variables of V_1 and V_2 with:
  $Conds_{V_1|V_2}^{R_i} := OConds_{V_1|V_2}^{R_i} \cup DConds_{V_1|V_2}^{R_i}$.

The auxiliaries are applied in Example 6.18.

Example 6.18 (Elements of a CRG expressed as variables and predicates):
Consider CRG composite RC_2 in Fig. 6.20, which consists of CRG R_1. Then, for the nodes $r_1, \ldots, r_6$ of RC_2, we define a variable for each node to be used in the RF where V denotes the set of variables with $V := \{v_1, v_2, v_3, v_4, v_5, v_6\}$. Then holds:

- $OConds_{\{v_1,v_2\}}^{R_1} := \{Pred(v_1, v_2)\}$,
- $OConds_{\{v_1,v_2\}|\{v_4,v_5\}}^{R_1} := \{Pred(v_1, v_4), Pred(v_4, v_2), Pred(v_2, v_5)\}$,
- $DConds_V^{R_1} := \emptyset$ (as no DIFF edges are used in RC_2), and
- $NConds_{\{v_1,v_2\}}^{R_1} := \{at(v_1) = A, at(v_2) = B, x(v_2) > 500\}$.

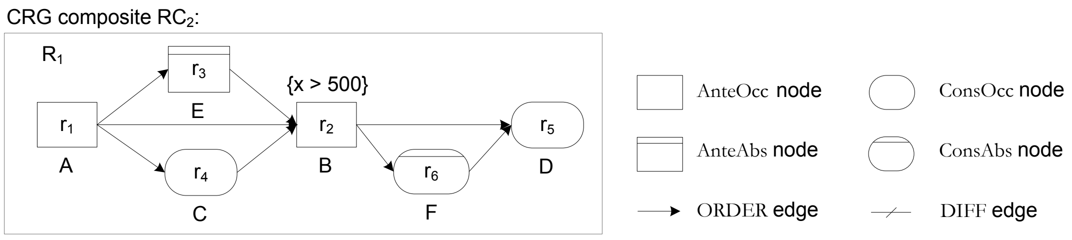


Figure 6.20.: A CRG composite consisting of one CRG

In order to derive a RF for a CRG composite, the rule counterparts have to be placed in a rule structure that reflects the intended semantics as explained in Section 6.2.2. Generally, unless there is no occurrence of the antecedent pattern within an execution trace, a CRG composite is satisfied over a trace, if for each occurrence of the antecedent pattern we can identify an occurrence of one of the consequence patterns. This if-then semantics must be reflected in RFs. Hence, the general structure of RFs is as follows (where $x_1, \ldots, x_j$ are ANTEOCC variables and $Consequence_i$ represents the formula for the consequence pattern C_i of a CRG composite):

$$\forall x_1 \ldots \forall x_j \ (Condition \Rightarrow Consequence_1 \vee \ldots \vee Consequence_p \ )$$

Condition contains predicates reflecting all conditions imposed on ANTEOCC nodes including relations among them. *Condition* further contains predicates reflecting the conditions imposed through ANTEABS nodes.

As CONSOCC nodes represent activity executions that shall occur when a compliance rule becomes activated, each $Consequence_i$ is a subformula over the existence of CONSOCC variables (assuming that $z_1, \ldots, z_l$ constitute the CONSOCC variables):

$$Consequence_i := \exists z_1 \ldots \exists z_l \ (Condition_i)$$

$Condition_i$ comprises all conditions on respective CONSOCC nodes, relations among them, and relations between CONSOCC nodes and the antecedent CRG. As CONSABS nodes impose absence constraints on the consequence pattern, they must also be reflected in $Condition_i$. Example 6.19 illustrates the mapping of a CRG composite to a RF.

Example 6.19:
Consider again RC_2 from Fig. 6.20. Then, the RF of RC_2 is as follows (the variables $v_1, \ldots, v_n$ correspond to the nodes $r_1, \ldots, r_n$):

$\forall v_1 \forall v_2$
$((at(v_1) = A \wedge at(v_2) = B \wedge x(v_2) > 500 \wedge Pred(v_1, v_2)) \wedge$ `[line 1]`
$\neg(\exists v_3 \ (Pred(v_1, v_3) \wedge Pred(v_3, v_2)))$ `[line 2]`
$\Rightarrow$
$\exists v_4 \exists v_5$

$((at(v_4) = C \wedge at(v_5) = D \wedge Pred(v_1, v_4) \wedge Pred(v_4, v_2) \wedge Pred(v_2, v_5))\wedge$ `[line 3]`
$\neg(\exists v_6\ (at(v_6) = F \wedge Pred(v_2, v_6) \wedge Pred(v_6, v_5)))))$ `[line 4]`

Although looking quite complex at a first glance, the underlying structure is fairly simple and follows $\forall v_1 \forall v_2\ (Condition \Rightarrow Consequence)$. In line 1 of *Condition*, the conditions imposed by AnteOcc nodes are expressed. In line 2 of *Condition*, the conditions in the rule's antecedent imposed by AnteAbs node r_3 are expressed. As the rule antecedent will only apply if no activity execution matching AnteAbs nodes is present, $\neg(\exists v Q_v)$ is used for the corresponding part in the RF. In line 3 of *Consequence*, all conditions imposed by ConsOcc nodes are expressed. This includes relations between ConsOcc and AnteOcc nodes. Finally, in line 4 of *Consequence*, the consequence pattern of the RF is completed by conditions imposed by ConsAbs node r_6.

Based on the above considerations, the mapping of CRG composites into RFs is generalized in Def. 6.7 using the auxiliaries in Def. 6.6. Basically, the RF counterparts of a CRG composite (i.e., the predicates) are placed in the RF structure.

Definition 6.7 (Mapping of CRG composites into RFs)
Let $RC = (A, \{C_1, \ldots, C_p\})$ be a CRG composite with $R_i = (A, C_i)$ being CRGs. Let further $X = \{x_1, \ldots, x_j\}$ be the variables corresponding to the AnteOcc and $Y = \{y_1, \ldots, y_m\}$ be the variables corresponding to the AnteAbs nodes of RC. Then, for $N_A \neq \emptyset$ the rule formula (RF) F of RC is structured as follows:

$\forall x_1 \ldots \forall x_j\ (Condition \Rightarrow Consequence_1 \vee \ldots \vee Consequence_p)$

where *Condition* is defined as:
$Condition :=$
$(\bigwedge_{con \in Conds_X^{R_1}} con)\ \wedge$
$\neg(\exists y_1\ (\bigwedge_{con \in (NConds_{\{y_1\}}^{R_1} \cup Conds_{X|\{y_1\}}^{R_1})} con)) \wedge \ldots \wedge$
$\neg(\exists y_m\ (\bigwedge_{con \in (NConds_{\{y_m\}}^{R_1} \cup Conds_{X|\{y_m\}}^{R_1})} con))$

and where for each C_i with $Z = \{z_1, \ldots, z_l\}$ being the variables corresponding to the ConsOcc and $W = \{w_1, \ldots, w_k\}$ being the variables corresponding to the ConsAbs nodes of C_i, $Consequence_i$ is structured as follows:

$Consequence_i := \exists z_1 \ldots \exists z_l\ (Condition_i)$

where $Condition_i$ is defined as:
$Condition_i :=$
$(\bigwedge_{con \in (Conds_Z^{R_i} \cup Conds_{X|Z}^{R_i})} con)\wedge$
$\neg(\exists w_1\ (\bigwedge_{con \in (NConds_{\{w_1\}}^{R_i} \cup Conds_{Z|\{w_1\}}^{R_i} \cup Conds_{X|\{w_1\}}^{R_i})} con)) \wedge \ldots \wedge$
$\neg(\exists w_k\ (\bigwedge_{con \in (NConds_{\{w_k\}}^{R_i} \cup Conds_{Z|\{w_k\}}^{R_i} \cup Conds_{X|\{w_k\}}^{R_i})} con)).$

If $N_A = \emptyset$, the RF is structured as follows:

$Consequence_1 \vee \ldots \vee Consequence_p.$

Note that $Conds_X^{R_1}$ is used in *Condition* as all these conditions refer only to antecedent nodes and all R_i of a CRG composite share the same antecedent. If no AnteOcc or ConsOcc nodes are contained in a CRG, the parts $\forall x_1 \ldots \forall x_j$ / $\exists z_1, \ldots, \exists z_l$ will be left empty in the RF, respectively, which simplifies the structure of RFs. Note that Def. 6.7, does not include conditions among AnteAbs or ConsAbs nodes and conditions between AnteAbs nodes and consequence nodes. This is because these relations would violate the correctness constraints introduced in Section 6.2.4.4.

The mapping is applied in Example 6.20 to derive a RF for a CRG composite. Clearly, this procedure can be automated. Thus, modeled CRG composites can be automatically mapped to a RF. For a given RF, in turn, the corresponding CRG notation can be derived automatically.

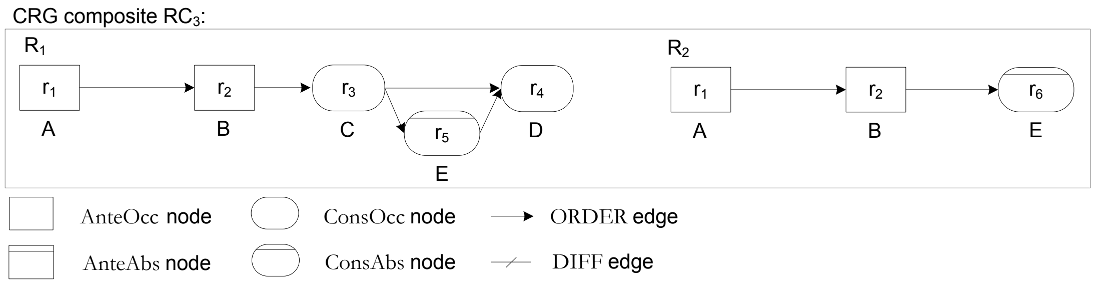


Figure 6.21.: A CRG composite

Example 6.20 (Mapping between CRG composites and rule formulas):
Consider CRG composite RC_3 that consists of two CRGs depicted in Fig. 6.21. Let further $\{v_1, \ldots, v_6\}$ be the variables for $r_1, \ldots, r_6$. Then, the RF of RC_3 is given through:

$\forall v_1 \forall v_2$
$((at(v_1) = A \wedge at(v_2) = B \wedge Pred(v_1, v_2))$
$\Rightarrow$
$\exists v_3 \exists v_4 \; ((at(v_3) = C \wedge at(v_4) = D) \wedge \neg(\exists v_5 \; (at(v_5) = E \wedge Pred(v_3, v_5) \wedge Pred(v_5, v_4))))$
$\vee$
$\neg(\exists v_6 \; (at(v_6) = E \wedge Pred(v_2, v_6))))$

RC_3 expresses that each pattern of A, B has to be followed by an occurrence of C and D without E being executed in between C and D or no E at all is allowed after B.

6.3.2. Interpretation of compliance rule formulas

So far, we showed how to derive a PL1 formula from a CRG composite. However, the derived RFs still do not have defined semantics. To achieve that, we have to stipulate how the variables, predicates, and functions of the RFs are interpreted with respect to an execution trace. An interpretation generally consists of a domain for variables and constants and a function assigning semantics to functions and predicates of a formula. As elaborated, CRG composites and RFs constrain the occurrence of activity executions and their relations within a process execution (i.e., an execution trace). Hence, variables of RFs refer to activity executions, which, in turn, are constituted by a pair of START and END events[10]. Given an execution trace, the domain of the interpretation is, therefore, populated by the activity executions contained in the trace. In addition, the domain also includes constants for activity types, node identifiers, and other values (of data allocations) occurring in the trace. The predicates and functions that are used by RFs as listed in Table 6.1 can be interpreted according to their designated semantics. In Def. 6.8, this is done by assigning to individuals of the domain those properties that are observable in the execution trace. Constants in RFs (e.g., activity types) are mapped to themselves and, hence, are omitted in Def. 6.8. Due to the structure of CRGs, derived RFs are closed formulas. Therefore, the interpretation does not consider free variables.

Definition 6.8 (Interpretation of rule formulas)
Let RC be CRG composite and let F be the RF of RC derived according to Def. 6.7. Let further $\sigma{=}{<}e_1,\ldots,e_m{>}$ be an execution trace of process model P consisting of START and END events. Then, the interpretation of F over σ is a tuple $I_\sigma{=}{<}D_\sigma, d_\sigma{>}$ with:

- D_σ is the domain of the interpretation I_σ and consists of all activity executions of σ (denoted as A_σ), which constitute the domain for variables, and all activities, process nodes, and data values of σ (denoted as C_σ) as constants.

  $A_\sigma := \{a_{i,j} \mid \exists e_i = (\text{START}, n, at, data_i),\ \exists e_j = (\text{END}, n, at, data_j)\ \mathit{in}\ \sigma :$
  $i < j \wedge \neg(\exists e_k = (\text{END}, n, at, data_k)\ \mathit{in}\ \sigma :\ i < k < j)\}.$

- For each activity execution $a_{i,j} \in A_\sigma$ with $e_i = (\text{START}, n, at, data_i)$ being $a_{i,j}$'s START and $e_j = (\text{END}, n, at, data_j)$ being $a_{i,j}$'s END event, let $ex_{i,j} = (\text{EX}, n, at, data_{i,j})$ be the EX event aggregated from $a_{i,j}$'s START and END events (i.e., $ex = actExEvent(e_i, e_j)$, cf. Def 5.2).

  Then, d_σ is a function interpreting functions and predicates of RFs over σ as follows:

  – Function nid is mapped to tuples of activity executions and their process nodes:

    $d_\sigma(nid) \mapsto \{(a_{i,j}, n) \mid ex_{i,j} = (\text{EX}, n, at, data_{i,j}),\ a_{i,j} \in A_\sigma\}.$

  – Function at is mapped to tuples of activity executions their activity types:

    $d_\sigma(at) \mapsto \{(a_{i,j}, at) \mid ex_{i,j} = (\text{EX}, n, at, data_{i,j}),\ a_{i,j} \in A_\sigma\}.$

  – The functions p are mapped to tuples of activity executions and data values of the corresponding parameters p:

    $d_\sigma(p) \mapsto \{(a_{i,j}, v) \mid v = data_{i,j}(p),\ a_{i,j} \in A_\sigma\ \mathit{and}\ p\ \mathit{is\ a\ parameter\ of}\ at\}.$

  – Predicate $Pred$ is mapped to pairs of activity executions $a_{i,j}$, $a_{k,l}$ of D_σ for which holds that $a_{i,j}$ has ended before $a_{k,l}$ has started:

    $d_\sigma(Pred) \mapsto \{(a_{i,j}, a_{k,l}) \mid j < k,\ a_{i,j}, a_{k,l} \in A_\sigma\}.$

  – The predicates $=$, $\neq$, and $\odot$ are interpreted as usual.

[10]For execution traces also containing EX events (cf. Section 5.2.4), an activity execution may also be represented by an EX event.

Note that Def. 6.8 considers execution traces consisting of only START and END events. Execution traces containing EX events are not covered. However, this is not a restriction since we can derive a pair of START and END events from an EX event (cf. Section 5.2.3). Def. 6.8 enables us to formally interpret RFs over execution traces. This is shown in Example 6.21.

Example 6.21 (Interpretation of CRG composites and RFs over execution traces):
Consider again CRG composite RC_2 from Fig. 6.20, its RF in Example 6.19, and the following execution trace:
$\sigma = <e_1, e_2, e_3, e_4, e_5, e_6, e_7, e_8>$ where

- $e_1 = (\text{START}, 1, A)$, $e_2 = (\text{END}, 1, A)$,
- $e_3 = (\text{START}, 2, C)$, $e_4 = (\text{END}, 2, C)$,
- $e_5 = (\text{START}, 3, B)$, $e_6 = (\text{END}, 3, B, \{x \mapsto 780\})$, and
- $e_7 = (\text{START}, 4, D)$, $e_8 = (\text{END}, 4, D)$.

Following Def. 6.8, $I_\sigma{=}<D_\sigma, d_\sigma>$ where

$D_\sigma := A_\sigma \cup C_\sigma$ with

- $A_\sigma := \{a_{1,2}, a_{3,4}, a_{5,6}, a_{7,8}\}$ where for each $a_{i,j}$, i / j indicates the START/ END event of a, respectively, and
- $C_\sigma := \{A, B, C, D, 1, 2, 3, 4, 780\}$.

d_σ interprets the functions and predicates of the RF as follows:

- $d_\sigma(at) := \{(a_{1,2}, A), (a_{3,4}, C), (a_{5,6}, B), (a_{7,8}, D)\}$,
- $d_\sigma(nid) := \{(a_{1,2}, 1), (a_{3,4}, 2), (a_{5,6}, 3), (a_{7,8}, 4)\}$,
- $d_\sigma(x) := \{(a_{5,6}, 780)\}$, and
- $d_\sigma(Pred) := \{(a_{1,2}, a_{3,4}), (a_{1,2}, a_{5,6}), (a_{1,2}, a_{7,8}), (a_{3,4}, a_{5,6}), (a_{3,4}, a_{7,8}), (a_{5,6}, a_{7,8})\}$.

Then, the RF in Example 6.19 is satisfied over I_σ.

Based on the interpretation specification provided in Def. 6.8, we can formally analyze execution traces with respect to satisfaction of RFs and, thus, of imposed CRG composites. Thus, we can provide a formal notion of satisfaction of RFs as shown in Def. 6.9.

Definition 6.9 (Satisfaction of rule formulas)
Let RC be a CRG composite and F be the RF derived from RC. Let further $\sigma{=}<e_1, \ldots, e_m>$ be an execution trace with $I_\sigma{=}<D_\sigma, d_\sigma>$ being the interpretation of F over σ.

Then, we say F / RC is satisfied over σ (notation: $\sigma \models F/RC$) iff holds: $I_\sigma \models F$.

The formal analysis of execution traces with respect to satisfaction of imposed RFs constitutes an approach to formally verify compliance. This corresponds to the general satisfaction problem of first-order predicate logic [Daw07], also referred to as model checking problem of first-order logic [Gro01, Gro08, CGK11] (i.e., checking whether a structure is a model for a formula). In Chapter 7, we will introduce operational semantics for CRGs, which enables the operational execution of CRGs over execution traces. The compliance state yielded when processing the execution trace is reflected through adequate markings of the CRG. Thus, instead of checking trace by trace for satisfaction of RFs, operational semantics can be applied to verify process models and process instances. This offers considerable advantages such as different levels of checking granularity or advanced feedback through analyzing the structure of CRGs.

6.3.3. Logical correctness

The syntactic constraints introduced in Section 6.2.4 do not ensure that a CRG composite is free of inconsistencies. A CRG composite is considered inconsistent (i.e., logically incorrect) if there is no execution trace whose interpretation according to Def. 6.8 satisfies its RF. Example 6.22 shows such a CRG composite.

Example 6.22 (Inconsistencies within CRG composites):
Consider RC_4 depicted in Fig. 6.22 a) with the RF as follows:
$\forall a\ ((at(a) = A) \Rightarrow \exists b\ ((at(b) = B \wedge p(b) = true \wedge Pred(a, b)) \wedge \neg(\exists c\ (at(c) = B \wedge Pred(a, c)))))$

RC_4 is not satisfiable, since, on the one hand, an execution of B with data condition $p = true$ is requested after A while, on the other hand, RC_4 also asks for the absence of B after A.

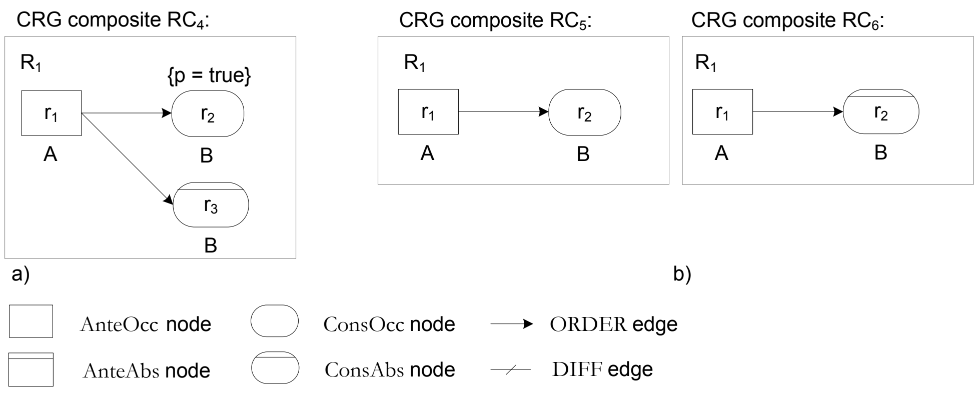


Figure 6.22.: Two conflicting CRGs whose conjunction is not satisfiable

Besides inconsistencies within a single CRG composite/ RF, a set of multiple RFs (i.e., a conjunction of the RFs) may contain inconsistencies due to conflicting requirements as shown in Example 6.23. Here, there is no execution trace whose interpretation according to Def. 6.8 satisfies all RFs of the rule set.

Example 6.23 (Inconsistencies within a set of CRG composites):
Consider RC_5 and RC_6 depicted in Fig. 6.22 b). Then, these two CRG composites are conflicting since RC_5 requests an execution of B after each execution of A while RC_6 prohibits B after A. It is obvious, that no execution trace will comply with both RC_5 and RC_6.

Clearly, it is desirable to be able to identify such inconsistencies as shown in the examples when creating new rules or imposing rules on a process. In order to detect inconsistencies, individual and sets of RFs have to be checked for satisfiability. It is well-known that the satisfiability problem is not decidable for first-order predicate logic [Chu36, Tur37]. In particular, an algorithm can be provided that halts when the formula to be checked is not satisfiable. Otherwise, the algorithm will not halt. However, many fragments of PL1 are known to be decidable. This particularly comprises the prefix classes, such as Bernays-Schönfinkel-Ramsey class ($\exists * \forall *$) [Ram30], and guarded fragments of PL1.

RFs are formalized as first-order logic formulas. It is known that the essentially finite class of first-order logic formulas containing a finite number of predicates and quantors and no functions is trivially decidable [BGG97]. When abstracting from conditions associated with CRG nodes (for example, by interpreting conditions only through quasi-propositional predicates), RFs derived from CRG composites would constitute an essentially finite class. For RFs containing functions, further investigations become necessary. This is beyond the scope of this work and is, therefore, left to future research.

While it is interesting to think about infinite models in theory, finite models are often interesting in practice. While the question whether a first-order logic formula has a finite model is still semi-decidable, the question whether a first-order logic formula has a model of size m where m is a finite number is known to be decidable. This specifically enables checking the satisfiability of formulas for a given boundary with respect to the size of the model and can be seized for practical detection of conflicting rules. As this is not the focus of this work, we abstain from providing an algorithm for this and refer to existing research, techniques, and tools for first-order logic satisfiability checking.

Besides the formal analysis of RFs, also analysis of the underlying CRGs can be conducted to identify potential sources of inconsistencies. Due to prohibiting cyclic definitions of ordering relations within CRGs, a potential source of inconsistency is already resolved. Further analyses, for example, checking that no ConsOcc node with an overlapping node profile is defined within the scope of a ConsAbs node, can be done to detect frequently occurring inconsistencies within CRGs.

6.4. Discussion

Visual modeling is common practice to document business processes and to transform them into executable specifications. Inspired by the well-established practice of specifying business process models by means of directed annotated graphs, we developed the compliance rule graph (CRG) language introduced in this chapter. CRGs were designed based on the goals described in Section 6.1.1 that were identified in our studies [LGRMD08, LRMD10, LRMKD11, LRMGD12]. With the design of CRGs, we aimed at providing a simple language consisting of modeling primitives that can be composed to capture sophisticated compliance rules. In particular, we aimed at capturing the frequent compliance rule patterns identified in literature [DAC99, ASW09, Pes08] while still enabling the definition of new compliance rules using a compositional language. Due to the pattern-oriented mental model of CRGs, it is easy to extend and refine existing CRGs by inserting or deleting modeling primitives such as absence or occurrence constraints. The explicit rule structure of CRGs facilitates the understanding of modeled rules and is exploited in the SeaFlows Toolset, our proof-of-concept prototype, to enable the separate modeling of rule antecedent and consequences. As shown in Section 6.2.2, a CRG can become activated multiple times within an execution trace. We will later show how fine-grained compliance reports for the individual rule activations and general reports for the overall compliance rule can be provided in Chapter 7.

CRGs are not only a notation but are also associated with defined formal semantics. In this chapter, we showed that each CRG composite corresponds to a rule formula (RF) specified in first-order predicate logic. This provides the basis for the formal analysis of CRGs, for example, for the detection of logical inconsistencies among a set of rules. The semantics of RFs is defined over execution traces by stipulating how they are interpreted over a trace. Extensions of the CRG language, such as time constraints, can be easily integrated into RFs as will be discussed in Section 6.4.2. In Chapter 7, we will describe how CRGs are operationalized in order to enable compliance verification of process model and process instances. In this context, the graph structure of CRGs is exploited for the operationalization and for representing compliance states in a transparent and interpretable manner.

In the following, alternatives to CRGs are discussed in Section 6.4.1. Section 6.4.2 compares CRGs to existing approaches and describes possible extensions of CRGs. Finally, this chapter closes with a description of further issues around CRGs in Section 6.4.3.

6.4.1. Related work

Formal languages and logics To avoid ambiguity, the semantics of compliance rules has to be precisely defined (cf. Section 2.1.1). Thus, their declarative formal semantics make formal languages and logics appealing for compliance rule specification. Due to their inherent notion of temporal ordering, *temporal logics* are often used in related work to capture compliance rules. In particular, *linear temporal logic* (LTL) [GLM$^+$05, Knu08, The09b, The09a, KLRM$^+$10, DGG$^+$10] or *computation tree logic* (CTL) [GK07] are employed by a multitude of approaches. The linear time semantics of LTL considers a single linear trace while in branching time semantics of CTL, each state may branch into various possible futures. As a result of the branching time semantics, LTL is considered to be more intuitive than CTL according to [Var01]. As linear time semantics seems more suitable in the business process context [ETHP10a, The09b, LMX07], LTL has been preferred over CTL for capturing compliance requirements. Besides the logic operators, the temporal operators G for globally, F for future (or eventually), X for next future, U for until and W for weak until are used in LTL. In addition to these, CTL further uses A for all and E for exists to quantify over branches. For example, $AF\phi$ means that for all branches ϕ has to hold in some future state. While LTL and CTL are expressively incomparable as there are things that can be expressed in LTL but not in CTL and vice versa, both languages can capture sophisticated compliance rules and can be enriched with first-order extensions as for example done in [GLM$^+$05, GMP06, Knu08, KLRM$^+$10] to enable data relations.

A major drawback of LTL and CTL is their complexity. To model compliance rules in LTL, for example, one has to "navigate" along the states using the temporal operators. As it is not possible to refer to a particular (earlier) state, it can be cumbersome to model compliance rules using LTL. Thus, it can be very difficult for compliance experts to model and read such formulas [YMHJ06]. This also makes it difficult to evaluate whether a modeled rule reflects the intended semantics. In consequence, a variety of approaches suggests graphical notations to hide formal details from the modeler. Aiming at facilitating the specification of LTL properties, Brambilla et al. introduce a visual notation for LTL in [Bra05, BDSV05]. In particular, their approach provides symbols for LTL operators that can be considered syntactic sugar. A similar approach for CTL is proposed by Feja et al. [FFS08, FS10], where graphical symbols are used to represent CTL and logical operators. In [LMX07, XLW08], Liu et al. propose a graphical *business property specification language* (BPSL) that can be translated into LTL and CTL. In BPSL, so-called simple temporal sequences (STS) can be used to relate events to each other. BPSL supports 14 stereotypes of STS that are available in the BPSL modeler tool. Some of these stereotypes directly correspond to LTL or CTL operators (e.g., *PossiblyLeadsto BeforeInfinity* for EF in CTL) while others constitute high-level stereotypes that correspond to templates of LTL/CTL formulas (e.g., *MultiWithin OnEvt* to specify that some event must occur multiple times within a certain scope). Then, so-called compound temporal sequences (CTS) can be used to connect STS to each other. These are basically notations for logic operators (e.g., `AND` or `OR`) and for predefined temporal relations (e.g., *Before* for W, *After* for F, and *Until* for U). While the syntactic sugaring and high level stereotypes can help to make a LTL or CTL formula appear less formal and to facilitate dealing with formulas, the underlying mental model of navigating through time and, thus, the structure of formulas remain unaffected.

Besides temporal logics, *deontic logic* has also been advocated by related work. The deontic operators *obligation* and *permission* are for example used in the OMG standard *semantic for business vocabulary and rules* (SBVR) [OMG08] that aims at providing the basis for formalizing business entities and rules. Originally designed to specify business contracts [GM05, GMS06, GM06], the *formal contract language* (FCL) can also be applied to capture compliance requirements as shown in [GHSW08, LSG07]. Besides the logical operators, deontic modalities *obligation*, *prohibition*, *permission*, and *contrary to duty* can be used in FCL formulas. A FCL formula resembles a production rule consisting of a rule antecedent and a rule consequence. According to [GM05, GMS06], a FCL compliance rule is of the form $\Gamma, E \vdash A_1 \otimes A_2 \otimes \cdots \otimes A_n$ where Γ is a set of state literals and E is the conjunction of event literals of the antecedent. A_1 is the rule consequence and $\otimes A_2 \otimes \cdots \otimes A_n$ is the reparation chain using the contrary to duty operator. Then, an execution trace will constitute an ideal situation w.r.t. the rule if the event sequence $E; A_1$ occurs in that order in the trace. Unlike temporal logics, FCL was designed for production rule like policies and, thus, does not have an inherent notion of time or ordering of events. In [SGN07], a designated time parameter is introduced to qualify each event w.r.t. time and to relate events to each other. However, formal or operational semantics for this extension is not provided in [SGN07]. A comparison of FCL and LTL w.r.t. modeling compliance requirements can be found in [The09b].

$\mathcal{S}$CIFF [ACG$^+$06, ACG$^+$08, CMMS07a] is a declarative language based on abductive logic programming. It enables the specification of forward-rules about obligations and prohibitions (referred to as positive and negative expectations) regarding events that hold at a certain time. A $\mathcal{S}$CIFF constraint expressing that after event A happens, event B is expected and after event B, event C is not expected to occur: $H(A, t_1) \rightarrow E(B, t_2), t_2 > t_1 \wedge NE(C, t_3), t_3 > t_2$. The operational specification of $\mathcal{S}$CIFF is constituted by a proof procedure.

Formalisms to specify *intertask dependencies* as used to schedule workflow transactions proposed by Klein [Kle91], Attie et al. [ASSR93], and Singh et al. [Sin96] are not sufficiently expressive to address compliance rules. *Concurrent transaction logic* (CTR) described in [DKRR98, DKR04, MDK$^+$03] has not established as a formalism for specifying compliance rules due to the availability of more suitable deontic and temporal logics.

Pattern-based approaches To avoid the drawbacks that often come with formal languages, such as high complexity, many researchers opted for pattern-based compliance rule specification. Key to such approaches is the formulation of a set of selected patterns that can be instantiated for concrete cases. The patterns are often provided in the form of visual notations (e.g., [AP06, PSSA07]), textual descriptions [AW09, The09a] or organized within a pattern classification (e.g., [YMHJ06, NS07c, SOS05]). To enable formal compliance checks, each pattern can be mapped to a logic formula. Here, existing approaches differ in the particular language to which patterns are mapped. Most often, patterns have corresponding LTL formulas.

The bulk of pattern-based approaches is based on the property specification patterns collected by Dwyer and Corbett [DAC99, DAC98]. In their work, they collected patterns from over 500 examples of property specifications from different domains. The patterns, namely *absence*, *universality*, *existence*, *bounded existence*, *precedence*, *response*, *precedence chain*, and *response chain*, can be variated by using scopes in which a property becomes relevant such as *after* or *before* scope (cf. description of the property specification patterns in Section 2.1.1.1). For these patterns, mappings into different formalisms, such as LTL or CTL, are provided. In [YMHJ06], Yu et al. introduce PROPOLS, a property pattern based specification language. Basically, PROPOLS extends the property specification patterns of Dwyer and Corbett by introducing the logical combination of patterns (so-called composite patterns) to accommodate the specification of more complex constraints. Other extensions of the property specification patterns are provided by the ConDec / DecServFlow approach [AP06, Pes08, PA06] that was originally developed to enable declarative process specification (cf. DECLARE approach [Pes08]). ConDec provides a visual notation and a mapping to LTL and $\mathcal{S}$CIFF constraints [CMMS07b] for each pattern. Besides for declarative process models, ConDec constraints are also used to capture compliance requirements [MMC$^+$11, MMWA11]. Except for choice constraints (e.g., one out of a set of activities has to be executed), ConDec constraints are unary or binary (i.e., involve at most two activities). Extensions are discussed in [Pes08], however only w.r.t. disjunction of source or target activities of constraints. While ConDec constraints can be composed using logic operators (particularly conjunction) to yield a declarative model as shown in [Pes08], a composition beyond this is not considered. Thus, the semantics encoded by the CRG depicted in Fig. 6.23 for example cannot be synthesized with ConDec constraints[11]. In particular, the conjunction of a *chain response* and an *absence* with *between* scope pattern is not sufficient to yield the semantics

[11]However, this CRG can be expressed using LTL ($G(A \rightarrow F(B \wedge \neg D\ U\ C))$).

of RC_7 as RC_7 does not generally prohibit D of occurring between B and C but rather requires *one* execution of B and C without D in between.

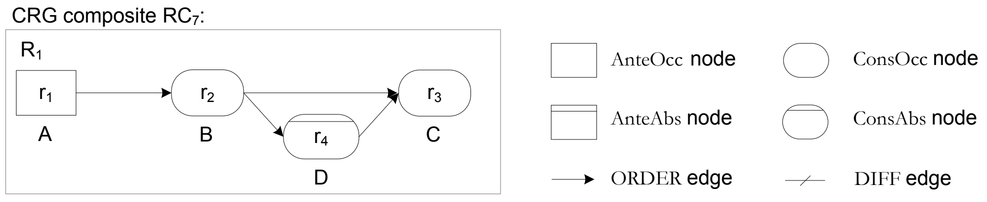


Figure 6.23.: A CRG composite that cannot be expressed as a ConDec model

The patterns described by Türetken et al. in [TEHP12] constitute extensions to ConDec / DecServFlow constraints. In addition to the order and occurrence patterns also known from ConDec, Türetken et al. further introduce time and resource (e.g., *PBondedWithQ* meaning that P and Q must be conducted by the same user) constraints.

A set of simple order, exclusion, and choice constraints that can be instantiated for concrete activities is also provided in [MS03, SOS05]. Becker et al. specifically address compliance issues in the financial sector [BBD+11, BAC+11]. Specifically, Becker et al. provide a set of compliance rule patterns for different semantic perspectives of a business process, i.e., flow rules (similar to rules proposed by Awad et al. [AW09]), organizational rules (e.g., 4-eyes-principle), and business object rules (e.g., to check whether all mandatory attributes of a data objects are provided) [BAC+11]. More sophisticated properties can be specified using the *process pattern specification language* (PPSL) proposed by Förster et al. [FES05, FESS06, FESS07]. This approach comprises a set of patterns for which a graphical notation and mappings into LTL are provided. Besides patterns that resemble those of Dwyer and Corbett, PPSL also enables patterns with so-called control nodes. Basically, these control nodes enable to refer to a set of activities within a pattern, for example, to express that a set of activities (rather than a single activity) has to be executed after the occurrence of some event. The *DecisionNode* pattern shown in Fig. 6.24, for example, expresses that a has to be followed by a b_i and each b_i has to be preceded by a. Thus, it represents the conjunction of two LTL formulas that correspond to the *response* and the *precedence* pattern [DAC99], respectively. PPSL is extended in [KGE11]. The major extensions are non-deterministic relations that correspond to CTL operators (e.g., *possiblyAll* corresponding to EG). Generally, in contrast to approaches that use two different languages for rule modeling and internal rule representation for compliance verification, our approach does not necessitate specific back-translations of the feedback obtained from compliance verification. This is because compliance verification is conducted directly based on the modeled CRGs.

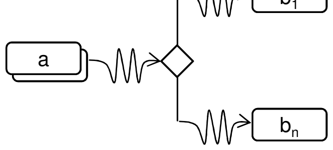


Figure 6.24.: The *DecisionNode* pattern of PPSL [FESS06]

Other approaches In addition to the rather general approaches discussed, there are also compliance specification approaches that are tailored towards specific issues. ExPDT (extended privacy definition tool) [SKGL08] proposed by Sackmann et al. and the approach proposed in [FWB$^+$10] by Feja et al., for example, are tailored towards privacy policies. In [Bit06], Bitsch describes a pattern-based approach for safety constraints based on temporal logic.

The approaches discussed so far enable the declarative specification of compliance rules. However, we also encounter approaches in literature, that propose procedural compliance rule representation. The bulk of them models compliance rules by means of some kind of automaton (e.g., [KRG07, RKG06, FUMK06]). While this may facilitate compliance checking as automatons, for example, can directly be used for model checking, we believe that declarative compliance rules correspond more to the mental model of compliance rules in practice.

Instead of capturing compliance requirements as explicit rules, an alternative approach is to specify so-called *violation patterns* representing cases violating a compliance requirement that can be used to query process executions. Querying formalism and languages such as regular path queries [LRY$^+$04] or complex event processing [JML09, WZM$^+$11] can be applied. However, as compliance rules can often be violated in multiple ways, determining all violation patterns given an informal compliance rule can be quite a challenge.

Conclusion Formal languages such as temporal logics are expressive. However, on the downside, they are often complex and can become a serious obstacle to compliance rule specification in practice. In our research, we, for example, experimented with LTL [Knu08, KLRM$^+$10] and implemented an approach to verify process models against LTL specifications [LKRM$^+$10]. Due to the complexity of LTL, we, however, dropped LTL and opted for CRGs in our later research. While $\mathcal{S}$CIFF is an interesting and expressive language, it falls short when it comes to providing support for conducting compliance checks at the operational level. Since rule violations are led back to the notion of logical inconsistency, the approach has difficulties in continuing after a violation is detected (specifically during compliance monitoring). This lacks support for continuous runtime monitoring as envisioned in Section 2.1.

Establishing set of compliance rule patterns for a business domain is an effective approach to facilitate compliance rule modeling. Although the rule patterns can be combined using logical operators, a fixed set of patterns can still be too restrictive for particular application scenarios (as, for example, shown in Section 2.1.1.1). In these cases, a compositional approach is advantageous. With the design of CRGs, we aim at combining the ease-of-use of a pattern-based approach and the flexibility of a compositional approach. The idea of establishing a pattern repository can be adopted for CRGs.

6.4.2. Expressiveness and extensions of compliance rule graphs

CRGs and CRG composites are a means to capture compliance requirements. In our approach, a compliance requirement is represented by a set of CRG composites implying that all these CRG composites have to be complied with. CRG composites, in turn, enable to define multiple consequences for a compliance rule, each representing an option to satisfy the rule.

The modeling primitives of CRGs can be used to compose complex compliance requirements. The patterns and scopes identified by Dwyer and Corbett [DAC99] can be modeled by means of CRGs as we will show in Section 11.1. However, we would like to stress that CRGs are not restricted to the property specification patterns. For example, CRGs enable the usage of ConsAbs nodes to qualify the activity executions that are required to occur (i.e., constraints of the form "There must exist an activity execution X after which no Y will occur"). As shown in Section 6.4.1 (cf. Fig. 6.23), such constraints cannot be composed just by connecting two patterns (e.g., ConDec patterns). The deontic operators *obligation* and *prohibition* can be related to the ConsOcc and ConsAbs nodes, respectively. In the prototype implementation, a mapping from a subset of CRGs into LTL has been implemented [LKRM$^+$10]. A detailed analysis of the expressiveness of CRGs (e.g., in comparison to LTL) is beyond the scope of this work and, therefore, left to future research.

So far, we only consider the conjunction of CRG composites. However, one can also think of other combinations of CRG composites (particularly disjunction) to enhance expressiveness. The contrary to duty operator discussed in [GMS06] enables the definition of reparation chains in case of violations. In [The09b], it is shown how this operator can be simulated in LTL. Similar considerations may be applied to CRGs. Moreover, CRGs can be extended with further primitives in order to add expressiveness in different respects as will be described in the following.

Data relations Data conditions can be assigned to CRG nodes in order to precisely specify activity executions that are relevant to the compliance rule. So far, we have not discussed how data parameters of different CRG nodes can be related. Such relations involving different CRG nodes are referred to as *data relations.* In particular, data relations are of the form $p_1 \odot p_2$ where p_1 and p_2 are parameters and $\odot$ is comparison operator (e.g., $=$). Data relations, for example, become necessary for expressing that the activity executions referenced by CRG nodes refer to the same data object (e.g., activities associated with the same item) or that two activities are not conducted by the same agent (four-eyes-principle).

As CRGs are formalized as PL1 formulas, such data relations can be easily integrated into RFs. While data relations are not part of CRGs as introduced in this Chapter, we already incorporated data relations into the prototype implementation. In particular, a CRG can be defined for a particular data object parameter where the latter sets the context for the CRG. Then, only activity executions that refer to the same data object are considered. The prototype implementation is described in more detail in Chapter 10. In our research, we further experimented with LTL where we use a first-order extension of LTL in order to introduce data relations. Details on this can be found in [Knu08, KLRM$^+$10]. Visualization-wise, data relations may, for example, be represented through designated nodes and edges.

Time constraints Compliance rules often involve not only qualitative (i.e., ordering) but also quantitative time constraints. Lanz et al. identified time patterns in process-aware information systems [LWR10], which we also often encounter in compliance requirements. Generally, to formally integrate time conditions, the event model introduced in Section 5.2 has to be extended with a notion of time. Time conditions that affect only single CRG nodes such as deadlines (e.g., an activity should take place no later than at a given due date) can be expressed by a designated data condition associated with that node. In practical applications, it is often necessary to impose constraints on the time distance of events. For example, the aftercare for an invasive treatment is required to take place within one day after the treatment. Such time relations can be supported using explicit time events that can be referred to by CRG nodes. We show how this can be realized in [LRMD10]. Alternatively, time relations may also be considered special data relations over predefined time parameters (e.g., start time) of activity executions. Both options should be investigated in future research.

Subrules In CRGs, absence nodes refer to single activity executions. However, sometimes it can be useful to describe the absence of a certain pattern. For example, if a patient is admitted and an invasive treatment is conducted without him being informed about the treatment and signing the confirmation of having been informed, some other post-treatment steps may become necessary. Then, in analogy to subprocesses in business process modeling, we envision that absence nodes can be assigned to patterns that are required to not occur. This can be realized with CRGs and requires adequate tool support for the modeling of such absence patterns. We leave further investigations on this to future research.

6.4.3. Further issues

Our approach can be leveraged by complementing it with further concepts from practice and research that can help to build a comprehensive compliance management framework, specifically with respect to the requirements concerning the management of compliance requirements (cf. Section 2.1.1.4). To provide support for managing compliance requirements, control objectives, and concrete checkable compliance rules, adequate tools and frameworks have to be provided that also enable to manage the lifecycle of and meta-data on these artifacts (e.g., informal description and enforcement levels (cf. Appendix A.4)). Some approaches already described the functionality to be supported [GLM$^+$05, NS07a, Nam08, GMP06, KSMP08a, The09a, The10a]. We would like to particularly refer to the work done in the COMPAS project [The09a, The10a] where also user interfaces for managing compliance requirements and checkable rules and policies were developed. Moreover, existing commercial risk and compliance tools, such as ARIS Risk & Compliance Manager, can also be used to provide support in this respect.

The integration of formal compliance modeling with business artifacts is a particular challenge. For that purpose, a domain model can be used to align terminology [SGN07] and to semantically organize the artifacts of the domain [LGRMD08, LRMGD12]. This would also contribute to facilitate compliance rule modeling as it enables to model compliance rules not based on specific activity types but for concepts of the domain model.

With respect to ease-of-use, we envision tool support for modeling CRG composites in natural language. Due to their clear structure, it should be possible to create templates for the verbalization of CRGs. One can further think of building a semantically high-level modeling approach based on CRG primitives exploiting, for example, the scopes of [DAC99] as semantic building blocks. Following the pattern idea as advocated in [DAC99, Pes08, ASW09, Nam08], we can further think of templates for frequently used CRG composites that can be instantiated for concrete cases.

7 Operational semantics of compliance rule graphs

Using the CRG language introduced in Chapter 6, compliance rules can be modeled as CRGs and CRG composites. This constitutes the first step towards automated compliance verification. In order to realize the vision of a compliance checking framework that supports the verification of process models and process instances against CRGs, adequate compliance checking mechanisms for CRGs become necessary. The direct implementation of the formal semantics of CRGs is not a viable solution as it neither exploits the ordering of events in execution traces nor the ordering of CRG nodes. In addition, it is not designed for dealing with evolving execution traces as required for compliance monitoring. This is why we opted for equipping CRGs with operational semantics that enables their execution over traces. Specifically, CRG operationalization constitutes an approach to incrementally verify compliance of execution traces with imposed CRGs. Thus, it provides the means for exploring process models with respect to compliance and for monitoring the compliance of running process instances. Key to this approach is the exploitation of the graph structure of CRGs for representing compliance states in a transparent and interpretable manner. This enables the derivation of fine-grained and comprehensive compliance diagnoses from each compliance state. Altogether, the operational semantics of CRGs introduced in this chapter provides the fundament for compliance checks in different process lifecycle scenarios as we will later discuss in Chapter 8.

In Section 7.1, we first discuss requirements and goals for CRG operationalization and introduce the basic ideas of our approach. In Section 7.2, we show how the graph structure of CRGs is exploited for representing compliance states and how the thus encoded compliance states are evaluated. Execution and marking rules for altering compliance states when START and END events are processed are introduced in Section 7.3. In Section 7.4, we show how the approach is applied to check compliance and elaborate on regulating the granularity of compliance checking. Section 7.5 discusses strategies to optimize CRG execution with regard to efficiency. The correctness of CRG operational semantics with respect to CRG formal semantics is discussed in Section 7.6. Finally, Section 7.7 summarizes the contributions of this chapter and discusses alternative approaches as well as further extension and optimization possibilities.

7.1. Introduction

In order to provide compliance support along the complete process lifecycle, an approach for verifying compliance with imposed CRGs is required that does not solely address isolated scenarios but is applicable to both process design and process runtime. However, it is not sufficient to merely detect noncompliance. As discussed in Section 2.1.2, the compliance checking approach must meet requirements w.r.t. compliance diagnoses raised by different application scenarios and agents interacting with the system. In the following, Section 7.1.1 describes requirements and goals for the compliance verification approach. Then, Section 7.1.2 describes the basic ideas of CRG operationalization before the approach is detailed in the remainder of this chapter.

7.1.1. Requirements and goals

Granularity As process models in practice can become very huge containing up two hundreds of artifacts [BRB07], a compliance rule may become activated multiple times within the model. While some activations of a rule may be satisfied, others may be violated. Therefore, fine-grained feedback becomes necessary in order to enable process designers to better evaluate the process model w.r.t. compliance. Specifically, it should be possible to reveal and pinpoint all sources of noncompliance in a process model. Similarly, a compliance rule may become activated multiple times within a process execution. In order to apply specific remedies in case of noncompliance (for example, to notify users involved in a particular case), it must be possible to provide a diagnosis for each observed rule activation when monitoring process instances.

Compliance diagnosis In Section 2.1.2, we described requirements with respect to comprehensiveness of the compliance diagnosis provided by a compliance checking framework. Clearly, in order to enable such comprehensive compliance diagnoses, the underlying compliance verification approach must be able to provide adequate feedback. Premise to this is the ability to interpret and explain detected compliance states such that meaningful compliance diagnoses, e.g., the possible cause of a violation, can be derived. Therefore, a major objective of our work is to enable the easy interpretation of effective compliance states detected through process model verification and process instance monitoring. Specifically, this should be enabled not only when compliance states bear violations but also before violations occur in order to allow for preventive measures at runtime (cf. Section 2.1.2.2).

Incremental approach In order to enable efficient compliance checks particularly at runtime, the compliance verification approach has to be able to deal with evolving traces. Verifying an evolving trace by rechecking the complete trace each time a new event is observed is not a viable solution. Instead, incremental processing must be supported.

7.1.2. Operationalization of compliance rule graphs

One way of verifying compliance with an imposed CRG is to derive an automaton that contains different compliance states of the CRG. For explicit LTL model checking, an automaton is generated from the (negated) LTL property, which is used for verification. In literature, automatons are also applied to realize compliance monitoring for LTL compliance rules [MMWA11]. In such approaches, the generated automaton observes the process execution and reaches an accepting state if the rule to be checked is satisfied. A draw-back of the automaton generation is that the states are not easily interpretable and are rather black boxes. For example, if the accepting state of an automaton used for compliance monitoring is not yet reached, one cannot use the current state to explain the compliance state of a compliance rule (for example, to derive the information that some events are still outstanding and some other events are prohibited in order to satisfy the compliance rule).

In Section 6.2.2 (cf. Example 6.4), we showed how a CRG can be manually verified over an execution trace in a pattern matching manner. This was done by checking whether events of the trace match with CRG nodes and form the patterns specified in the CRG. The basic idea of CRG operationalization is to automate this by exploiting the graph structure of CRGs. In particular, we utilize the graph structure of CRGs to represent reachable compliance states for the respective CRGs. For that purpose, we introduce execution states to mark CRG nodes in order to indicate which parts of the CRG patterns can be observed in an execution trace. Thus, instead of representing compliance states as black boxes, each compliance state of a CRG is represented by thus marked patterns. By providing rules to alter the markings when a new event is processed, we enable to update the compliance state for each observed event. This is illustrated in Fig. 7.1. Based on the markings one can easily conclude whether a CRG is activated and satisfied. Being encoded directly over CRGs, compliance states are easily interpretable and can be analyzed for providing intelligible compliance diagnoses.

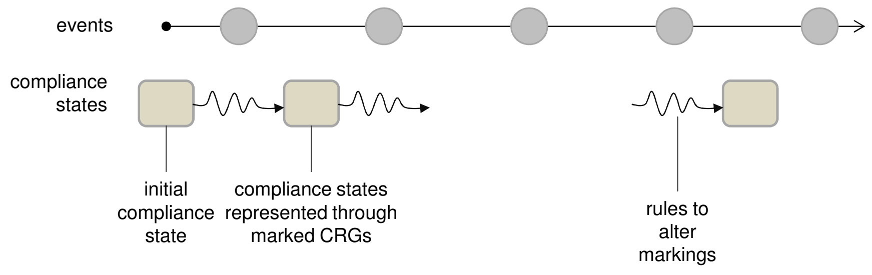


Figure 7.1.: Compliance states and transitions through events and rules

At design time, the proposed approach can be applied to explore a process model and to detect which compliance states with respect to imposed CRGs a process model is able to yield[1]. For monitoring process instances, the compliance state is incrementally updated for each new event observed during process execution.

To illustrate the CRG operational semantics, we will show how it is applied to verify compliance of an execution trace. Chapter 8 will later discuss the application of CRG operational semantics to enable compliance checks covering the process design and process runtime.

[1]We will later show how this is done based on a PEG.

For intelligibility, we will focus on CRGs instead of CRG composites (i.e., compliance rules with multiple consequences). This is not a true limitation, since the introduced concepts also apply to rules with multiple consequence parts. Since the latter do not yield additional conceptual challenges over rules with one consequence, we abstain from providing special execution and marking rules for them. In Section 7.6.2, we will describe how they are dealt with.

In order to enable CRG operationalization as shown in Fig. 7.1, suitable markings for CRGs to capture compliance states, rules to alter these when new events are processed, and notions to interpret them are required. These "ingredients" will be provided in the following.

7.2. Markings and compliance notions

In Section 7.2.1, we introduce marking structures (MarkStructures), a data structure to capture compliance states based on CRGs. In Section 7.2.2, we further introduce compliance notions based on the semantics of CRG nodes that enable the evaluation of MarkStructures and, thus, of compliance states.

7.2.1. Execution markings and marking structures

As described the basic idea of our approach is to mark CRG nodes to indicate whether parts of the CRG patterns are observable in the execution trace. To indicate directly in a CRG which events (and activity executions) have been observed in the execution trace, we introduce execution states for CRG nodes. Exploiting similarities between processes and CRGs, we adopted execution states that are employed by many process description languages such as ADEPT [Rei00] and Object Life Cycles [Mül09]. These states correspond exactly to START, END and EX events. The notation for these execution states is depicted in Fig. 7.2.

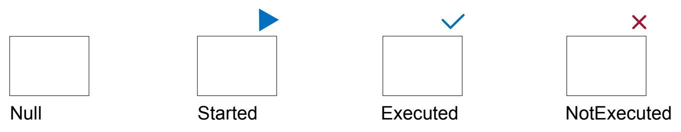


Figure 7.2.: Execution states of CRG nodes

NULL In a marked CRG, NULL indicates that for the corresponding nodes, no matching activity execution has been observed yet.

STARTED In a marked CRG, STARTED indicates that for these nodes, corresponding START events have been observed.

COMPLETED In a marked CRG, COMPLETED indicates that activity executions matching the thus marked nodes have been observed.

NOTEXECUTED In a marked CRG, NOTEXECUTED indicates that activity executions for respective nodes have not been and will not be observed[2].

[2]When applying pruning strategies for optimizing CRG execution, NOTEXECUTED is also used to indicate that a node is no longer relevant (cf. Section 7.5.1).

A CRG with nodes marked using these execution states is referred to as an execution marking (ExMark). Example 7.1 illustrates how ExMarks are used to represent compliance states of a CRG that are yielded when verifying an execution trace.

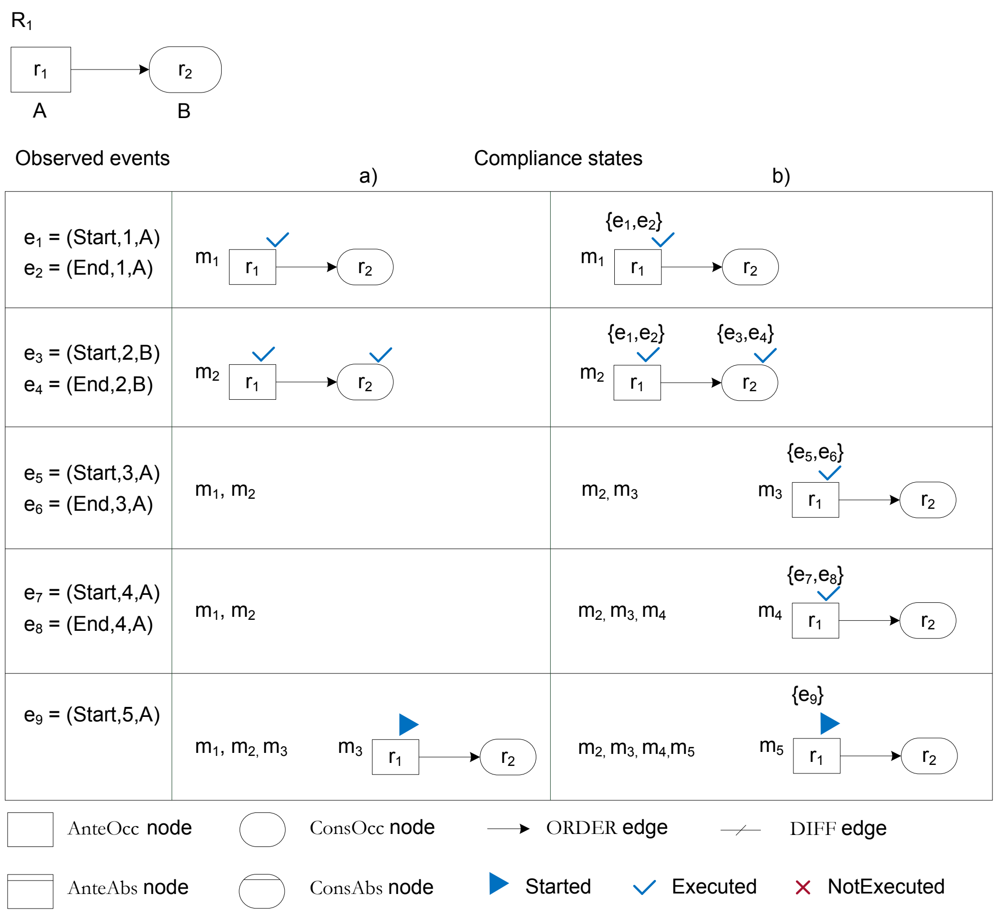


Figure 7.3.: Representing compliance states using ExMarks

Example 7.1 (CRG execution markings for capturing observed event patterns): Consider CRG R_1 and the events depicted in Fig. 7.3. Then, in Fig. 7.3 a), ExMarks are used to capture the compliance state of R_1 in different states when processing the events. Each ExMark represents an interpretation of the observed events w.r.t. the event patterns specified in R_1. For example, after the first execution of A (e_1 and e_2), ExMark m_1 can be used to reflect that an execution of A without subsequent B was observed. After execution of B, m_1 is replaced by m_2 reflecting the situation that A and subsequent B was observed. After event e_6, the compliance state can be represented by m_1 and m_2 reflecting that both an A with subsequent B as well as an A with outstanding B was observed.

In Fig. 7.3 b), ExMarks are used, in which nodes are not only assigned an execution state but are also associated with events. These ExMarks are *event-specific*. Thus, each activation of R_1 is represented by a separate ExMark. For example, m_3 and m_4 represent two different activations of

R_1 that are in the same state while in Fig. 7.3 a), these two activations of R_1 are both associated with the *event-independent* ExMark m_1.

The effective compliance state can be easily assessed by interpreting the ExMarks at any stage. In Fig. 7.3 a), for example, the compliance state after observing e_9 is reflected by m_1, m_2, and m_3. From the latter, we can conclude that there are satisfied activations of R_1 in the trace (m_2). However, there are also activations of R_1 that still necessitate further events to become satisfied (m_1 and m_3).

As shown in Example 7.1, event-independent and event-specific ExMarks provide valuable insights into compliance states. Particularly event-specific ExMarks enable to identify the events that lead to activation of a CRG and also reveal events involved in a compliance violation. Def. 7.1 formalizes ExMarks and other notions for marking CRGs. In addition to an execution state assigned through function ns, a CRG node can further be assigned a set of events (through function nl). This enables to capture event-specific patterns. The execution state and the event assignment together constitute the *state of a CRG node*.

Definition 7.1 (ExMarks, StateMarks, AnteExMarks, and ConsExMarks)
Let $R = (A, C)$ be a CRG and $NodeStates := \{\texttt{Null}, \texttt{Started}, \texttt{Completed}, \texttt{NotExecuted}\}$. We define the following:

- $ns_A : N_A \rightarrow NodeStates$ is a function assigning an *execution state* to each node of A. We refer to ns_A as antecedent state marking (AnteStateMark) of R.
- $ns_C : N_C \rightarrow NodeStates$ is a function assigning an *execution state* to each node of C. We refer to ns_C as consequence state marking (ConsStateMark) of R.
- $nl_A : N_A \rightarrow 2^{E^*}$ is a function assigning a (possibly empty) set of *events* to each node of A. We refer to nl_A as antecedent event marking (AnteEventMark) of R.
- $nl_C : N_C \rightarrow 2^{E^*}$ is a function assigning a (possibly empty) set of *events* to each node of C. We refer to nl_C as consequence event marking(ConsEventMark) of R.

We further denote as

- $NS^*_A := NodeStates^{N_A}$ the set of all AnteStateMarks,
- $NS^*_C := NodeStates^{N_C}$ the set of all ConsStateMarks,
- $NL^*_A := \mathcal{P}(E^*)^{N_A}$ the set of all AnteEventMarks, and as
- $NL^*_C := \mathcal{P}(E^*)^{N_C}$ the set of all ConsEventMarks over R.

Then, we denote

- the tuple (ns_A, ns_C), $ns_A \in NS^*_A$, $ns_C \in NS^*_C$ as *state marking* (StateMark) and as NS^*_R the set of all state markings,

- the tuple (ns_A, nl_A), $ns_A \in NS^*_A$, $nl_A \in NL^*_A$ as *antecedent marking* (AnteExMark) and as $NSL^*_A := (NS^*_A \times NL^*_A)$ the set of all antecedent markings,
- the tuple (ns_C, nl_C), $ns_C \in NS^*_C$, $nl_C \in NL^*_C$ as *consequence marking* (ConsExMark) and as $NSL^*_C := (NS^*_C \times NL^*_C)$ the set of all consequence markings, and
- the tuple $((ns_A, nl_A), (ns_C, nl_C))$ as *execution marking* (ExMark), and as M^*_R the set of all execution markings of R.

As ExMarks are based on the CRG structure, they are easy to interpret. However, sometimes, a more complex structure is necessary for adequately capturing compliance states as illustrated in Example 7.2.

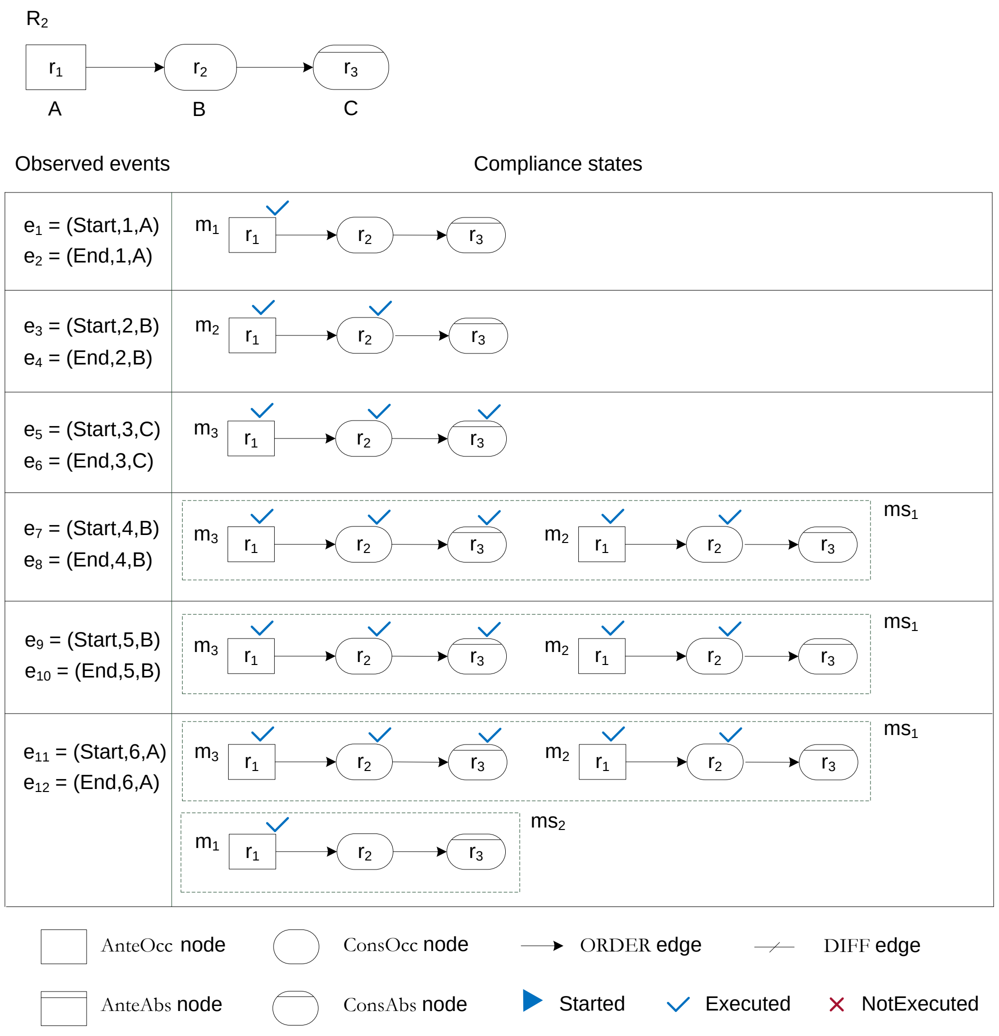


Figure 7.4.: Representing compliance states using groups of ExMarks

Example 7.2 (Groups of ExMarks):
Consider R_2 and the event depicted in Fig. 7.4. The compliance state of R_2 after the first six events can be reflected by ExMark m_3. However, after the second execution of B (i.e., after event e_8), two ExMarks with the same AnteExMark and differing ConsExMarks are necessary to adequately capture the compliance state: One ConsExMark indicating that the pattern B and a subsequent C and a further ConsExMark indicating that the pattern B without subsequent C have been observed. Note that it is also possible to use solely ExMark m_2 to represent the compliance state as the first execution of B does not lead to a satisfaction of the rule activation. However, then, the information encoded in ExMark m_3 would be dropped. The third execution of B does not lead to new ExMarks.

After the second execution of A, a further ExMark is required reflecting that A without subsequent B has been observed. Hence, the compliance state is represented by the ExMarks m_1, m_2, and m_3. However, these three ExMarks are not independent. Instead, as indicated in Fig. 7.4 they constitute two different groups of ExMarks, the first group consisting of m_2 and m_3 representing first and the second group consisting of m_1 representing the second activation of R_2. These two groups of ExMarks capture precisely the behavior with regard to R_2 displayed by the execution trace.

As Example 7.2 shows, *groups of* ExMark*s* can become necessary for capturing compliance states. For compact representation of groups of ExMarks, we introduce marking structures (MarkStructures). A MarkStructure is constituted by an AnteExMark and a set of ConsExMarks. Thus, the compliance state of a CRG can be represented through a set of MarkStructures as illustrated by Example 7.2. Definition 7.2 formalizes the notion of MarkStructures.

Definition 7.2 (MarkStructure)
Let $R = (A, C)$ be a CRG. Then, a marking structure (MarkStructure) of R is defined as a tuple

$ms := ((ns_A, nl_A), \{(ns_C^1, nl_C^1), \ldots, (ns_C^k, nl_C^k)\})$ where

- $(ns_A, nl_A) \in NSL_A^*$ is an AnteExMark,
- $(ns_C^i, nl_C^i) \in NSL_C^*$, $i = 1, \ldots, k$, is a ConsExMark, and
- $((ns_A, nl_A), (ns_C^i, nl_C^i))$, $i = 1, \ldots, k$, is an ExMark of R.

Additionally, we denote

- as $MS_R^* := NSL_A^* \times (2^{NSL_C^*})$ the set of all MarkStructures of R.

A MarkStructure ms of R is further referred to as *event-independent* if Completed nodes are not assigned any events and as *event-specific* if the opposite applies. An event-independent

MarkStructure can be considered an *equivalence class* for event sequences. For example, for R_2 from Fig. 7.4, the event sequences $<A, B>$, $<A, B, A, B>$, and $<A, B, A, B, A, B>$ can be represented by the same MarkStructure consisting of ExMark m_2 from Fig. 7.4. Thus, an event-independent MarkStructure may be associated with multiple activations of a CRG. We will focus on event-independent MarkStructures in the following. However, we will later show in Section 7.4.3 how event-specific MarkStructures can be used to provide fine-grained compliance diagnoses. In the remainder of this thesis, we will illustrate a MarkStructure by depicting the ExMarks it consists of as, for example, shown in ms_1 in Fig. 7.4.

As a MarkStructure corresponds to a set of ExMarks with the same AnteExMark and varying ConsExMarks, we provide an auxiliary function to aggregate such ExMarks to a MarkStructure in Def. 7.3. This function will be used in the CRG execution procedure.

Definition 7.3 (Aggregation of ExMarks to MarkStructures)
Let $R = (A, C)$ be a CRG. Then,

$aggregate_R : 2^{M_R^*} \rightarrow 2^{MS_R^*}$

is a function aggregating a given set of ExMarks $M = \{m_1, \ldots, m_n\}, m_i \in M_R^*$ to a set of MarkStructures $MS = \{ms_1, \ldots, ms_l\}, ms_j \in MS_R^*$ with

$aggregate_R(M) :=$
$\{ms = ((ns_A, nl_A), \{(ns_C^1, nl_C^1), \ldots, (ns_C^k, nl_C^k)\}) \in MS_R^* \mid$
$(\exists m = ((ns_A, nl_A), (ns_C, nl_C)) \in M) \wedge$
$(\forall (ns_C^i, nl_C^i), i \in \{1, \ldots, k\},\ \exists m = ((ns_A, nl_A), (ns_C, nl_C)) \in M :\ (ns_C^i, nl_C^i) = (ns_C, nl_C)) \wedge$
$(\forall m = ((ns_A, nl_A), (ns_C, nl_C)) \in M\ \exists (ns_C^i, nl_C^i), i \in \{1, \ldots, k\} :\ (ns_C^i, nl_C^i) = (ns_C, nl_C))\}.$

Recall Fig. 7.1. A compliance state in our approach is represented by a set of MarkStructures. Being based on CRGs marked with execution states, MarkStructures are suitable for capturing compliance states in a transparent and interpretable manner. To represent the initial compliance state (cf. Fig. 7.1), which is comparable to the start state of an automaton, we utilize a MarkStructure where all CRG nodes are assigned state (Null, $\emptyset$)). IR1 derives such a MarkStructure for a given CRG.

Initialization Rule IR1 (Intialization of a MarkStructure):
For all CRGs $R = (A, C)$, we define *init* as a function returning an initialized MarkStructure *ms* of the given CRG R with:

$init(R) := ms = ((ns_A, nl_A), \{(ns_C, nl_C)\}) \in MS_R^*$ with

- $(ns_A, nl_A) \in NSL_A^*$ is an AnteExMark of R and
- $(ns_C, nl_C) \in NSL_C^*$ is a ConsExMark of R
- where holds:
  - $(\forall n \in N_A :\ (ns_A(n), nl_A(n)) := (\text{Null}, \emptyset)) \wedge$

– $(\forall n \in N_C : \ (ns_C(n), nl_C(n)) := (\text{NULL}, \emptyset))$.

Example 7.3 (Initialization of a MARKSTRUCTURE for a CRG):
Consider again R_2 from Fig. 7.4. Then, $init(R_2) = ms = ((ns_A, nl_A), \{(ns_C, nl_C)\})$ where:

- $ns_A(r_1) = \text{NULL}$, $nl_A(r_1) = \emptyset$, $ns_C(r_2) = ns_C(r_3) = \text{NULL}$, and $nl_C(r_2) = nl_C(r_3) = \emptyset$.

7.2.2. Evaluation of marking structures

In order to assess the compliance states of a CRG detected through process model or process instance verification, criteria to evaluate the MARKSTRUCTUREs become necessary. According to the semantics of CRG nodes (cf. Section 6.2.2) and the marking semantics of MARKSTRUCTUREs, occurrence nodes have to be marked with COMPLETED while absence nodes must not be marked with COMPLETED in order to reflect that the overall event pattern was observed. Based on this consideration, Def. 7.4 formalizes notions to evaluate whether a CONSEXMARK represents an occurrence of a CRG's consequence pattern.

SATISFIED means that the CONSEXMARK represents an occurrence of the CRG's consequence pattern.

VIOLATED means that the CONSEXMARK does not represent an occurrence of the CRG's consequence pattern. Hence, a VIOLATED CONSEXMARK cannot contribute to the satisfaction of a CRG.

VIOLABLE means that the CONSEXMARK does not yet represent an occurrence of the CRG's consequence pattern.

PENDING means that a VIOLABLE CONSEXMARK contains CONSOCC nodes for which no matching activity execution has been observed yet.

Definition 7.4 (Notions for CONSEXMARKS)
Let $R = (A, C)$ be a CRG and $cm = (ns_C, nl_C) \in NSL_C^*$ be a CONSEXMARK over R. Then, we distinguish between the following states of cm:

- cm is SATISFIED if holds:
  $(\forall n \in N_R : nt_R(n) = \text{CONSOCC} \Rightarrow ns_C(n) = \text{COMPLETED}) \wedge$
  $(\forall n \in N_R : nt_R(n) = \text{CONSABS} \Rightarrow ns_C(n) = \text{NOTEXECUTED})$.
- cm is VIOLATED if holds:
  $(\exists n \in N_R : nt_R(n) = \text{CONSOCC} \Rightarrow ns_C(n) = \text{NOTEXECUTED}) \vee$
  $(\exists n \in N_R : nt_R(n) = \text{CONSABS} \Rightarrow ns_C(n) = \text{COMPLETED})$.
- cm is VIOLABLE if holds:
  cm is neither SATISFIED nor VIOLATED.

- cm is PENDING if holds:
  cm is VIOLABLE $\wedge$ $(\exists n \in N_R : nt_R(n) = \text{ConsOcc} \wedge ns_C(n) \in \{\text{Null}, \text{Started}\})$.

Further,

- cm is referred to as FINAL if holds:
  $\forall n \in N_C : ns_C(n) \notin \{\text{Null}, \text{Started}\}$.
- Otherwise, cm is referred to as NON-FINAL.

Note that if a VIOLABLE ConsExMark is not PENDING, no further events are required in order to render the ConsExMark SATISFIED. However, the observation of certain events that match with ConsAbs nodes of the CRG may render the ConsExMark VIOLATED.

Example 7.4 (Evaluation of ConsExMarks):
Consider again the MarkStructures ms_1 and ms_2 from Fig. 7.4. Then, in ms_1 the ConsExMark of m_3 is VIOLATED as ConsAbs node r_3 is marked as Completed while the ConsExMark of m_2 is VIOLABLE but not PENDING. MarkStructure ms_2, however, contains a PENDING ConsExMark.

Based on Def. 7.4, we can now provide notions for assessing MarkStructures:

Activation The activation property indicates whether a MarkStructure represents an activation of the corresponding CRG. An ACTIVATED MarkStructure indicates that the antecedent pattern activating the CRG was observed. In contrast, a DEACTIVATED MarkStructure is not associated with an occurrence of the CRG's antecedent pattern. Thus, a DEACTIVATED MarkStructure constitutes no activation of the CRG and, therefore, is irrelevant to the compliance with the corresponding CRG. Finally, an ACTIVATABLE MarkStructure is not yet but can still become an activation of the CRG. An ACTIVATABLE MarkStructure, however, may also become DEACTIVATED when certain events are or are not observed in the further execution

Satisfaction The satisfaction property refers to whether an ACTIVATED MarkStructure is also associated with a satisfied consequence pattern. A MarkStructure is only considered VIOLATED if it becomes apparent that the observed behavior precludes the satisfaction of the CRG's consequence pattern in the future therewith rendering the MarkStructure no longer satisfiable.

Finality A FINAL MarkStructure contains only ExMarks whose nodes can no longer change their states. This is the case when nodes are Completed or NotExecuted. Thus, a FINAL MarkStructure does not require any further processing when verifying compliance.

Definition 7.5 (Notions for MarkStructures)
Let $R = (A, C)$ be a CRG. Let further $ms = ((ns_A, nl_A), \{(ns_C^1, nl_C^1), \ldots, (ns_C^k, nl_C^k)\}) \in MS_R^*$ be a MarkStructure of R. Then,

- we will say ms is ACTIVATED if holds:
  - $(\forall n \in N_R : nt_R(n) = \texttt{AnteOcc} \Rightarrow ns_A(n) = \texttt{Completed}) \wedge$
    $(\forall n \in N_R : nt_R(n) = \texttt{AnteAbs} \Rightarrow ns_A(n) = \texttt{NotExecuted})$.
- We will say ms is DEACTIVATED if holds:
  - $(\exists n \in N_R : nt_R(n) = \texttt{AnteOcc} \wedge ns_A(n) = \texttt{NotExecuted}) \vee$
    $(\exists n \in N_R : nt_R(n) = \texttt{AnteAbs} \wedge ns_A(n) = \texttt{Completed})$.
- Otherwise, we say ms is ACTIVATABLE.

For the ACTIVATED MarkStructure ms, we further distinguish between the following states:

- ms is SATISFIED if holds:
  $\exists(ns_C^i, nl_C^i), i \in \{1, \dots, k\} : (ns_C^i, nl_C^i)$ is SATISFIED.
- ms is VIOLATED if holds:
  $\forall(ns_C^i, nl_C^i), i \in \{1, \dots, k\} : (ns_C^i, nl_C^i)$ is VIOLATED.
- ms is VIOLABLE if holds:
  ms is neither SATISFIED nor VIOLATED.
- ms is PENDING if holds:
  ms is VIOLABLE $\wedge$ $\forall(ns_C^i, nl_C^i), i \in \{1, \dots, k\}$: $((ns_C^i, nl_C^i)$ is VIOLABLE $\Rightarrow$ (ns_C^i, nl_C^i) is PENDING$)$.

Finally, a MarkStructure ms

- is referred to as FINAL if holds:
  $\forall n \in N_A : ns_A(n) \notin \{\texttt{Null}, \texttt{Started}\} \wedge \forall(ns_C^i, nl_C^i), i \in \{1, \dots, k\}$: (ns_C^i, nl_C^i) is FINAL.
- Otherwise, ms is referred to as NON-FINAL.

The notions from Def. 7.5 enable the assessment whether a MarkStructure constitutes an activation of a CRG and whether a MarkStructure is satisfied. A reached compliance state, as depicted in Fig. 7.1, is constituted by the individual states of its MarkStructures. This is shown in Example 7.5.

Example 7.5 (Evaluation of MarkStructures):
Consider again the MarkStructures ms_1 and ms_2 from Fig. 7.4. Then, applying Def. 7.5 reveals that both are ACTIVATED and VIOLABLE. Therefore, R is not enforced. However, while ms_1 is not PENDING as it contains a ConsExMark that is not PENDING, ms_2 is still PENDING. As a result, ms_2 still requires a further activity execution (namely of B) in order to become SATISFIED while ms_1 would become SATISFIED if not further events are observed. This information can be helpful, for example, to decide on activities to be scheduled when monitoring process instances.

7.3. Execution and marking rules

So far, we showed how compliance states can be represented using event-independent and event-specific MarkStructures and provided criteria to assess MarkStructures and compliance states. To complete the operationalization of CRGs, the transitions between compliance states have to be defined. For this purpose, we define execution and marking rules that alter MarkStructures in accordance to an event observed during the exploration of a process model or when monitoring a process instance. For clarity, we will focus on *event-independent* MarkStructures. In Section 7.4.3, we will show how *event-specific* MarkStructures can be addressed.

Section 7.3.1 discusses important considerations for devising execution and marking rules. Then, rules for processing START and END events are introduced in Section 7.3.2 and 7.3.3, respectively. A marking rule for finalizing MarkStructures when ending CRG execution is introduced in Section 7.3.4. Section 7.4 then illustrates the application of the rules.

7.3.1. Fundamentals

When exploring a process model or verifying a process instance execution, each new event observed may affect the compliance state. Thus, the MarkStructures that constitute the effective compliance state have to be updated for each new event. The processing of one event is referred to as an *execution iteration* or *iteration* in brief. In each iteration, it is checked based on the current MarkStructures whether the new event matches with any CRG node. If so, it is attempted to use the new event to form patterns specified in the CRG by integrating it into the existing MarkStructures (using the node execution states described in Section 7.2.1) yielding new MarkStructures. This way, the updated MarkStructures are not built from scratch for each new event but are derived from the existing ones. This enables dealing with incrementally evolving execution traces as required for compliance monitoring.

In order to derive new MarkStructures from existing ones such that the compliance state is reflected correctly, the following questions arise:

Matching How to determine whether an observed event is relevant to a CRG node? As described in Section 6.2.2, premise is that the event matches the node's profile. However, whether a node's state has to be altered may also depend on the markings of its predecessors. For END events, attention has to be paid to the correspondence of START and END events.

Marking How to mark nodes in a MarkStructure such that the compliance notions from Section 7.2.2 can be applied at each compliance state? Started and Completed are intuitive, for these states correspond to observations of suitable START and END events. Intuitively, a CRG node is marked as Started to reflect that a matching START event is observed. In analogy, Completed indicates that a matching activity execution is observed (i.e., a START and a corresponding END event). In contrast, NotExecuted requires further clarification.

Exploring Provided that the observed event is relevant to at least a CRG node, how to automatically derive resulting MarkStructures? Due to the different node types of CRGs, execution rules from process operationalization as, for example, known from ADEPT [Rei00] are not applicable. In fact, the semantics of the CRG node types has to be considered in order to rigorously implement the formal semantics of CRGs. In order to automatically attempt to form the patterns specified in the CRG using observed events, different options of integrating a new observed event into an existing `MarkStructure` have to be explored. Each of these results in a `MarkStructure`.

In the following, the above questions will be addressed in detail before the execution and marking rules for dealing with START and END events are provided in Section 7.3.2 and 7.3.3, respectively. Section 7.3.1.1 discusses the notion of match between CRG nodes and events. Section 7.3.1.2 discusses premises to starting a CRG node. The correspondence of START and END events is discussed in Section 7.3.1.3. Section 7.3.1.4 describes how `NotExecuted` is used to indicate not executed predecessor nodes. Section 7.3.1.5 discusses execution semantics to execute CRG nodes over START and END events. Section 7.3.1.6 and Section 7.3.1.7 discuss firing conflicts that result from concurrently executable CRG nodes and how they are dealt with.

7.3.1.1. Matching between compliance rule graph nodes and events

To update a `MarkStructure`, it first has to be checked whether the observed event is relevant to any node of the respective CRG. As described in Section 6.2.2, premise to this is that conditions associated with the node apply to the event. Otherwise, the event is irrelevant to the node. As a CRG node describes a complete activity execution while START and END events mark the start and end of activity executions, the notion of match between a CRG node and such events has to be devised accordingly.

While START and END events always contain static information (i.e., node identifier and activity assigned), they may also contain dynamic information in the form of data allocations (cf. Section 5.2.3). A CRG node, in turn, can be assigned conditions on static as well as on dynamic information. These conditions are, thus, partitioned into start and end conditions depending on whether the conditions can be evaluated over START or END events (cf. Def. 6.2). Based on this, Def. 7.6 formalizes functions to assess whether a START/ an END event matches the specifications of a CRG node. These functions will be used to check the necessary conditions for executing CRG nodes (cf. Sections 7.3.2.1 and 7.3.3.1).

Definition 7.6 (Matching of CRG nodes with START and END events)
Let $R = (A, C)$ be a CRG and $s = (\text{START}, n_s, at_s, data_s) \in E^{Start}$ be a START event and $e = (\text{END}, n_e, at_e, data_e) \in E^{End}$ be an END event. Then,

- $matchStart_R : N_R \times E^{Start} \to \mathbb{B}$ is a function returning `true` if the given START event satisfies the start conditions (i.e., conditions in $startConds_R(n)$, cf. Def. 6.2) of the given node. Otherwise `false` will be returned.
- $matchEnd_R : N_R \times E^{End} \to \mathbb{B}$ is a function returning `true` if the given END event satisfies the end conditions (i.e., conditions in $endConds_R(n)$, cf. Def. 6.2) of the given node. Otherwise `false` will be returned.

Note that even though a START event matches a node, there is still the possibility of the corresponding END event not matching the CRG node due to the end conditions (cf. Example 7.6). This has to be provided for by the CRG execution and marking rules.

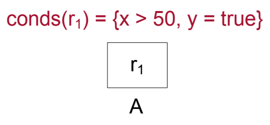


Figure 7.5.: A CRG node associated with data conditions

Example 7.6 (Matching of START and END events with CRG nodes):
Fig. 7.5 depicts a CRG node r_1. Now, assume that x is an input and y is an output parameter. Then, START event $e_1 = (\text{START}, A, 1, \{x \mapsto 60\})$ matches with r_1 as e_1 is not only associated with the expected activity but the start condition (on x) also applies. While END event $e_2 = (\text{START}, A, 1, \{y \mapsto true\})$ also matches with r_1, END event $e_3 = (\text{START}, A, 1, \{y \mapsto false\})$ does not due to the end condition (on y).

7.3.1.2. Completed predecessor nodes

A premise to start a process node is usually that all predecessors have been completed or, if a node has one-of-all join semantics, have been marked as no longer executable [Rei00]. Exploiting the ordering of CRG nodes for operationalization clearly helps to prevent unnecessary operations. Consider, for example, a CRG antecedent consisting of a sequence of AnteOcc nodes. Then, starting the last AnteOcc node (when observing a matching START event) before activity executions for its predecessors are observed (and these nodes are marked accordingly) does not contribute to identifying the antecedent pattern in the trace and, therefore, is superfluous.

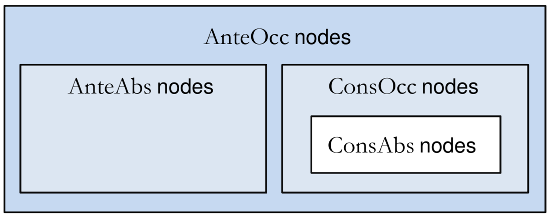


Figure 7.6.: Hierarchy of nodes within a CRG / RF

In contrast to process models, however, CRGs consist of four different node types with different semantics. Each CRG node, therefore, may have predecessors of other node types. To determine whether a CRG node can be started over a matching START event, we can exploit the hierarchy of nodes within a CRG depicted in Fig. 7.6. This hierarchy can be derived from the semantics of the nodes within a RF and shows the potential relations between nodes. Nodes from an inner block may refer to nodes from the outer blocks (e.g., ConsAbs nodes may refer to ConsOcc and AnteOcc nodes but not to AnteAbs nodes) or to nodes from the same type (does not apply to absence nodes). Based on the node hierarchy, we can stipulate the following:

AnteOcc nodes are started independently from the state of their predecessors of other node types (i.e., AnteAbs, ConsOcc, and ConsAbs). This is necessary to detect all occurrences of a CRG's antecedent pattern.

AnteAbs nodes request the absence of certain activity executions. The particular location where the associated activities must not occur can be specified through AnteOcc predecessors and successors. Due to this location dependency, AnteAbs nodes can only be started if all AnteOcc predecessors have been completed.

ConsOcc nodes request the occurrence of certain activity executions. The particular requested location of these is specified through relations of ConsOcc nodes to other ConsOcc nodes as well as to AnteOcc nodes. Thus, ConsOcc nodes can only be started if all ConsOcc as well as AnteOcc predecessors have been completed.

ConsAbs nodes The particular location where the activity executions associated with ConsAbs nodes must not occur can be specified through AnteOcc as well as ConsOcc predecessors and successors. Hence, ConsAbs nodes can only be started if all ConsOcc and AnteOcc predecessors have been completed.

Table 7.1 summarizes the start preconditions w.r.t. completion of predecessor nodes. Completed means that predecessors of that type have to be completed first. Combinations that cannot occur due to syntactic constraints are marked as "not applicable" (i.e., no direct predecessors of this node type can be contained in a CRG). Example 7.7 illustrates these preconditions.

	AnteOcc	AnteAbs	ConsOcc	ConsAbs
AnteOcc predecessors	Completed	Completed	Completed	Completed
AnteAbs predecessors		not applicable	not applicable	not applicable
ConsOcc predecessors		not applicable	Completed	Completed
ConsAbs predecessors		not applicable		not applicable

Table 7.1.: Preconditions for transforming a CRG node from Null into Started

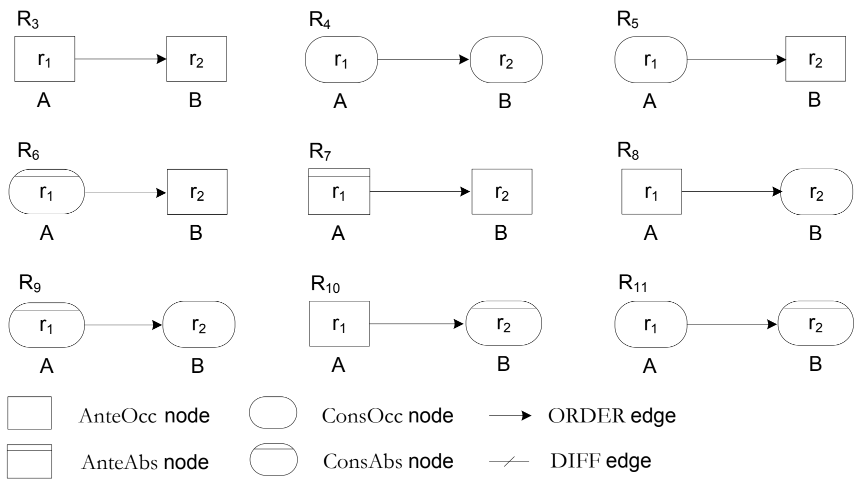


Figure 7.7.: CRG fragments

Example 7.7 (Execution preconditions of CRG nodes):
Consider the CRG fragments R_3 and R_4 from Fig. 7.7. Then, it is apparent to not start r_2 until r_1 has been completed. However, the situation is different for R_5 and R_6 stating that before B is executed, A must / must not be executed. To implement the semantics of AnteOcc nodes, r_2 must be started independently of r_1 in these cases in order to account for the case that B occurs but not A. For the same reason, r_2 has to be started independently of r_2 for R_7 as well.

In R_8, ConsOcc node r_2 is supposed to appear as response to r_1. Therefore, r_2 must not be started before completion of r_1. In contrast, ConsOcc node r_2 must be started independently from ConsAbs predecessor r_1 in R_9 in order to account for the execution of B without prior execution of A.

In R_{10} and R_{11}, ConsAbs node r_2 must not be started until its predecessors are completed since the location where the activity execution associated with r_2 must not occur is defined by the predecessors, respectively.

7.3.1.3. Correspondence of START and END events

Assuming a correct execution trace (cf. Section 5.2.4), a START event is always followed by a corresponding END event. In particular, for an observed START event, the corresponding END event is the next END event that is associated with the same process node as shown in Fig. 7.8. Clearly, a STARTED CRG node must not be completed over an END event that does not belong to the same activity execution (cf Example 7.8). To ensure this, we have to be able to determine whether an observed END event is *expected* by a START event. For this purpose, it is necessary to record START events associated with a STARTED CRG node[3] in order to be able to identify relevant END events occurring subsequently in the execution trace. To record the START events associated with a CRG in a MarkStructure, we use the function *nl* of ExMarks (cf. Def. 7.1). As a result, a STARTED process node is always associated with a set of START events.

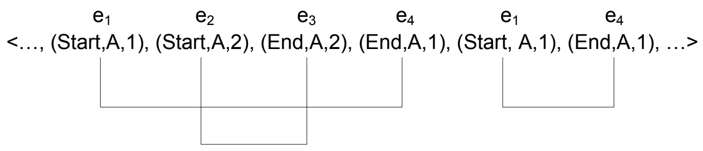

e1
e2
e3
e4
e1
e4
<..., (Start,A,1), (Start,A,2), (End,A,2), (End,A,1), (Start, A,1), (End,A,1), ...>

Figure 7.8.: Correspondence of START and END events

Example 7.8 (Correspondence of START and END events):
Consider, for example, the execution trace depicted in Fig. 7.8. Then, it is apparent that a CRG node started on e_1 must not be completed on e_3 or the *second occurrence* of e_4 as these events do not belong to the same activity execution. In particular, e_3 belongs to a different process node and the second occurrence of e_4 belongs to a later activity execution. In fact, the

[3]We will later show that a STARTED CRG node can be associated with multiple START events and, thus, can be completed by multiple END events.

END event corresponding to e_1 is the first occurrence of e_4. Even though e_4 occurs twice in the trace, these occurrences must not be confused as they belong to different executions of the same process node.

7.3.1.4. Not completed predecessors

As learned in Section 7.3.1.2, AnteOcc nodes can be started independently from the execution state of predecessors of other node types while ConsOcc nodes can be started independently from the execution state of ConsAbs predecessors. Therefore, it is possible that predecessors are still in state Null or Started when starting an occurrence node n. Then, its predecessor nodes represent expectations on the past and, hence, cannot be completed according to the semantics of CRGs. To reflect this in the MarkStructure, such predecessors are marked as no longer executable using NotExecuted. This enables the evaluation of MarkStructures using the notions provided in Section 7.2.2.

Example 7.9 (Not yet completed predecessors):
Consider MarkStructure ms_1 depicted in Fig. 7.9. Then, when marking r_3 as Started, the ConsOcc and ConsAbs predecessors r_1 and r_2 will be marked as NotExecuted as they impose constraints on the past. This results in ms_2.

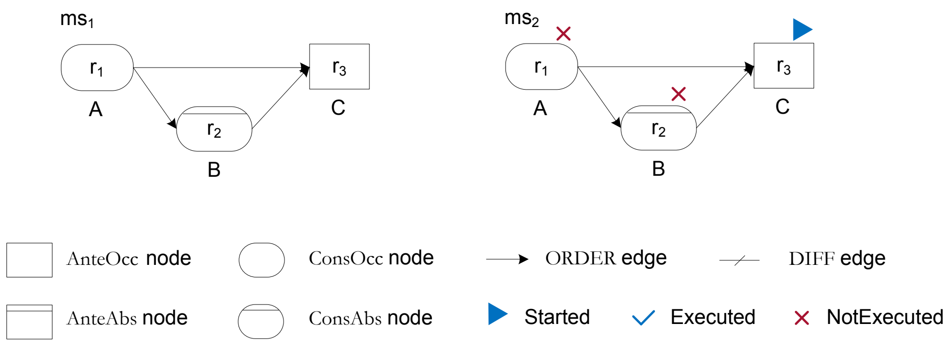


Figure 7.9.: Marking not yet completed predecessors for a started node

Def. 7.7 provides functions to identify not yet completed predecessors. These functions will be applied by the marking rules (MR1 and MR3).

Definition 7.7 (*deadConsOcc*, *deadConsAbs*, *deadAnteAbs*)
Let $R = (A, C)$ be a CRG. Then,

- $deadConsOcc_R : NS_C^* \times \mathcal{P}(N_R) \rightarrow \mathcal{P}(N_R)$ with

$deadConsOcc_R(ns_C, Q) :=$
$\{n \in N_R \mid ns_C(n) \in \{\texttt{Null}, \texttt{Started}\} \ \wedge \ \exists l \in Q : n \in predConsOcc^*(l)\}$

is a function assigning a set of `ConsOcc` predecessor nodes that have not been completed yet in `ConsStateMark` ns_C to a set of nodes Q. These `ConsOcc` nodes are regarded as dead predecessors when the nodes in Q are started.

- $deadConsAbs_R : NS_C^* \times \mathcal{P}(N_R) \rightarrow \mathcal{P}(N_R)$ with

  $deadConsAbs_R(ns_C, Q) :=$
  $\{n \in N_R \mid ns_C(n) \in \{\texttt{Null}, \texttt{Started}\} \ \wedge \ \exists l \in Q : \ n \in predConsAbs^*(l)\}$

  is a function assigning a set of `ConsAbs` predecessor nodes that have not been completed yet in `ConsStateMark` ns_C to a set of nodes Q. These `ConsAbs` nodes are regarded as dead predecessors when the nodes in Q are started.

- $deadAnteAbs_R : NS_A^* \times \mathcal{P}(N_R) \rightarrow \mathcal{P}(N_R)$ with

  $deadAnteAbs_R(ns_A, Q) :=$
  $\{n \in N_R \mid ns_A(n) \in \{\texttt{Null}, \texttt{Started}\} \ \wedge \ \exists l \in Q : n \in predAnteAbs(l)\}$

  is a function assigning a set of `AnteAbs` predecessor nodes that have not been completed yet in `AnteStateMark` ns_A to a set of nodes Q. These `AnteAbs` nodes are regarded as dead predecessors when the nodes in Q are started.

7.3.1.5. Node execution semantics for START and END events

Assuming that all preconditions for executing a node over a START or an END event apply, the question is how the node's state has to be adapted. The execution semantics of processes cannot be adopted as we aim at the automatic detection of event patterns that match with the patterns defined in the CRG. Therefore, the different node types of CRGs have to be considered in order to correctly implement the designated semantics. The active and passive state transitions[4] of CRG nodes are illustrated in Fig. 7.10. In the following, we elaborate on the execution semantics of CRG nodes for START and END events based on the example provided in Fig. 7.11. The latter shows the `MarkStructure`s of different compliance states when processing events of an execution trace.

Dealing with START events How a CRG node's state (i.e., execution state represented by ns and associated events represented by nl) in a `MarkStructure` is altered over a matching START event depends on the particular node type.

`AnteOcc` nodes For a CRG node n in state `Null` and a matching START event e, two situations have to be accounted for: Event e may contribute to an event pattern that matches with the antecedent pattern of the CRG. To account for that case, n should become `Started` over e. There

[4] A passive state change results from marking not completed predecessor nodes as `NotExecuted` and is indicated by the dashed line in Fig. 7.10.

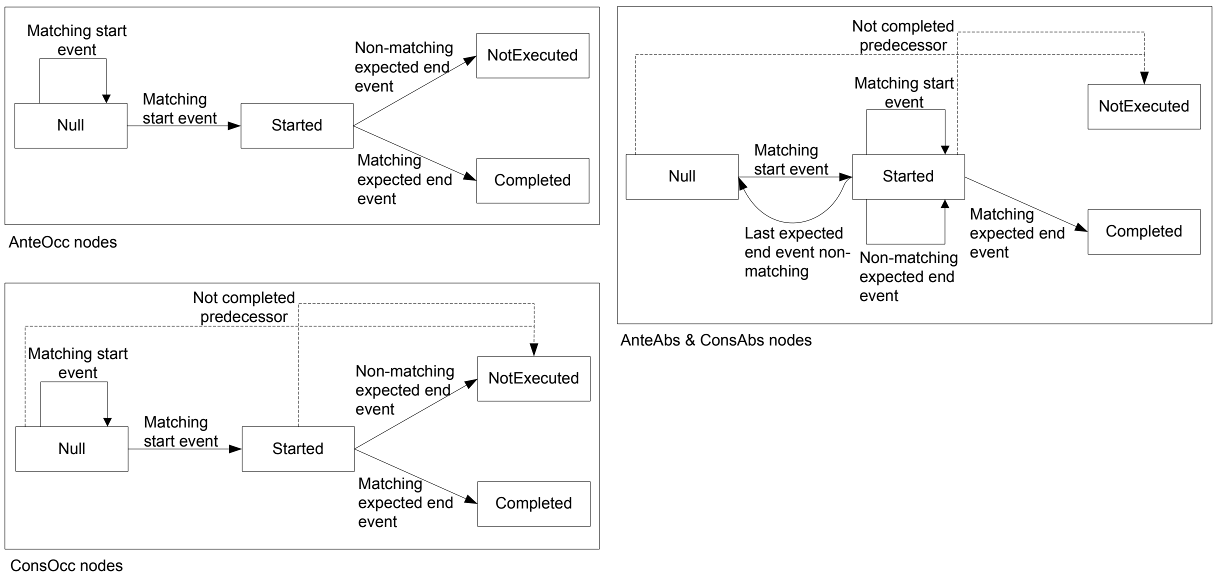


Figure 7.10.: Execution states of CRG nodes and corresponding transitions

is, however, also the possibility of e not contributing to the CRG's antecedent pattern but some future matching START may will. To account for that case, n should remain in state NULL in order to be started over future matching START events. As both these cases should be accounted for, we explore both options for each ANTEOCC node n that can be started over a START event. This results in a set of MARKSTRUCTUREs, one for each option. This nondeterministic execution semantics (as reflected in the diagram in Fig. 7.10) implements the $\forall$ semantics of ANTEOCC nodes.

Example 7.10 (Execution semantics of ANTEOCC nodes for START events):
Consider MARKSTRUCTURE ms_1 in Fig. 7.11. Then, when START event e_1 is observed, ANTEOCC node r_1 becomes executable. This results in two MARKSTRUCTUREs, namely ms_1 and ms_2. While ms_2 represents the attempt to form the CRG's antecedent pattern using the just observed e_1 (i.e., r_1 becomes STARTED over e_1), ms_1 provides for the detection of later occurrences of A in the trace (i.e., r_1 does not become STARTED over e_1). After processing e_1, the compliance state is represented by ms_1 and ms_2, which are ACTIVATABLE (cf. Section 7.2.2). Thus, no activation of the CRG is detected yet.

CONSOCC nodes A matching START event can only affect CONSOCC nodes that are still in state NULL. Despite their $\exists$-semantics (cf. Section 6.3), starting CONSOCC nodes over a matching START event can be insufficient to detect the consequence pattern of a CRG in an execution trace. One reason for this is that we cannot tell whether the expected END event is going to match a STARTED CONSOCC node due to end conditions. Another reason is that the first START event matching a CONSOCC node n may not belong to an activity execution that leads to an occurrence of the CRG's consequence pattern (cf. Example 7.11). That is why CONSOCC nodes are started nondeterministically like ANTEOCC nodes. As a result of executing a set of CONSOCC

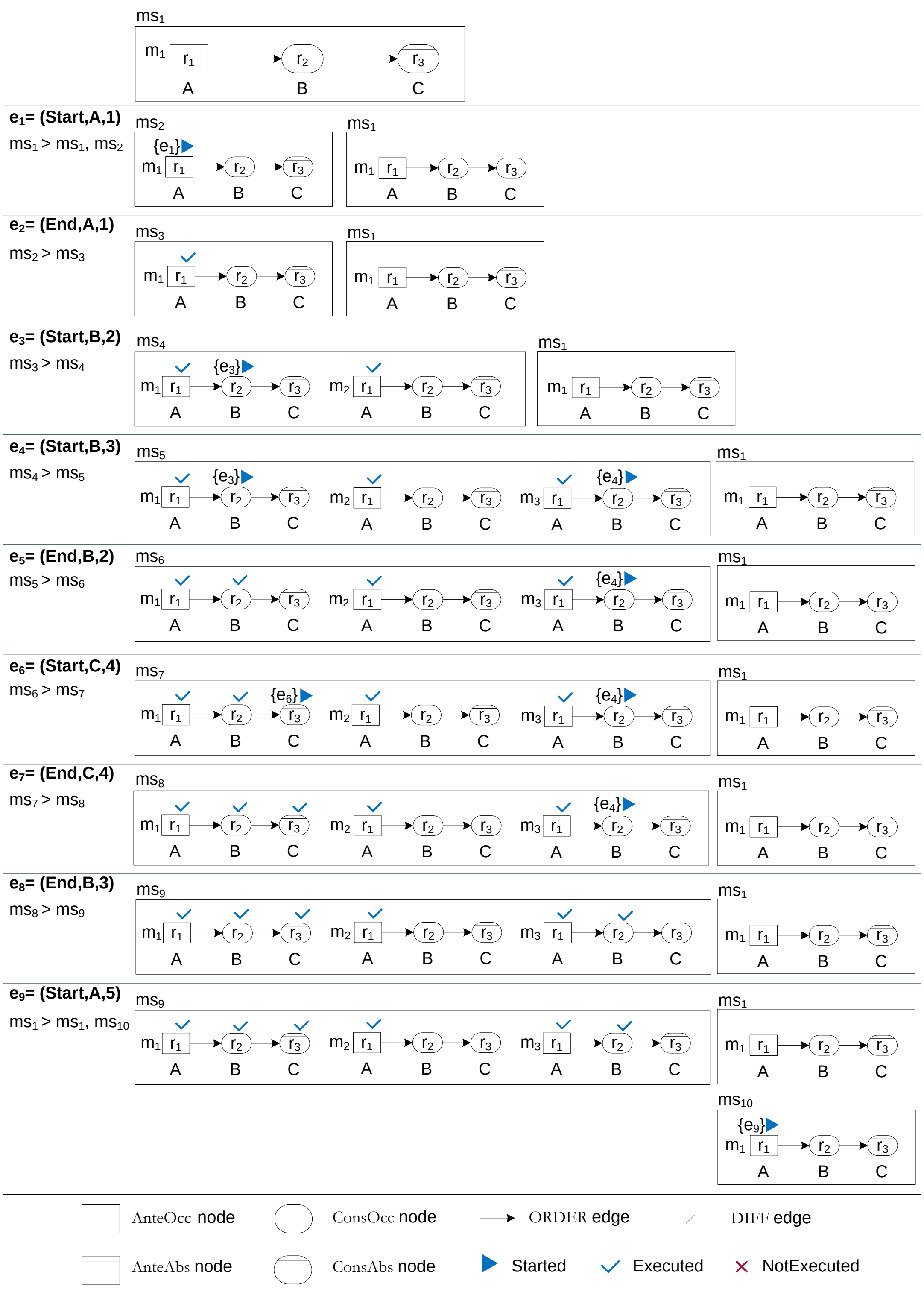


Figure 7.11.: Application of the execution semantics

nodes in a ConsExMark of a MarkStructure, we receive a set of resulting ConsExMarks each of which represents one attempt to identify the consequence pattern in the execution trace.

Example 7.11 (Execution semantics of ConsOcc nodes for START events):
Consider ms_3 in Fig. 7.11. When e_3, the start of B, is observed, ConsOcc node r_2 of ms_3's ConsExMark becomes executable. As a result, we obtain two ConsExMarks in the resulting MarkStructure ms_4. When observing another start of B as in e_4, r_2 is still executable in the ConsExMark of m_2 of ms_4. Processing e_4, therefore, yields ms_5 containing three ConsExMarks: one in which r_2 is not started yet and two in which r_2 is started for e_3 and e_4, respectively[5].

AnteAbs nodes are utilized to model the absence of activity executions within a CRG's antecedent pattern (i.e., $\neg\exists$-semantics, cf. Section 6.3). When being in state Null, the AnteAbs node will be put into state Started when the first matching START event is observed. To implement the semantics of AnteAbs nodes, we collect *all* subsequent matching START events (recorded by the *nl* function of AnteExMarks) as long as the AnteAbs node is not yet Completed. This is because each of these START events may result in a matching activity execution. The semantically corresponds to starting multiple instances of the AnteAbs node, each of which is associated with a single START event and each of which may result in a matching activity execution depending on the expected END events. If one of these node instances yields a matching activity execution, obviously the associated absence constraint does not hold. This is why we also refer to this as starting an AnteAbs node (even though technically the node is already Started).

Example 7.12 (Execution semantics of AnteAbs nodes over START events):
Fig. 7.12 depicts the state changes of AnteAbs node r_1 when being executed over an execution trace containing two interleaved executions of activity A. We assume that the START events e_1 and e_2 both match r_1. Through processing the first START event e_1, r_1 is put into state Started. Since subsequent e_2 also matches r_1, e_2 is also recorded as the START event of a possibly matching activity execution. As a result, r_1 can be completed by two expected END events.

ConsAbs nodes are utilized to model the absence of activity executions within a CRG's consequence pattern (i.e., $\neg\exists$-semantics, cf. Section 6.3). Hence, the firing behavior of AnteAbs nodes over START events can be adopted for ConsAbs nodes.

Example 7.13 (Execution semantics of ConsAbs nodes for START events):
Consider ms_6 in Fig. 7.11. Then, ConsAbs node r_3 of m_1 becomes executable over e_6. This results in ms_7. If now a further START event matching r_3 would be recorded, r_3 would be associated with both START events in analogy to AnteAbs nodes.

[5] As discussed in Section 7.3.1.3, it is necessary to record associated START events in order to be able to identify corresponding expected END events. In this case, each Started occurrence node is only associated with a single START event. For implementation of the concepts introduced, one may also think of alternative approaches.

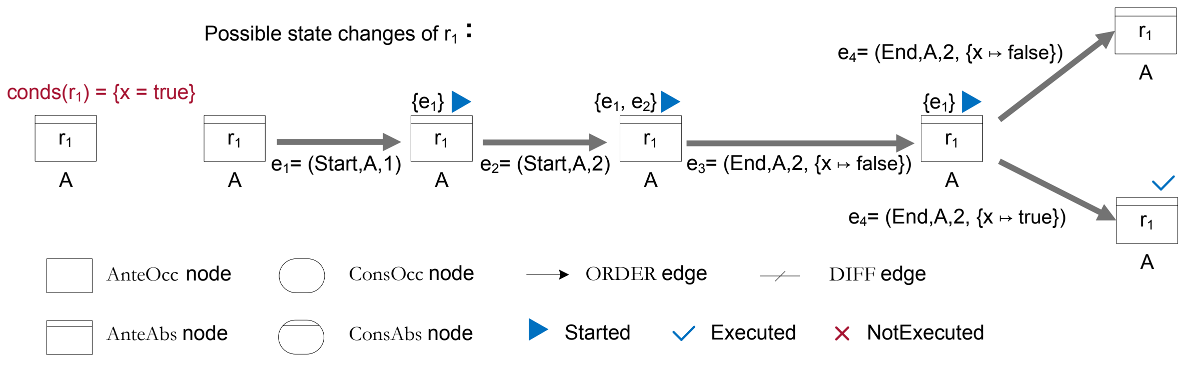


Figure 7.12.: Firing of AnteAbs nodes

Dealing with END events An observed END event is only relevant to STARTED CRG nodes of a MARKSTRUCTURES (cf. Fig. 7.10). However, an END event will only be relevant to a STARTED node, if it corresponds to one of the START events associated with the STARTED node (cf. Section 7.3.1.3). When observing such an expected END event for a STARTED CRG node n, there are different scenarios to be taken into account depending on n's node type.

Occurrence nodes A STARTED occurrence node n is only associated with a single START event. Thus, only a single END event is expected and only a single END event can lead to completion of n. If the expected END event matches n, n will become COMPLETED. Otherwise n will be put into state NOTEXECUTED as no further END event is expected[6].

Example 7.14 (Execution semantics of occurrence nodes over END events):
Consider for example ms_5 in Fig. 7.11. Then, END event e_5 is the expected END event of START event e_3, which is associated with r_2 of EXMARK m_1 of ms_5. Since e_5 matches r_2, r_2 becomes COMPLETED, which results in ms_6. If, however, e_5 would not match r_2, r_2 would be assigned execution state NOTEXECUTED.
While r_2 is also STARTED in m_3 of ms_5, e_5 does not lead to completion of r_2 in m_3 as it is not the expected END event of e_4.

Absence nodes As a STARTED absence node n can be associated with multiple START events, it may be expecting multiple END events. If an expected END event e matches n, n will become COMPLETED. If an expected END event, however, does not match n, there are two scenarios. If e is the last expected END event, clearly no further END event will be able to correctly complete the execution of n. This semantically corresponds to the situation of n not being started at all. Apparently, n's current execution state STARTED becomes obsolete as all started start-end-transactions associated with n are already dismissed. That is why the execution state of n is changed back to NULL in this case.

[6]Note that it is also possible to set the node's execution state back to NULL. However, as upon starting an occurrence node, not yet executed predecessors may become discarded (cf. Section 7.3.1.4), other node markings may have to be undone in this case. Therefore, assigning execution state NOTEXECUTED is the more elegant solution. To prevent that this results in "dead" structures causing unnecessary costs, pruning rules can be applied to discard such MARKSTRUCTURES (cf. Section 7.5.1)

If e is not the only expected END event as n is still associated with multiple START events, future END events can still lead to completion of n. Hence, n will remain STARTED. In order to prevent completing n over a second occurrence of e that does not belong to the same activity execution, we have to undo the association of n and the corresponding START event. In the marking rules, this is done by removing the START event from the set of START events associated with n.

Example 7.15 (Execution semantics of absence nodes over END events):
Consider, for example, STARTED ANTEABS node r_1 associated with e_1 and e_2 in Fig. 7.12. Then, END event e_3, the expected END event of e_2 is observed next. As e_3 turns out to not lead to a matching activity execution (due to the end condition $x = true$), e_2 is removed from the set of associated START events to indicate that now only the END event of e_1 is still expected. If subsequent e_4, the expected END event of e_1, matches r_1, a matching activity execution is recorded and r_1 will, thus, become COMPLETED. If, however, e_4 turns out to not match r_1, r_1 will be put back into state NULL since no further END events are expected.

7.3.1.6. Firing conflicts between ordered nodes

As ANTEOCC nodes and CONSOCC nodes can be started without all predecessors having to be completed, it is possible to run into *firing conflicts* between ordered CRG nodes as illustrated in Example 7.16. Firing conflicts between ordered nodes occur when two ordered nodes become executable over the same START event at the same time. Due to their ordering relation, executing such nodes over the same START event would lead to corrupt MARKSTRUCTUREs with respect to CRG and MARKSTRUCTURE semantics.

Example 7.16 (Firing conflict between ordered nodes):
Consider MARKSTRUCTURE ms_1 and observed START event e_1 to be processed depicted in Fig. 7.13 a). Assuming that e_1 matches both r_2 and r_3, both nodes can be started over e_1[7]. This situation is referred to as *firing conflict* between ordered nodes as apparently, starting both r_2 and r_3 over the same START event is inconsistent to CRG semantics as ordered nodes must not be associated with the same activity execution.

As Example 7.16 shows, situations in which ordered nodes can be started concurrently ask for precise arrangements. In order to prevent starting two ordered nodes over the same event, we prioritize the execution of certain node types instead of processing all nodes at once. In order to implement the formal semantics, the prioritization follows the hierarchy of CRG nodes (cf. Fig. 7.6 in Section 7.3.1.2). Thus, ANTEOCC nodes are processed first in each execution iteration. This results in a set of MARKSTRUCTUREs as ANTEOCC nodes are started nondeterministically. As described in Section 7.3.1.4, not yet completed predecessors of STARTED ANTEOCC nodes are marked as NOTEXECUTED. Thus, nodes that have been identified as startable at the beginning of the execution iteration might get marked as no longer executable. Such nodes consequently

[7]Note that r_2 can be started independently from r_3's execution state.

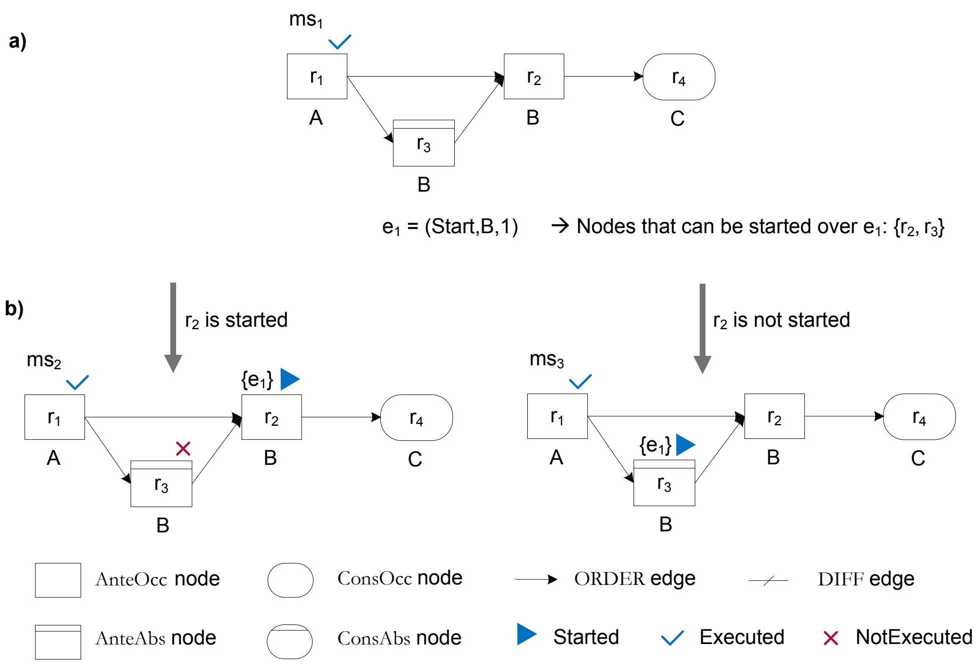


Figure 7.13.: A firing conflict between CRG nodes and its resolution

cannot be started any longer in order to stick to CRG formal semantics. After processing ANTEOCC nodes, for each obtained MARKSTRUCTURE still executable ANTEABS nodes are processed. Then, still executable consequence nodes are processed. As starting CONSOCC nodes can discard not yet completed CONSABS predecessors, CONSOCC nodes have to be processed prior to CONSABS nodes. As CONSOCC nodes are started nondeterministically, each CONSEXMARK of an obtained MARKSTRUCTURE may result in a set of CONSEXMARKS. Finally, for each thus obtained CONSEXMARK executable CONSABS nodes that have not been marked with NOTEXECUTED in the meantime can be processed. By applying the described procedure, marking not yet completed predecessors as NOTEXECUTED (cf. Section 7.3.1.4) prevents starting ordered nodes over a START event in the same iteration and, thus, ensures that ordered nodes are not associated with the same activity execution.

Example 7.17 (Step-wise processing CRG nodes in an execution iteration):
The CRG from Fig. 7.13 expresses that for every pair of A and the next subsequent B, an execution of C is required after B. Fig. 7.13 b) shows how prioritization of node types resolves firing conflicts. As r_2 is an ANTEOCC node, it is processed first in the iteration, which results in two MARKSTRUCTURES ms_2 and ms_3. As described in Section 7.3.1.5, ANTEOCC nodes are started nondeterministically. For the case that r_2 is started over e_1, ANTEABS node r_3 will be marked as NOTEXECUTED as shown in MARKSTRUCTURE ms_2. Hence, r_3 can no longer be started over e_1 in the same iteration. Indicating that an execution of A followed by a start of B (without another execution of B in between) was witnessed, ms_2 correctly reflects the observed process behavior so far. For the case that r_2 is not started over e_1, r_3 remains startable and, hence, will be started resulting in ms_3. When observing END event of e_1, r_3 in ms_3 will become COMPLETED. Thus, ms_3 will become DEACTIVATED (cf. Def. 7.5), which follows the CRG semantics as the

first execution of B after A would already have been observed.

7.3.1.7. Firing conflicts between unordered nodes

So far, we have not yet discussed the effect of DIFF edges on the execution of CRG nodes. According to the CRG semantics, two nodes connected through a DIFF edge must not be executed over the same activity execution. This has to be be provided for by the CRG operational semantics (cf. Example 7.18). As nodes connected through a DIFF edge must not be in any ordering relation (cf. Section 6.2.4), they can become executable over the same START events.

Example 7.18 (The effect of DIFF edges on CRG execution):
Fig. 7.14 depicts an antecedent pattern that will become activated if only one execution of A is present in an execution trace. When observing START event e_1, both r_1 and r_2 are startable. Due to prioritization, AnteOcc node r_1 is processed first and is started nondeterministically. MarkStructure ms_2 is what we receive after starting r_1. Clearly, r_2 must not be started over e_1 in ms_2 due to the imposed DIFF relation. This will be ensured by the execution rules. When r_1 is not started over e_1, r_2 can be started, which results in ms_3.

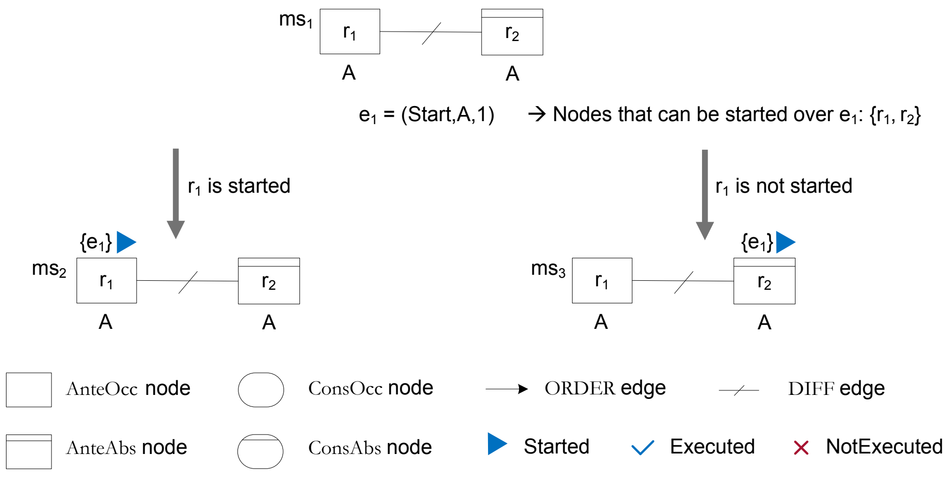


Figure 7.14.: Effect of DIFF edges on CRG execution

Since execution of MarkStructures over END events is deterministic, it suffices to ensure that nodes connected through DIFF edges are not started over the same START event within an execution iteration. The execution rules will ensure that.

7.3.1.8. Summary

In Section 7.3.1, we discussed the various aspects to be considered for operationalizing CRGs. Recall Fig. 7.1. Then, we can now provide an algorithm for executing a CRG over a trace.

The algorithm in Listing 1 executes a CRG R over an execution trace σ with $executeStart_R$ / $executeEnd_R$ being the function applying execution and marking rules to MarkStructure ms when processing a START/ an END event, respectively. We will describe these functions in detail in Section 7.3.2 and 7.3.3. The execution starts with the initialization of a MarkStructure of R reflecting the initial compliance state. For each observed event, the current MarkStructures are updated using execution and marking rules. When end of trace is reached, a further marking rule is applied to put all obtained MarkStructures into FINAL state (cf. Def. 7.5). At each stage, the compliance notions introduced in Section 7.2.2 can be applied to evaluate the MarkStructures that constitute the compliance state.

Algorithm 1 Executing CRG R over execution trace σ ($verify(R, \sigma)$)

```
1: $R$ is a CRG
2: $\sigma = <e_1, \ldots, e_n>$ is an execution trace
   {INITIALIZATION}
3: $ms_0 = init(R)$;
4: $MS = \{ms_0\}$;
   {ITERATION}
5: while $\sigma$.length $> 0$ do
6:   $e = \sigma[0]$;
7:   $MS_{new} = \emptyset$;
8:   if $e$ is a START event then
9:     for all $ms \in MS$ do
10:      $MS_{new} = MS_{new} \cup$ $executeStart_R(ms, e)$;
11:    end for
12:  else if $e$ is an END event then
13:    for all $ms \in MS$ do
14:      $MS_{new} = MS_{new} \cup$ $executeEnd_R(ms, e)$;
15:    end for
16:  end if
17:  $\sigma = \sigma \setminus \sigma[0]$;
18:  $MS = MS_{new}$;
19: end while
   {FINALIZATION}
20: $MS_{new} = \emptyset$;
21: for all $ms \in MS$ do
22:   $MS_{new} = MS_{new} \cup$ $markEnd_R(ms)$;
23: end for
24: return $MS_{new}$;
```

As described in Section 7.3.1.6, each execution iteration consists of four consecutive steps: execution of AnteOcc, of AnteAbs, of ConsOcc, and of ConsAbs nodes. To formalize CRG operational semantics, we introduce execution and marking rules for each node type. Execution rules identify executable nodes in an iteration and choose options to be explored while marking rules adapt node markings accordingly. To keep the execution and marking rules easily intelligible, we opted for defining them over ExMarks instead of over MarkStructures. This is possible as a MarkStructure is constituted by a set of ExMarks. As a result, the obtained ExMarks have to be aggregated to resulting MarkStructures at the end of the iteration. Table 7.2 summarizes the execution and marking rules that will be discussed in detail in Sections 7.3.2, 7.3.3, and 7.3.4. We will also introduce rules for executing CRGs over EX events as optimization in Section 7.5.2.

	AnteOcc	AnteAbs	ConsOcc	ConsAbs
START events	MR1, ER1	MR2, ER2	MR3, ER3	MR4, ER4
END events	MR5, ER5	MR6, ER6	MR7, ER7	MR8, ER8
EOT	MR9			

Table 7.2.: Execution and marking rules for processing START and END events and the end of a trace (EOT)

7.3.2. Execution and marking rules for START events

Based on the considerations described in Section 7.3.1, Fig. 7.15 illustrates how a MarkStructure is altered when processing a START event. As AnteOcc nodes are executed nondeterministically over START events, which results in a set of child MarkStructures. As predecessors may no longer be executable, not only the AnteExMark of a MarkStructure is affected by the start of AnteOcc nodes but the ConsExMarks may also be altered. AnteAbs nodes are executed deterministically and alter the MarkStructure's AnteExMark. Similar to AnteOcc nodes, ConsOcc nodes are started nondeterministically. In consequence, the set of ConsExMarks of a MarkStructure will be altered when starting ConsOcc nodes. Ultimately, ConsAbs nodes are executed altering the ConsExMarks of a MarkStructure.

Algorithm 2 defines *executeStart*, which processes a MarkStructure over a START event (cf. Algorithm 1). As the execution rules are defined over ExMarks instead of over MarkStructures for brevity reasons, the obtained ExMarks have to be aggregated to MarkStructures at the end of the iteration. Execution rules are applied consecutively in lines 7 to 17. As pointed out in Section 7.3.1.6, the processing of CRG nodes follows a particular order in order to prevent firing conflicts between ordered nodes. The functions to execute the particular node types are introduced in Sections 7.3.2.2 to 7.3.2.5, respectively. The nondeterministic start of AnteOcc and ConsOcc nodes as discussed in Section 7.3.1.5 shows in lines 8 and 11, respectively, as the functions *executeAnteOccStart* and *executeConsOccStart* each returns a set of ExMarks. The obtained ExMarks are finally aggregated to MarkStructures in line 18.

As discussed in Section 7.3.1.3, the START events associated to a node must be memorized in order to be able to determine whether an observed END event belongs to an activity execution a node was started on. To capture the associated events, we use the function *nl* of ExMarks, which assigns a (possibly empty) set of events[8] to each CRG node. The marking rules for dealing with START events will, therefore, set / alter the *nl*-property for executed nodes.

[8]Note that absence nodes can be associated with multiple START events

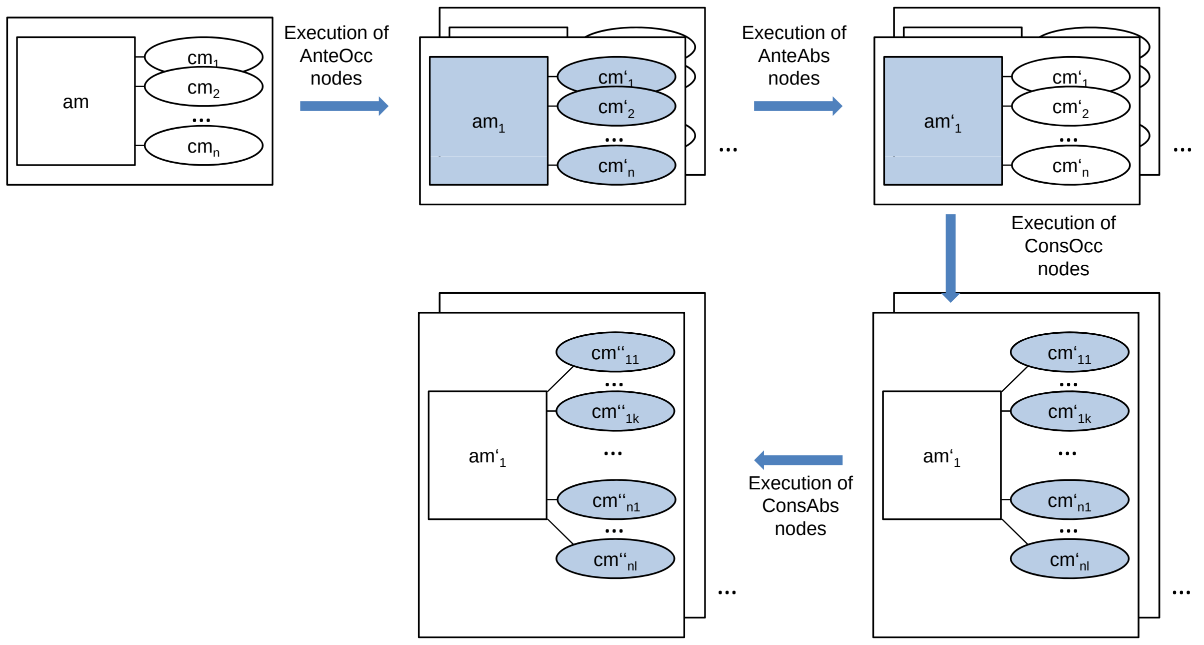

am
cm1
cm2
cmn
Execution of AnteOcc nodes
am1
cm'1
cm'2
cm'n
Execution of AnteAbs nodes
am'1
Execution of ConsOcc nodes
cm'11
cm'1k
cm'n1
cm'nl
Execution of ConsAbs nodes
cm"11
cm"1k
cm"n1
cm"nl

Figure 7.15.: Alternation of a MarkStructure when processing a START event

In the following, Section 7.3.2.1 discusses the necessary conditions for executing a CRG node over a START event before execution and marking rules for the particular node types are introduced.

7.3.2.1. Necessary conditions

Based on the considerations in Section 7.3.1, the necessary conditions for executing a CRG node over a START event incorporate conditions in different respects:

1. matching between the node and the START event
2. the node's current execution state
3. the execution state of the node's predecessors

Def. 7.8 formalizes the necessary conditions. The matching between the CRG node and the START event is covered by condition (i) of the functions in Def. 7.8. While occurrence nodes are only executable over START events when being in state Null, absence nodes are executable over a START event even if they are already Started (cf. Section 7.3.1.5). In this case, observation of a further matching START event will not lead to a different execution state but will lead to a changed set of START events associated with the node. This is reflected in condition (ii) of the functions in Definition 7.8. The conditions (iii) and (iv) reflect the conditions regarding a node's predecessors as summarized by Table 7.1 in Section 7.3.1.2. Common to all four node types is that they can only be started if AnteOcc predecessors have been completed. Additionally, ConsOcc and ConsAbs nodes cannot be started until their ConsOcc predecessors have been completed.

Algorithm 2 Executing a MarkStructure over a START event ($executeStart_R(ms, e)$)

```
1: $R = (A, C)$ is a CRG;
2: $e \in E^{Start}$ is a START event;
3: $ms = ((ns_A, nl_A), \{(ns_C^1, nl_C^1), \dots, (ns_C^k, nl_C^k)\}) \in MS_R^*$ is a MarkStructure of $R$;
4: $M_{ms} = \{m_1 = ((ns_A, nl_A), (ns_C^1, nl_C^1)), \dots, m_k = ((ns_A, nl_A), (ns_C^k, nl_C^k))\}$ is the set of
   ExMarks of $ms$

   {INITIALIZATION}
5: $M_{AnteOcc}, M_{ConsOcc}, M_{Res} = \emptyset$;
6: $MS_{Res} = \emptyset$

   {ITERATION}
7: for all $m \in M_{ms}$ do
8:   $M_{AnteOcc} = executeAnteOccStart_R(m, e)$;
9:   for all $m \in M_{AnteOcc}$ do
10:    $m_{AnteAbs} = executeAnteAbsStart_R(m, e)$;
11:    $M_{ConsOcc} = executeConsOccStart_R(m_{AnteAbs}, e)$;
12:    for all $m \in M_{ConsOcc}$ do
13:      $m_{ConsAbs} = executeConsAbsStart_R(m, e)$;
14:      $M_{Res} = M_{Res} \cup \{m_{ConsAbs}\}$;
15:    end for
16:  end for
17: end for

   {Aggregation of obtained ExMarks to MarkStructures }
18: $MS_{res} = aggregate_R(M_{Res})$;

   {The obtained set of resulting MarkStructures is returned}
19: return $MS_{res}$;
```

Definition 7.8 (Necessary conditions for executing CRG nodes over START events) Let $R = (A, C)$ be a CRG and $e \in E^{Start}$ be a START event. Then,

- $executableAnteStart_R : NS_A^* \times N_A \times E^{Start} \to \mathbb{B}$
  is a function determining whether an antecedent node n is executable under an AnteStateMark ns_A of R over a START event e with:

  $\forall n \in N_A$ with $nt_R(n) =$ AnteOcc:

$$executableAnteStart_R(ns_A, n, e) := \begin{cases} true,\ if \\ \ (i)\ matchStart_R(n, e) = true \land \\ \ (ii)\ ns_A(n) = \texttt{Null} \land \\ \ (iii)\ \forall l \in predAnteOcc(n) : ns_A(l) = \texttt{Completed} \\ false,\ otherwise. \end{cases}$$

$\forall n \in N_A$ with $nt_R(n) = \texttt{AnteAbs}$:

$$executableAnteStart_R(ns_A, n, e) := \begin{cases} true,\ if \\ \ (i)\ matchStart_R(n, e) = true\ \land \\ \ (ii)\ ns_A(n) \in \{\texttt{Null}, \texttt{Started}\}\ \land \\ \ (iii)\ \forall l \in predAnteOcc(n) : ns_A(l) = \texttt{Completed} \\ false,\ otherwise. \end{cases}$$

- $executableConsStart_R : NS_R^* \times N_C \times E^{Start} \to \mathbb{B}$

  is a function determining whether a consequence node n is executable under a $\texttt{StateMark}$ (ns_A, ns_C) of R over a START event e with:

  $\forall n \in N_C$ with $nt_R(n) = \texttt{ConsOcc}$:

$$executableConsStart_R((ns_A, ns_C), n, e) := \begin{cases} true,\ if \\ \ (i)\ matchStart_R(n, e) = true\ \land \\ \ (ii)\ ns_C(n) = \texttt{Null}\ \land \\ \ (iii)\ \forall l \in predAnteOcc(n) : \\ \quad ns_A(l) = \texttt{Completed}\ \land \\ \ (iv)\ \forall l \in predConsOcc(n) : \\ \quad ns_C(l) = \texttt{Completed} \\ false,\ otherwise. \end{cases}$$

  $\forall n \in N_C$ with $nt_R(n) = \texttt{ConsAbs}$:

$$executableConsStart_R((ns_A, ns_C), n, e) := \begin{cases} true,\ if \\ \ (i)\ matchStart_R(n, e) = true\ \land \\ \ (ii)\ ns_C(n) \in \{\texttt{Null}, \texttt{Started}\}\ \land \\ \ (iii)\ \forall l \in predAnteOcc(n) : \\ \quad ns_A(l) = \texttt{Completed}\ \land \\ \ (iv)\ \forall l \in predConsOcc(n) : \\ \quad ns_C(l) = \texttt{Completed} \\ false,\ otherwise. \end{cases}$$

- $exAnteNodesStart_R : NS_A^* \times E^{Start} \times \{\texttt{AnteOcc}, \texttt{AnteAbs}\} \to \mathcal{P}(N_R)$

  is a function determining the set of antecedent nodes of type t that are executable under an $\texttt{AnteStateMark}$ ns_A over START event e with:
  $exAnteNodesStart_R(ns_A, e, t) :=$
  $\{n \in N_R \mid nt_R(n) = t\ \land\ executableAnteStart_R(ns_A, n, e) = true\}$.

- $exConsNodesStart_R : NS_R^* \times E^{Start} \times \{\texttt{ConsOcc}, \texttt{ConsAbs}\} \to \mathcal{P}(N_R)$

  is a function determining the set of consequence nodes of type t that are executable under an $\texttt{StateMark}$ (ns_A, ns_C) over START event e with:
  $exConsNodesStart_R((ns_A, ns_C), e, t) :=$
  $\{n \in N_R \mid nt_R(n) = t\ \land\ executableConsStart_R((ns_A, ns_C), n, e) = true\}$.

Example 7.19 (CRG nodes executable over a START event):
In Fig. 7.16, r_1 in m_1 is executable when observing a start of A since ANTEOCC nodes can be started independently from predecessors of other node types. For the same reason, both r_2 and r_3 are executable in EXMARK m_2 when the start of B is observed[9]. In contrast, no node becomes executable when observing the start of D. This is because r_5 must not be started until all ANTEOCC predecessors have been completed. In m_3, r_2 and r_3 are executable when the start of B is observed. This is because ANTEABS nodes can be executed over START events when already in state STARTED. In m_4, r_2 can be started when observing the start of B. In contrast, no node becomes startable when observing the start of C.

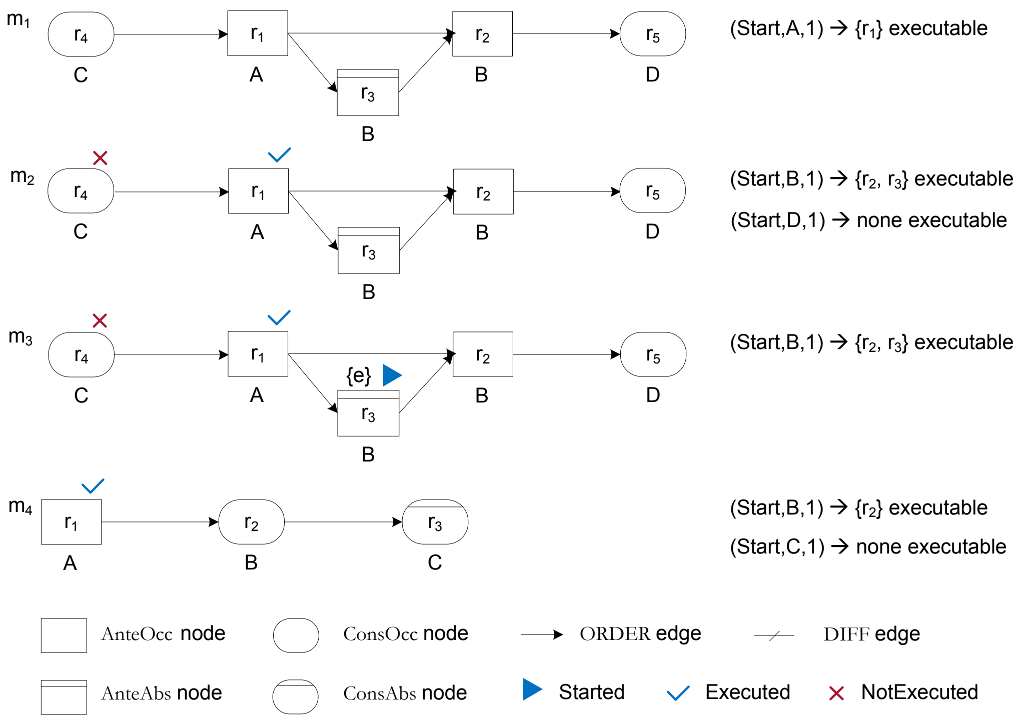

m1
r4
C
r1
A
r3
B
r2
B
r5
D
(Start,A,1) → {r1} executable
m2
(Start,B,1) → {r2, r3} executable
(Start,D,1) → none executable
m3
{e}
(Start,B,1) → {r2, r3} executable
m4
(Start,B,1) → {r2} executable
(Start,C,1) → none executable
AnteOcc node
ConsOcc node
ORDER edge
DIFF edge
AnteAbs node
ConsAbs node
Started
Executed
NotExecuted

Figure 7.16.: Necessary conditions for executing nodes over START events

7.3.2.2. Execution of ANTEOCC nodes

Based on the set of ANTEOCC nodes of an EXMARK that are executable over a START event, execution rule ER1 identifies sets of ANTEOCC nodes that are started in the iteration. Each set represents an attempt to match the antecedent pattern and the events in the trace and results in a child EXMARK. The determination of such sets follows some constraints:

[9]Note that this is a firing conflict between ordered nodes (cf. Section 7.3.1.6). We will later show in Fig. 7.17 how this is dealt with.

1. Following the execution semantics illustrated in Fig. 7.10 (cf. Section 7.3.1.5), two options have to be explored for each startable AnteOcc node.
2. No pair of AnteOcc nodes must be started at the same time if they are directly connected through a DIFF edge (cf. Section 7.3.1.7).

This results in the sets of AnteOcc nodes contained in Q^* in ER1. It contains exactly the subsets of executable AnteOcc nodes that are free of nodes directly related through DIFF edges. For each node set in Q^*, marking rule MR1 is applied to yield a child ExMark.

Execution Rule ER1 (Execution of AnteOcc nodes over START events):
Let $R = (A, C)$ be a CRG, $m = ((ns_A, nl_A), (ns_C, nl_C)) \in M_R^*$ be an ExMark of R, and $e \in E^{Start}$ be a START event with

- $QAO := exAnteNodesStart_R(ns_A, e, \texttt{AnteOcc})$ denoting the set of AnteOcc nodes of m satisfying the necessary execution conditions and
- $Q^* := \{Q \subseteq QAO \mid \forall n_1 \forall n_2 \in Q : (n_1, n_2) \notin DiffE_R \land (n_2, n_1) \notin DiffE_R\}$ denoting the set of subsets of QAO, where each $Q \in Q^*$ is free of nodes that are directly linked through a DIFF edge.

Then,

- $executeAnteOccStart_R : M_R^* \times E^{Start} \to 2^{M_R^*}$
  is a function assigning child ExMarks to an ExMark m and a START event e with:

  $executeAnteOccStart_R(m, e) := \bigcup_{Q \in Q^*} markAnteOccStart_R(m, Q, e)$.

MR1 puts AnteOcc nodes into state Started and alters the *nl*-property to associate the started nodes with the processed START event. Moreover, not yet completed predecessors of the started AnteOcc nodes are marked as NotExecuted using the functions *deadAnteAbs*, *deadConsOcc*, and *deadConsAbs* introduced in Section 7.3.1.4.

Marking Rule MR1 (Marking of AnteOcc nodes over START events):
Let $R = (A, C)$ be a CRG. Then,

$markAnteOccStart_R : M_R^* \times \mathcal{P}(N_R) \times E^{Start} \to M_R^*$

is a function assigning an ExMark m' to an original ExMark $m = ((ns_A, nl_A), (ns_C, nl_C))$, a set Q of AnteOcc nodes to be executed, and a START event e with

$markAnteOccStart_R(m, Q, e) := m' = ((ns'_A, nl'_A), (ns'_C, nl'_C))$ with:

- $\forall n \in N_R$ with $nt_R(n) = \texttt{AnteOcc}$:

$$(ns'_A(n), nl'_A(n)) := \begin{cases} (\textsc{Started}, \{e\}) \; if \; n \in Q \\ (ns_A(n), nl_A(n)), \; otherwise. \end{cases}$$

- $\forall n \in N_R$ with $nt_R(n) = \textsc{AnteAbs}$:

$$(ns'_A(n), nl'_A(n)) := \begin{cases} (\textsc{NotExecuted}, \emptyset) \; if \; n \in deadAnteAbs_R(ns_A, Q) \\ (ns_A(n), nl_A(n)), \; otherwise. \end{cases}$$

- $\forall n \in N_R$ with $nt_R(n) = \textsc{ConsOcc}$:

$$(ns'_C(n), nl'_C(n)) := \begin{cases} (\textsc{NotExecuted}, \emptyset) \; if \; n \in deadConsOcc_R(ns_C, Q) \\ (ns_C(n), nl_C(n)), \; otherwise. \end{cases}$$

- $\forall n \in N_R$ with $nt_R(n) = \textsc{ConsAbs}$:

$$(ns'_C(n), nl'_C(n)) := \begin{cases} (\textsc{NotExecuted}, \emptyset) \; if \; n \in deadConsAbs_R(ns_C, Q) \\ (ns_C(n), nl_C(n)), \; otherwise. \end{cases}$$

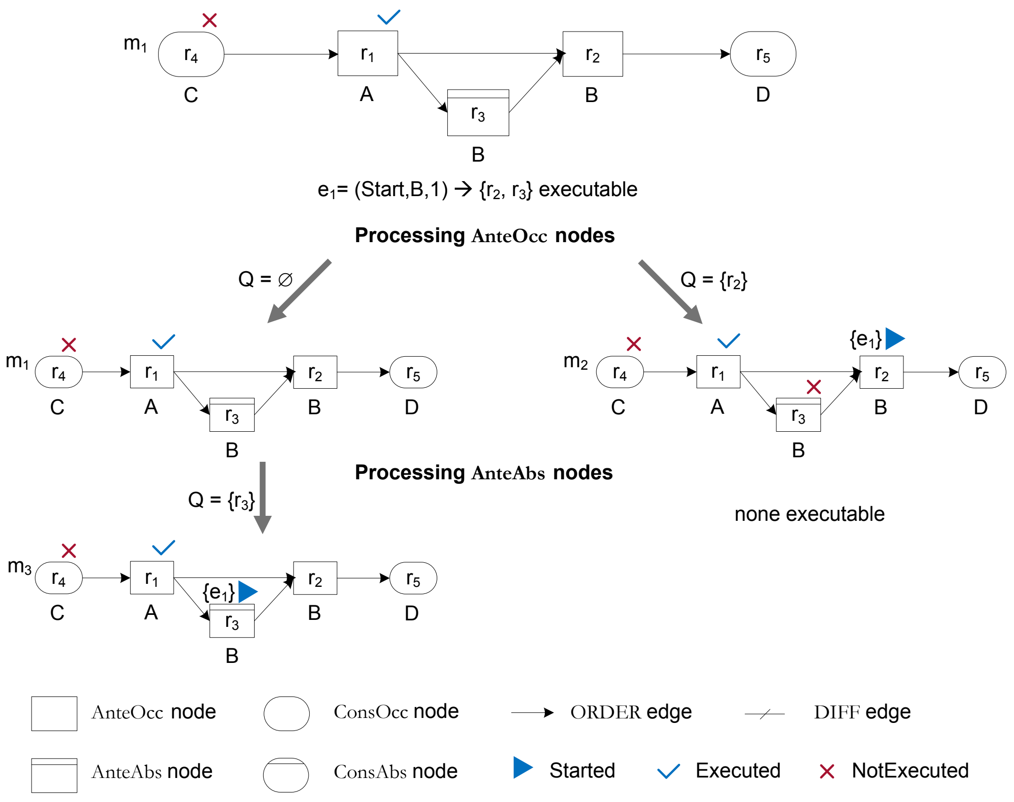


Figure 7.17.: Application of ER1 and MR1 and ER2 and MR2

Example 7.20 (Execution of $\textsc{AnteOcc}$ nodes over a $\textsc{start}$ event):
In Fig. 7.17, $\textsc{AnteOcc}$ node r_2 satisfies the necessary conditions when observing the start of B (i.e., $QAO = \{r_2\}$ in ER1). This results in $Q^* = \{\emptyset, \{r_2\}\}$ when applying ER1. Application

of MR1 for each set $Q \in Q^*$ results in two ExMarks, m_1 for $Q = \emptyset$ and m_2 for $Q = \{r_2\}$. In m_2, AnteAbs node r_3 is marked as NotExecuted as it is a not yet completed predecessor node of r_2. In contrast, in m_1, r_3 is still executable over e_1 and will be executed in the same iteration when AnteAbs nodes are processed in the next step.

Fig. 7.18 illustrates how AnteOcc nodes directly linked through a DIFF edge are dealt with. While both AnteOcc nodes r_1 and r_2 are executable over e_1, $\{n_1, n_2\}$ is not contained in Q^* when applying ER1. This prevents r_1 and r_2 from being executed over the same activity execution.

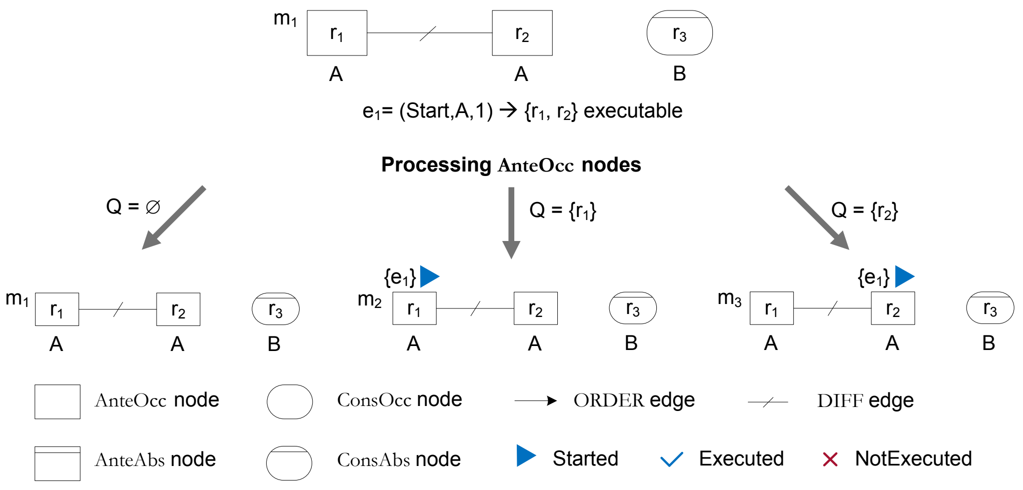


Figure 7.18.: Application of ER1 and MR1 when a DIFF edge is involved

7.3.2.3. Execution of AnteAbs nodes

In an execution iteration, AnteAbs nodes are processed after processing AnteOcc nodes. As a result, when processing AnteAbs nodes for an ExMark, the latter may contain AnteOcc nodes that have been started in the very iteration. To correctly implement the semantics of DIFF edges, startable AnteAbs nodes directly linked through DIFF edges to AnteOcc nodes that have been started in the very same execution iteration must not be executed (cf. Section 7.3.1.7). This is enforced in execution rule ER2. As AnteAbs nodes are executed deterministically (cf. Fig. 7.10), *executeAnteAbsStart* returns a single child ExMark.

Execution Rule ER2 (Execution of AnteAbs nodes over START events):
Let $R = (A, C)$ be a CRG, $m = ((ns_A, nl_A), (ns_C, nl_C)) \in M_R^*$ be an ExMark of R and $e \in E^{Start}$ be a START event with

- $QAA := exAnteNodesStart_R(ns_A, e, \text{AnteAbs})$ denoting the set of AnteAbs nodes of m satisfying the necessary execution conditions,
- $ExAnteOcc_m$ being the set of AnteOcc nodes that have been marked as Started in m over e in the same iteration,

- $D := \{n \in QAA \mid \exists l \in ExAnteOcc_m : (n,l) \in Diff_R \vee (l,n) \in Diff_R\}$ being the set of AnteAbs nodes in QAA that are directly linked to nodes in $ExAnteOcc_m$ through DIFF edges, and
- $Q := QAA \backslash D$ being the subset of QAA that is free of nodes that are directly linked through a DIFF edge to AnteOcc nodes executed in same iteration.

Then,

- $executeAnteAbsStart_R : M_R^* \times E^{Start} \rightarrow M_R^*$
  is a function assigning a child ExMark m' to an ExMark m and a START event e with:

  $executeAnteAbsStart_R(m,e) := markAnteAbsStart_R(m,Q,e)$.

Marking rule MR2 is used in ER2 to yield a child ExMark. It implements the considerations discussed in Section 7.3.1.5: An AnteAbs node to be executed will become Started if it is in execution state Null. If the node is already Started, it will remain so. The *nl*-property of the AnteAbs nodes is further altered to associate them with the processed event by adding the latter to the associated event set.

Marking Rule MR2 (Marking of AnteAbs nodes over START events):
Let $R = (A,C)$ be a CRG. Then,

$markAnteAbsStart_R : M_R^* \times \mathcal{P}(N_R) \times E^{Start} \rightarrow M_R^*$

is a function assigning an ExMark m' to an original ExMark $m = ((ns_A, nl_A), (ns_C, nl_C))$, a set Q of AnteAbs nodes to be executed, and a START event e with

$markAnteAbsStart_R(m,Q,e) := m' = ((ns'_A, nl'_A), (ns'_C, nl'_C))$ with:

- $\forall n \in N_R$ with $nt_R(n) = \text{AnteAbs}$:

$$(ns'_A(n), nl'_A(n)) := \begin{cases} (\text{Started}, \{e\}) \; if \; n \in Q \wedge ns_A(n) = \text{Null} \\ (ns_A(n), nl_A(n) \cup \{e\}) \; if \; n \in Q \wedge ns_A(n) = \text{Started} \\ (ns_A(n), nl_A(n)), \; otherwise. \end{cases}$$

- $\forall n \in N_R$ with $nt_R(n) = \text{AnteOcc}$:
  $(ns'_A(n), nl'_A(n)) := (ns_A(n), nl_A(n))$.
- $\forall n \in N_R$ with $nt_R(n) \in \{\text{ConsOcc}, \text{ConsAbs}\}$:
  $(ns'_C(n), nl'_C(n)) := (ns_C(n), nl_C(n))$.

Example 7.21 (Execution of AnteAbs nodes over a START event):
Fig. 7.17 shows an iteration where both AnteOcc and AnteAbs nodes become executable over a START event. Then, after processing AnteOcc nodes, AnteAbs node r_3 is still executable in m_1. Therefore, application of ER2 and MR2 yields m_3.

In ExMark m_1 from Fig. 7.19, AnteAbs node r_2 becomes executable when e_2 is observed. Application of ER2 and MR2 yields m_2, in which r_2 is associated with two started activity executions represented by the start events e_1 and e_2 as indicated by the updated nl-property of r_2.

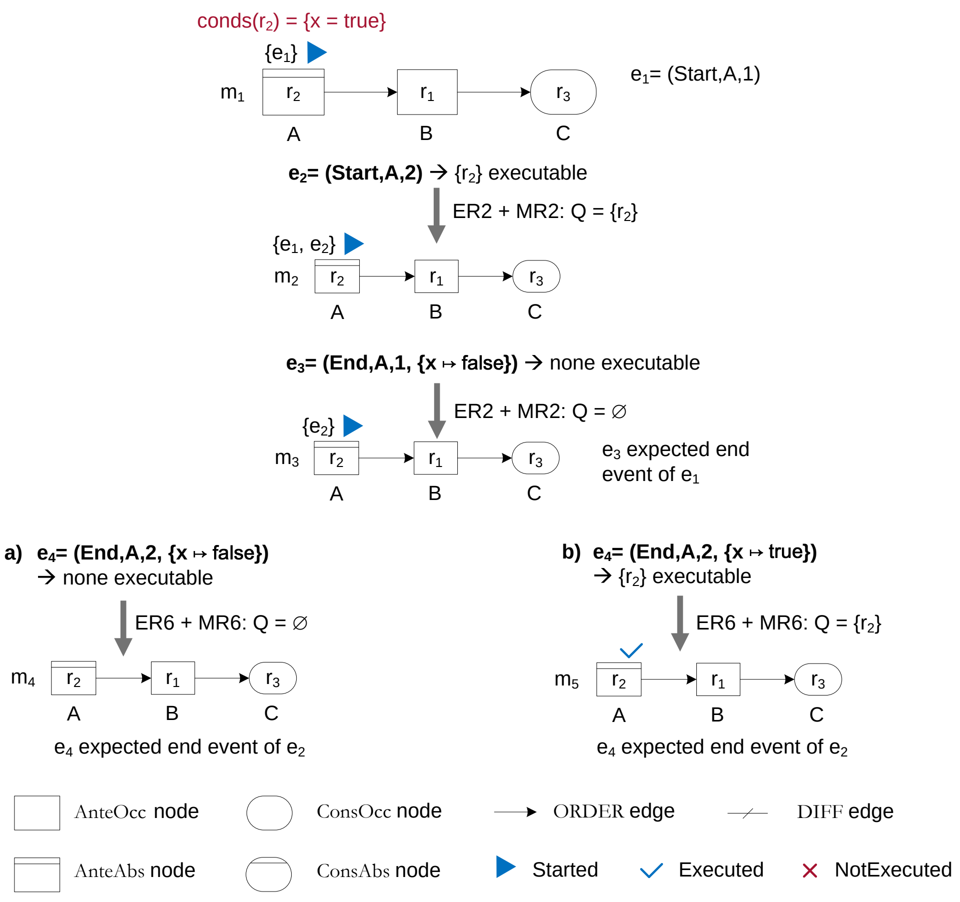

conds(r2) = {x = true}
{e1}
m1
r2
r1
r3
A
B
C
e1= (Start,A,1)
e2= (Start,A,2) → {r2} executable
ER2 + MR2: Q = {r2}
{e1, e2}
m2
e3= (End,A,1, {x ↦ false}) → none executable
ER2 + MR2: Q = ∅
{e2}
m3
e3 expected end event of e1
a) e4= (End,A,2, {x ↦ false}) → none executable
ER6 + MR6: Q = ∅
m4
e4 expected end event of e2
b) e4= (End,A,2, {x ↦ true}) → {r2} executable
ER6 + MR6: Q = {r2}
m5
e4 expected end event of e2
AnteOcc node
ConsOcc node
ORDER edge
DIFF edge
AnteAbs node
ConsAbs node
Started
Executed
NotExecuted

Figure 7.19.: Application of ER2 and MR2 and ER6 and MR6

7.3.2.4. Execution of ConsOcc nodes

As described in Section 7.3.1.5, ConsOcc nodes are started nondeterministically like AnteOcc nodes. The rationale behind this is to account for the case that the first START event matching the ConsOcc node does not lead to a matching activity execution (e.g., due to end conditions or to the corresponding END event appearing later than activity executions matching with successor nodes)[10].

Based on the set of ConsOcc nodes of an ExMark that satisfy the necessary execution conditions, execution rule ER3 identifies sets of ConsOcc nodes that are started in the iteration. For each

[10]For optimization, a ConsOcc node n may be started deterministically if it has neither end conditions nor successor nodes.

set, marking rule MR3 is invoked to yield a child ExMark. The determination of such sets follows some constraints:

1. Following the execution semantics illustrated in Fig. 7.10 (cf. Section 7.3.1.5), two options have to be explored for each startable ConsOcc node.
2. No pair of ConsOcc nodes must be started at the same time if they are directly linked to each other through a DIFF edge (cf. Section 7.3.1.7).
3. It has to be ensured that no ConsOcc node will be started in the iteration if it is directly linked through a DIFF edge to an AnteOcc node that was started in the same iteration. This is necessary in order to implement the semantics of DIFF edges.

These constraints are enforced by ER3 as Q^* contains exactly those subsets of executable ConsOcc nodes that are free of nodes that are directly linked to each other or to started AnteOcc nodes through DIFF edges.

Execution Rule ER3 (Execution of ConsOcc nodes over START events):
Let $R = (A, C)$ be a CRG, $m = ((ns_A, nl_A), (ns_C, nl_C)) \in M_R^*$ be an ExMark of R and $e \in E^{Start}$ be a START event with

- $QCO := exConsNodesStart_R((ns_A, ns_C), e, \texttt{ConsOcc})$ being the set of ConsOcc nodes of m satisfying the necessary execution conditions,
- $ExAnteOcc_m$ being the set of AnteOcc nodes that have been marked as Started in m over e in the same iteration,
- $D := \{n \in QCO \mid \exists l \in ExAnteOcc_m : (n, l) \in DiffE_R \vee (l, n) \in DiffE_R\}$ being the set of ConsOcc nodes in QCO that are directly linked to AnteOcc nodes executed in this iteration through DIFF edges, and
- $Q^* := \{Q \subseteq (QCO \backslash D) \mid \forall n_1 \forall n_2 \in Q : (n_1, n_2) \notin DiffE_R \wedge (n_2, n_1) \notin DiffE_R\}$ denoting the set of subsets of QCO, where each $Q \in Q^*$ is free of nodes that are directly linked to each other or to AnteOcc nodes executed in the same iteration through DIFF edges.

Then,

- $executeConsOccStart_R : M_R^* \times E^{Start} \rightarrow 2^{M_R^*}$
  is a function assigning child ExMarks to an ExMark m and a START event e with:

  $executeConsOccStart_R(m, e) := \bigcup_{Q \in Q^*} markConsOccStart_R(m, Q, e)$.

Marking rule MR3 takes care of marking ConsOcc nodes. In this process, not yet completed ConsAbs predecessors of started ConsOcc nodes will be marked as NotExecuted.

Marking Rule MR3 (Marking of ConsOcc nodes over START events):
Let $R = (A, C)$ be a CRG. Then,

$markConsOccStart_R : M_R^* \times \mathcal{P}(N_R) \times E^{Start} \to M_R^*$

is a function assigning an ExMark m' to an original ExMark $m = ((ns_A, nl_A), (ns_C, nl_C))$, a set Q of ConsOcc nodes to be executed, and a START event e with

$markConsOccStart_R(m, Q, e) := m' = ((ns'_A, nl'_A), (ns'_C, nl'_C))$ with:

- $\forall n \in N_R$ with $nt_R(n) \in \{\text{AnteOcc}, \text{AnteAbs}\}$:

  $(ns'_A(n), nl'_A(n)) := (ns_A(n), nl_A(n))$.
- $\forall n \in N_R$ with $nt_R(n) = \text{ConsOcc}$:

$$(ns'_C(n), nl'_C(n)) := \begin{cases} (\text{Started}, \{e\})\ if\ n \in Q \\ (ns_C(n), nl_C(n)),\ otherwise. \end{cases}$$

- $\forall n \in N_R$ with $nt_R(n) = \text{ConsAbs}$:

$$(ns'_C(n), nl'_C(n)) := \begin{cases} (\text{NotExecuted}, \emptyset)\ if\ n \in deadConsAbs_R(ns_C, Q) \\ (ns_C(n), nl_C(n)),\ otherwise. \end{cases}$$

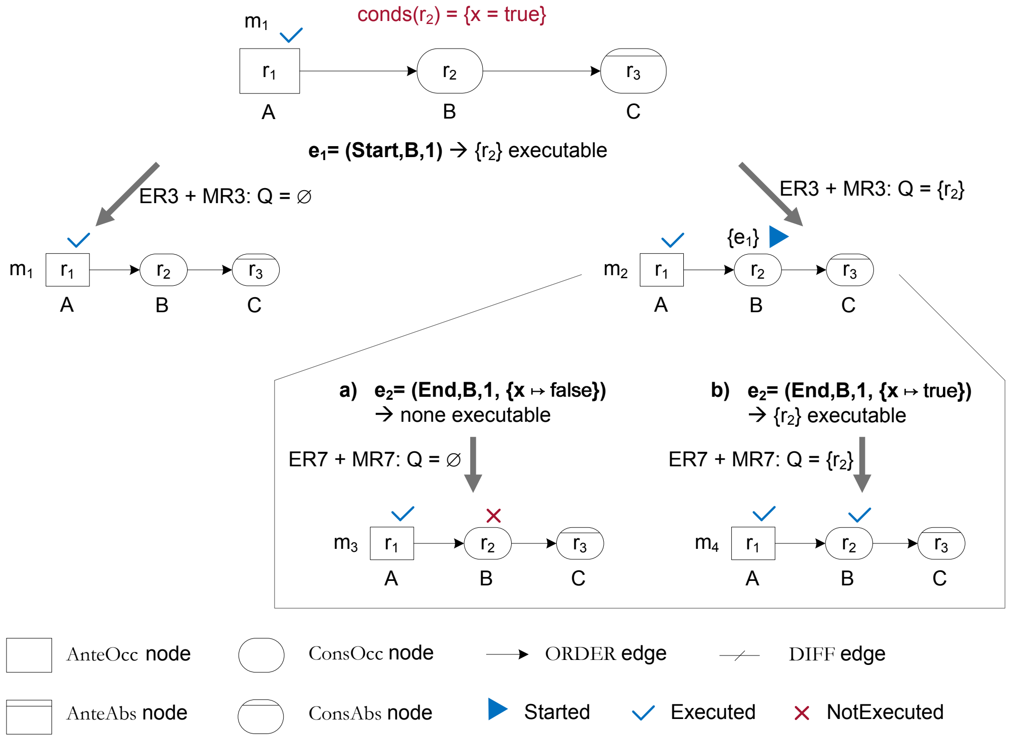


Figure 7.20.: Application of ER3 and MR3 and ER7 and MR7

Example 7.22 (Execution of ConsOcc nodes over a START event):
In Fig. 7.20, ConsOcc node r_2 of ExMark m_1 becomes executable when e_1 is observed. Due to non-deterministic execution of ConsOcc nodes over START events, application of ER3 and MR3 results in two child ExMarks m_1 and m_2. As these ExMarks both have the same AnteExMark, aggregation of them at the end of the iteration would yield a single MarkStructure with two ConsExMarks capturing the ConsExMarks of both ExMarks. According to Def. 7.5, this MarkStructure is Activated (i.e., represents an activation of the CRG) and both violable and pending (i.e., still awaiting events).

Fig. 7.21 illustrates how the semantics of DIFF edges is enforced through ER3. When e_1 is observed, both r_2 and r_3 become executable. As, however, they are linked through a DIFF edge, not both of them can be started in the same iteration. As a result, we obtain three child ExMarks from m_1. At the end of the iteration for e_1, these three ExMarks will be aggregated to a MarkStructure as they are associated with the same AnteExMark.

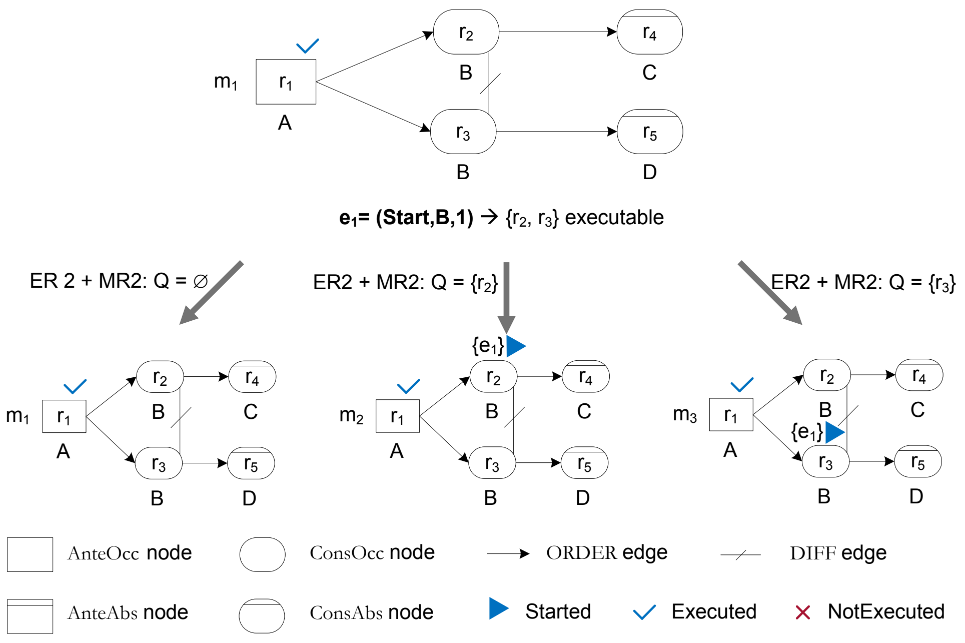


Figure 7.21.: Application of ER3 and MR3 when a DIFF edge is involved

7.3.2.5. Execution of ConsAbs nodes

The execution rule for ConsAbs nodes is very similar to the one for AnteAbs nodes. In order to implement the formal semantics of DIFF edges, it has to be ensured that ConsAbs nodes that are directly linked through DIFF edges to ConsOcc or AnteOcc nodes started in the same iteration are not executed. Execution rule ER4 enforces this as such nodes (represented by D) are excluded from execution.

Execution Rule ER4 (Execution of ConsAbs nodes over START events):
Let $R = (A, C)$ be a CRG, $m = ((ns_A, nl_A), (ns_C, nl_C)) \in M_R^*$ be an ExMark of R and $e \in E^{Start}$ be a START event with

- $QCA := exConsNodesStart_R((ns_A, ns_C), e, \texttt{ConsAbs})$ being the set of ConsAbs nodes of m satisfying the necessary execution conditions,
- $ExAnteOcc_m$ being the set of AnteOcc nodes that have been marked as Started in m over e in the same iteration,
- $ExConsOcc_m$ being the set of ConsOcc nodes that have been marked as Started in m over e in the same iteration,
- $D := \{n \in QCA \mid \exists l \in ExAnteOcc_m \cup ExConsOcc_m : (n, l) \in Diff_R \vee (l, n) \in Diff_R\}$ being the set of ConsAbs nodes in QCA directly linked through DIFF edges to ConsOcc and AnteOcc nodes that have been executed in this iteration, and
- $Q := QCA \backslash D$ being the subset of of QCA that is free of nodes that are directly linked to AnteOcc nodes or ConsOcc nodes executed in same iteration through DIFF edges.

Then,

- $executeConsAbsStart_R : M_R^* \times E^{Start} \rightarrow M_R^*$
  is a function assigning a child ExMark to an ExMark m and a START event e with:

  $executeConsAbsStart_R(m, e) := markConsAbsStart_R(m, Q, e)$.

Marking rule MR4 takes care of marking ConsAbs nodes to be executed. A ConsAbs node to be executed will be assigned execution state Started if it is in execution state Null. If the node is already Started, it will remain so. The *nl*-property of the ConsAbs node is further altered to associate the node with the processed event by adding the latter to the node's event set.

Marking Rule MR4 (Marking of ConsAbs nodes over START events):
Let $R = (A, C)$ be a CRG. Then,

$markConsAbsStart_R : M_R^* \times \mathcal{P}(N_R) \times E^{Start} \rightarrow M_R^*$

is a function assigning an ExMark m' to an original ExMark $m = ((ns_A, nl_A), (ns_C, nl_C))$, a set Q of ConsOcc nodes to be executed, and a START event e with

$markConsAbsStart_R(m, Q, e) := m' = ((ns'_A, nl'_A), (ns'_C, nl'_C))$ with:

- $\forall n \in N_R$ with $nt_R(n) \in \{\texttt{AnteOcc}, \texttt{AnteAbs}\}$:

  $(ns'_A(n), nl'_A(n)) := (ns_A(n), nl_A(n))$.
- $\forall n \in N_R$ with $nt_R(n) = \texttt{ConsOcc}$:

  $(ns'_C(n), nl'_C(n)) := (ns_C(n), nl_C(n))$.

- $\forall n \in N_R$ with $nt_R(n) = \textsc{ConsAbs}$:

$$(ns'_C(n), nl'_C(n)) := \begin{cases} (\textsc{Started}, \{e\})\ if\ n \in Q \wedge ns_C(n) = \textsc{Null} \\ (ns_C(n), nl_C \cup \{e\})\ if\ n \in Q \wedge ns_C(n) = \textsc{Started} \\ (ns_C(n), nl_C(n)),\ otherwise. \end{cases}$$

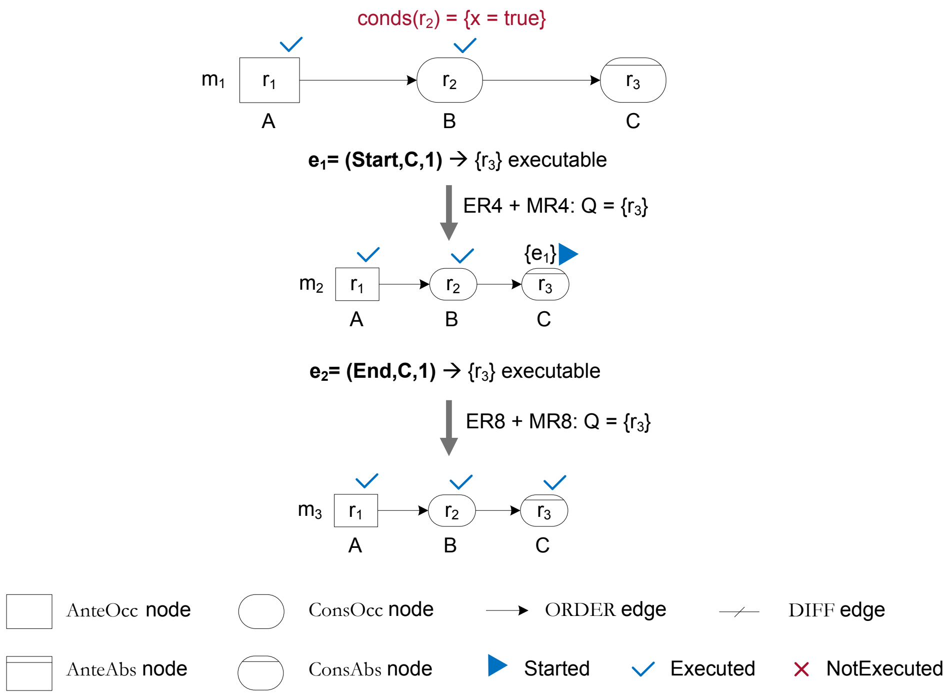


Figure 7.22.: Application of ER4 and MR4 and ER8 and MR8

Example 7.23 (Execution of ConsAbs nodes over a START event):
In Fig. 7.22, ConsAbs node r_3 becomes executable over START event e_1. Application of ER4 and MR4 results in ExMark m_2. Note that if another START event matching r_3 would be observed next, r_3 would still be executable.

7.3.3. Execution and marking rules for END events

Based on the considerations described in Section 7.3.1, Fig. 7.23 illustrates how a MarkStructure is altered when processing an END event. Due to deterministic execution of all CRG node types over END events, an execution iteration for an END event results in a single child MarkStructure[11].

[11]Note that the amount of ConsExMarks associated with a MarkStructure may also diminish as originally different ConsExMarks may result in the same ConsExMark when being altered.

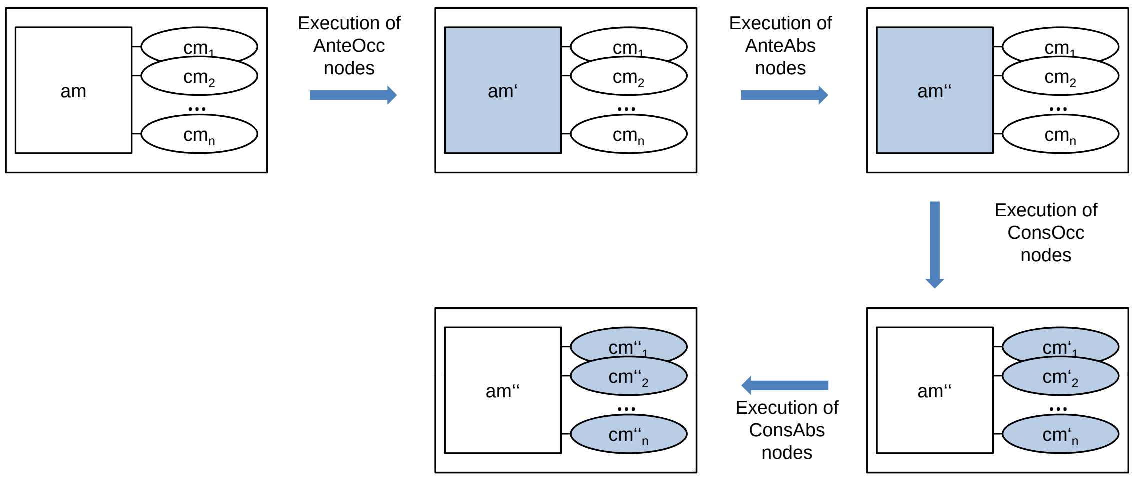

am
cm_1
cm_2
...
cm_n
Execution of AnteOcc nodes
am'
Execution of AnteAbs nodes
am"
Execution of ConsOcc nodes
cm'_1
cm'_2
cm'_n
Execution of ConsAbs nodes
cm''_1
cm''_2
cm''_n

Figure 7.23.: Alternation of a MARKSTRUCTURE when processing an END event

Algorithm 3 defines function *executeEnd* processing a MARKSTRUCTURE over an END event. For intelligibility, the execution and marking rules are defined over EXMARKs instead of MARKSTRUCTUREs. Therefore, obtained EXMARKs are aggregated to a MARKSTRUCTURE in line 13 in Algorithm 3. In lines 6 to 12, execution rules are applied. As firing conflicts between ordered nodes are already prevented by starting nodes in a particular order, a particular order is not necessary when processing END events. However, we still stick to the processing order for uniformity reasons. The functions to execute the particular node types are introduced in Sections 7.3.3.2 to 7.3.3.5, respectively.

Generally, the execution rules for processing END events are simpler than the ones for processing START events. The reason for this is that DIFF edges are already taken care of when processing START events. Therefore, this can be neglected when END events are processed. As the discard of not yet completed predecessor nodes (cf. Section 7.3.1.4) is taken care of when processing START events, marking rules for END events only affect the nodes to be executed. In the following, Section 7.3.3.1 discusses the necessary conditions for executing a CRG node over an END event before execution and marking rules for the particular node types are introduced.

7.3.3.1. Necessary conditions

As emphasized in Section 7.3.1.3, a CRG node must only be completed over an END event if it was also started over the START event from the same activity execution. To ensure this, Def. 7.9 provides a function to determine whether an END event is the expected by a started CRG node. Note that Def. 7.9 does not account for multiple occurrences of the same events, which can occur when an activity is carried out multiple times. This, therefore, has to be taken care of by the execution and marking rules.

Algorithm 3 Executing MARKSTRUCTURE over an END event ($executeEnd_R(ms, e)$)

```
1:  $R = (A, C)$ is a CRG;
2:  $e \in E^{End}$ is a END event;
3:  $ms = ((ns_A, nl_A), \{(ns_C^1, nl_C^1), \ldots, (ns_C^k, nl_C^k)\}) \in MS_R^*$ is a MARKSTRUCTURE of $R$;
4:  $M_{ms} = \{m_1 = ((ns_A, nl_A), (ns_C^1, nl_C^1)), \ldots, m_k = ((ns_A, nl_A), (ns_C^k, nl_C^k))\}$ is the set of
    EXMARKs of $ms$

    {INITIALIZATION}
5:  $M_{Res} = \emptyset$;

    {ITERATION}
6:  for all $m \in M_{ms}$ do
7:      $m_{AnteOcc} = executeAnteOccEnd_R(m, e)$;
8:      $m_{AnteAbs} = executeAnteAbsEnd_R(m_{AnteOcc,e})$;
9:      $m_{ConsOcc} = executeConsOccEnd_R(m_{AnteAbs,e})$;
10:     $m_{ConsAbs} = executeConsAbsEnd_R(m_{ConsOcc,e})$;
11:     $M_{Res} = M_{Res} \cup \{m_{ConsAbs}\}$;
12: end for

    {Aggregation of obtained EXMARKs to a MARKSTRUCTURE }
13: $ms_{res} = aggregate_R(M_{Res})$;

    {The obtained MARKSTRUCTURE is returned}
14: return $ms_{res}$;
```

Definition 7.9 (*expectedEnd*)
Let $R = (A, C)$ be a CRG and $e = (\text{END}, n_e, at_e, data_e) \in E^{End}$ be an END event. Then,
$expectedEnd_R : NL_A^* \cup NL_C^* \times N_R \times E^{End} \rightarrow \mathbb{B}$
is a function returning `true` if n is associated with a START event that belongs to the END event e, with:

$$expectedEnd_R(nl, n, e) := \begin{cases} \texttt{true} \; if \; \exists s \in nl(n) \; with \; s = (\text{START}, n_e, at_e, data_s) \\ \texttt{false}, \; otherwise. \end{cases}$$

Based on the considerations described in Section 7.3.1, the necessary conditions for executing a CRG node over an END event are constituted by conditions on:

1. the node's current execution state,
2. the matching between the node's specification and the event (end conditions), and
3. the correspondence of the END event with a START event the node is associated with.

These necessary conditions are formalized in Definition 7.10. Condition (i) ensures that the node is STARTED. Condition (ii) ensures that the observed END event matches the node's specifications and condition (iii) ensures that a node will only become executable over an END event if the latter is expected.

Definition 7.10 (Necessary conditions for executing CRG nodes over END events)
Let $R = (A, C)$ be a CRG and let $e \in E^{End}$ be an END event. Then,

- $executableAnteEnd_R : NSL_A^* \times N_A \times E^{End} \rightarrow \mathbb{B}$ is a function determining whether an antecedent node n is executable under an `AnteExMark` (ns_A, nl_A) over an END event e with:

$$executableAnteEnd_R((ns_A, nl_A), n, e) := \begin{cases} true, \ if \\ \ (i)\ ns_A(n) = \texttt{Started} \wedge \\ \ (ii)\ matchEnd_R(n, e) = true \wedge \\ \ (iii)\ expectedEnd_R(nl_A, n, e) = true \\ false, \ otherwise. \end{cases}$$

- $executableConsEnd_R : NSL_C^* \times N_C \times E^{End} \rightarrow \mathbb{B}$

  is a function determining whether a consequence node n is executable under a `ConsExMark` (ns_C, nl_C) over an END event e with:

$$executableConsEnd_R((ns_C, nl_C), n, e) := \begin{cases} true, \ if \\ \ (i)\ ns_C(n) = \texttt{Started} \wedge \\ \ (ii)\ matchEnd_R(n, e) = true \wedge \\ \ (iii)\ expectedEnd_R(nl_C, n, e) = true \\ false, \ otherwise. \end{cases}$$

- $exAnteNodesEnd_R : NSL_A^* \times E^{End} \times \{\texttt{AnteOcc}, \texttt{AnteAbs}\} \rightarrow \mathcal{P}(N_R)$ is a function determining the set of antecedent nodes of type t that are executable under `AnteExMark` (ns_A, nl_A) over END event e with:
  $exAnteNodesEnd_R((ns_A, nl_A), e, t) :=$
  $\{n \in N_R \mid nt_R(n) = t \ \wedge \ executableAnteEnd_R((ns_A, nl_A), n, e) = true\}$.
- $exConsNodesEnd_R : NSL_C^* \times E^{End} \times \{\texttt{ConsOcc}, \texttt{ConsAbs}\} \rightarrow \mathcal{P}(N_R)$

  is a function determining the set of consequence nodes of type t that are executable under `ConsExMark` (ns_C, nl_C) over END event e with:
  $exConsNodesEnd_R((ns_C, nl_C), e, t) :=$
  $\{n \in N_R \mid nt_R(n) = t \ \wedge \ executableConsEnd_R((ns_C, nl_C), n, e) = true\}$.

Example 7.24 (Necessary conditions):
Consider for example Fig. 7.24. In case a), END event e_3 is not the expected END event of r_1 as it refers to a different node than the START events associated with r_1. Therefore, the necessary execution conditions do not apply. In case b), e_3 is the expected END event of e_2. However, the end condition is not satisfied by e_3. Therefore, the necessary execution conditions do not

apply in this case either. In case c), e_3 is the expected END event of e_2 and also satisfies the end condition. The necessary execution conditions therefore apply and r_1 is executable over e_3.

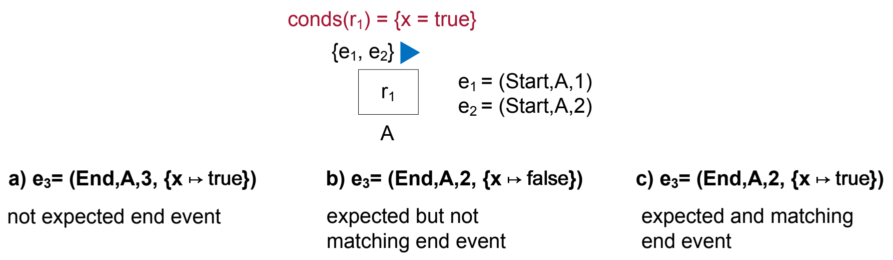

conds(r₁) = {x = true}
{e₁, e₂}
r₁
A
e₁ = (Start,A,1)
e₂ = (Start,A,2)
a) e₃= (End,A,3, {x ↦ true})
not expected end event
b) e₃= (End,A,2, {x ↦ false})
expected but not matching end event
c) e₃= (End,A,2, {x ↦ true})
expected and matching end event

Figure 7.24.: Necessary conditions for executing a node over an END event

7.3.3.2. Execution of AnteOcc nodes

The execution of occurrence nodes over END event is fairly straight-forward. Execution rule ER5 invokes marking rule MR5 for the set of AnteOcc nodes satisfying the necessary execution conditions.

Execution Rule ER5 (Execution of AnteOcc nodes over END events):
Let $R = (A, C)$ be a CRG, $m = ((ns_A, nl_A), (ns_C, nl_C)) \in M_R^*$ be an ExMark of R, and $e \in E^{End}$ be an END event with
$Q := exAnteNodesEnd_R((ns_A, nl_A), e,$ AnteOcc$)$ being the set of AnteOcc nodes of m satisfying the necessary execution conditions.

Then,

- $executeAnteOccEnd_R : M_R^* \times E^{End} \rightarrow M_R^*$
  is a function assigning a child ExMark to ExMark m and END event e with:

  $executeAnteOccEnd_R(m, e) := markAnteOccEnd_R(m, Q, e)$.

Marking rule MR5 follows the considerations described in Section 7.3.1.5 (cf. Fig. 7.10). If a matching activity execution is found, the corresponding node will be assigned execution state Completed. In this case, the nl-property will be set to $\emptyset$[12]. If however, a Started AnteOcc node does not satisfy the necessary conditions (i.e., is not contained in Q) despite the observed END event being expected by the node, the node will be assigned execution state NotExecuted, as the corresponding activity execution does not match with the CRG node.

[12]Note that this yields event-independent ExMarks. In Section 7.4.3, we will show how the marking rules can be adapted to yield event-specific ExMarks that reveal the particular activations of a CRG.

Marking Rule MR5 (Marking of $\textsc{AnteOcc}$ nodes over $\textsc{End}$ events):
Let $R = (A, C)$ be a CRG. Then,

$markAnteOccEnd_R : M_R^* \times \mathcal{P}(N_R) \times E^{End} \rightarrow M_R^*$

is a function assigning an $\textsc{ExMark}$ m' to an original $\textsc{ExMark}$ $m = ((ns_A, nl_A), (ns_C, nl_C))$, a set Q of $\textsc{AnteOcc}$ nodes to be executed, and an $\textsc{End}$ event e with

$markAnteOccEnd_R(m, Q, e) := m' = ((ns'_A, nl'_A), (ns'_C, nl'_C))$ with:

- $\forall n \in N_R$ with $nt_R(n) = \textsc{AnteOcc}$:

$$(ns'_A(n), nl'_A(n)) := \begin{cases} (\textsc{Completed}, \emptyset) \; if \; n \in Q \\ (\textsc{NotExecuted}, \emptyset) \; if \; n \notin Q \wedge expectedEnd_R(nl_A, n, e) = \texttt{true} \\ (ns_A(n), nl_A(n)), \; otherwise. \end{cases}$$

- $\forall n \in N_R$ with $nt_R(n) = \textsc{AnteAbs}$:

  $(ns'_A(n), nl'_A(n)) := (ns_A(n), nl_A(n))$.

- $\forall n \in N_R$ with $nt_R(n) \in \{\textsc{ConsOcc}, \textsc{ConsAbs}\}$:

  $(ns'_C(n), nl'_C(n)) := (ns_C(n), nl_C(n))$.

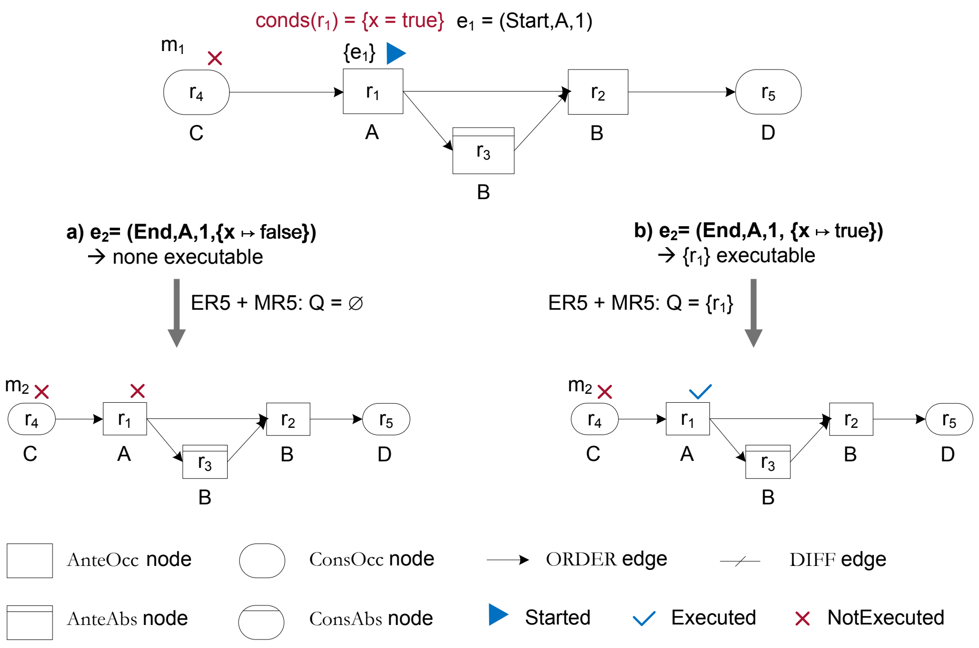


Figure 7.25.: Application of ER5 and MR5

Example 7.25 (Execution of AnteOcc nodes over an END event):
Consider, for example, ExMark m_1 depicted in Fig. 7.25. In case a), the processed END event is the expected END event of r_1 but does not match r_1 due to r_1's end condition. Application of MR5 therefore results in r_1 being marked as NotExecuted. In case b), the END event matches r_1. Application of ER5 and MR5, hence, results in r_1 being marked as Completed.

7.3.3.3. Execution of AnteAbs nodes

Execution rule ER6 determines the set of AnteAbs nodes satisfying the necessary execution conditions and invokes marking rule MR6 for these nodes to obtain a child ExMark.

Execution Rule ER6 (Execution of AnteAbs nodes over END events):
Let $R = (A, C)$ be a CRG, $m = ((ns_A, nl_A), (ns_C, nl_C)) \in M_R^*$ be an ExMark of R, and $e \in E^{End}$ be an END event with

- $Q := exAnteNodesEnd_R((ns_A, nl_A), e, \texttt{AnteAbs})$ being the set of AnteAbs nodes of m satisfying the necessary execution conditions.

Then,

- $executeAnteAbsEnd_R : M_R^* \times E^{End} \to M_R^*$
  is a function assigning a child ExMark to ExMark m and END event e with:
  $executeAnteAbsEnd_R(m, e) := markAnteOccEnd_R(m, Q, e)$.

In contrast to occurrence nodes, Started absence nodes may be associated with multiple START events. Based on the considerations discussed in Section 7.3.1.5, MR6 accounts for three cases. If an AnteAbs node satisfies the necessary conditions (i.e., it is contained in Q), it will be assigned execution state Completed as a matching activity execution is found. If the END event is expected by a Started AnteAbs node that, however, is not contained in Q (i.e., the expected END event does not lead to a matching activity execution), two cases will be left: If the node is still expecting other END events that could lead to a matching activity execution, the node will remain Started and the nl-property will be updated[13]. If, however, the node is not expecting other END events than the one being processed, it will be assigned execution state Null as it is no longer associated with a started activity execution.

Marking Rule MR6 (Marking of AnteAbs nodes over END events):
Let $R = (A, C)$ be a CRG and $m = ((ns_A, nl_A), (ns_C, nl_C)) \in M_R^*$ be an ExMark of R. Let further Q be a set of AnteAbs nodes in R and let $e \in E^{End}$ be an END event. Then,

$markAnteAbsEnd_R : M_R^* \times \mathcal{P}(N_R) \times E^{End} \to M_R^*$

[13]This is done to reflect the fact that the END event just processed is no longer expected.

is a function assigning an ExMark m' to an original ExMark m, a set Q of AnteAbs nodes to be executed, and an END event e with

$markAnteAbsEnd_R(m, Q, e) := m' = ((ns'_A, nl'_A), (ns'_C, nl'_C))$ with:

- $\forall n \in N_R$ with $nt_R(n) = \texttt{AnteAbs}$:

$$(ns'_A(n), nl'_A(n)) := \begin{cases} (\texttt{Completed}, \emptyset)\ if\ n \in Q \\ (\texttt{Null}, \emptyset)\ if\ n \notin Q \wedge expectedEnd_R(nl_A, n, e) = \texttt{true} \wedge \\ \quad nl_A(n) \backslash \{e\} = \emptyset \\ (ns_A(n), nl_A(n) \backslash \{e\})\ if\ n \notin Q \wedge \\ \quad expectedEnd_R(nl_A, n, e) = \texttt{true} \wedge nl_A(n) \backslash \{e\} \neq \emptyset \\ (ns_A(n), nl_A(n)),\ otherwise. \end{cases}$$

- $\forall n \in N_R$ with $nt_R(n) = \texttt{AnteOcc}$:

  $(ns'_A(n), nl'_A(n)) := (ns_A(n), nl_A(n))$.

- $\forall n \in N_R$ with $nt_R(n) \in \{\texttt{ConsOcc}, \texttt{ConsAbs}\}$:

  $(ns'_C(n), nl'_C(n)) := (ns_C(n), nl_C(n))$.

Example 7.26 (Execution of AnteAbs nodes over an END event):
Consider, for example, Fig. 7.19. In ExMark m_2, no AnteAbs nodes become executable when END event e_3 is processed. As e_3 does not match r_2 but is the expected END event of e_1, application of MR6 yields m_3.

In the next iteration, e_4 is processed. In case a), e_4 does not match r_2. Hence, MR6 puts r_2 back into execution state Null. In case b), e_4 matches r_2 (i.e., a matching activity execution is found). In consequence, r_2 becomes Completed.

7.3.3.4. Execution of ConsOcc nodes

Execution rule ER7 identifies ConsOcc nodes of an ExMark that satisfy the necessary execution conditions and invokes marking rule MR7 for these nodes to obtain a child ExMark.

Execution Rule ER7 (Execution of ConsOcc nodes over END events):
Let $R = (A, C)$ be a CRG, $m = ((ns_A, nl_A), (ns_C, nl_C)) \in M^*_R$ be an ExMark of R, and $e \in E^{End}$ be an END event with

- $Q := exConsNodesEnd_R((ns_C, nl_C), e, \texttt{ConsOcc})$ being the set of ConsOcc nodes of m satisfying the necessary execution conditions.

Then,

- $executeConsOccEnd_R : M^*_R \times E^{End} \rightarrow M^*_R$
  is a function assigning a child ExMark to an ExMark m and an END event e with:

  $executeConsOccEnd_R(m, e) := markConsOccEnd_R(m, Q, e)$.

Marking rule MR7 takes care of marking ConsOcc nodes to be executed. Similar to MR5, a ConsOcc node to be executed will become Completed. If however, a Started ConsOcc node does not satisfy the necessary conditions (i.e., is not contained in Q) despite the observed END event being expected by the node, the node will be assigned execution state NotExecuted.

Marking Rule MR7 (Marking of ConsOcc nodes over END events):
Let $R = (A, C)$ be a CRG and $m = ((ns_A, nl_A), (ns_C, nl_C)) \in M^*_R$ be an ExMark of R. Let further Q be a set of ConsOcc nodes in R and let $e \in E^{End}$ be an END event. Then,

$markConsOccEnd_R : M^*_R \times \mathcal{P}(N_R) \times E^{End} \rightarrow M^*_R$

is a function assigning an ExMark m' to an original ExMark m, a set Q of ConsOcc nodes to be executed, and an END event e with

$markConsOccEnd_R(m, Q, e) := m' = ((ns'_A, nl'_A), (ns'_C, nl'_C))$ with:

- $\forall n \in N_R$ with $nt_R(n) \in \{\texttt{AnteOcc}, \texttt{AnteAbs}\}$:

  $(ns'_A(n), nl'_A(n)) := (ns_A(n), nl_A(n))$.

- $\forall n \in N_R$ with $nt_R(n) = \texttt{ConsOcc}$:

$$(ns'_C(n), nl'_C(n)) := \begin{cases} (\texttt{Completed}, \emptyset) \ if \ n \in Q \\ (\texttt{NotExecuted}, \emptyset) \ if \ n \notin Q \wedge expectedEnd_R(nl_C, n, e) = \texttt{true} \\ (ns_C(n), nl_C(n)), \ otherwise. \end{cases}$$

- $\forall n \in N_R$ with $nt_R(n) = \texttt{ConsAbs}$:

  $(ns'_C(n), nl'_C(n)) := (ns_C(n), nl_C(n))$.

Example 7.27 (Execution of ConsOcc nodes over an END event):
In Fig. 7.20, ConsOcc node r_2 of ExMark m_2 becomes executable when e_2 is observed (case b)). Application of ER7 and MR7, therefore, yields an ExMark, in which r_2 is marked as Completed.

Case a) illustrates the situation that e_2, the expected event of e_1, does not match r_2. In this case, application of MR7 results in r_2 being marked as NotExecuted.

7.3.3.5. Execution of ConsAbs nodes

Execution rule ER8 identifies ConsAbs nodes of an ExMark that satisfy the necessary execution conditions and invokes marking rule MR8 to obtain a child ExMark.

Execution Rule ER8 (Execution of ConsAbs nodes over END events):
Let $R = (A, C)$ be a CRG, $m = ((ns_A, nl_A), (ns_C, nl_C)) \in M_R^*$ be an ExMark of R, and $e \in E^{End}$ be an END event with

- $Q := exConsNodesEnd_R((ns_C, nl_C), e, \texttt{ConsAbs})$ being the set of ConsAbs nodes of m satisfying the necessary execution conditions.

Then,

- $executeConsAbsEnd_R : M_R^* \times E^{End} \to M_R^*$
  is a function assigning a child ExMark to an ExMark m and an END event e with:

  $executeConsAbsEnd_R(m, e) := markConsAbsEnd_R(m, Q, e)$.

Similar to MR6, MR8 accounts for three cases: If a Started ConsAbs node was identified as executable, it will be assigned execution state Completed as a matching activity execution is found. If the processed END event is expected by a Started ConsAbs node but does not belong to a matching activity execution, the node will remain in execution state Started if it is still expecting other END events. If the node is not expecting any other END events, it will be put back into execution state Null as it is no longer associated with a started activity execution.

Marking Rule MR8 (Marking of ConsAbs nodes over END events):
Let $R = (A, C)$ be a CRG and $m = ((ns_A, nl_A), (ns_C, nl_C)) \in M_R^*$ be an ExMark of R. Let further Q be a set of ConsAbs nodes in R and let $e \in E^{End}$ be an END event. Then,

$markConsAbsEnd_R : M_R^* \times \mathcal{P}(N_R) \times E^{End} \to M_R^*$

is a function assigning an ExMark m' to an original ExMark m, a set Q of ConsAbs nodes to be executed, and an END event e with

$markConsAbsEnd_R(m, Q, e) := m' = ((ns'_A, nl'_A), (ns'_C, nl'_C))$ with:

- $\forall n \in N_R$ with $nt_R(n) \in \{\texttt{AnteOcc}, \texttt{AnteAbs}\}$:

  $(ns'_A(n), nl'_A(n)) := (ns_A(n), nl_A(n))$.

- $\forall n \in N_R$ with $nt_R(n) = \texttt{ConsOcc}$:

  $(ns'_C(n), nl'_C(n)) := (ns_C(n), nl_C(n))$.

- $\forall n \in N_R$ with $nt_R(n) = \texttt{ConsAbs}$:

$$(ns'_C(n), nl'_C(n)) := \begin{cases} (\texttt{Completed}, \emptyset)\ if\ n \in Q \\ (\texttt{Null}, \emptyset)\ if\ n \notin Q \wedge expectedEnd_R(nl_C, n, e) = \texttt{true} \wedge \\ \quad nl_C(n) \backslash \{e\} = \emptyset \\ (ns_C(n), nl_C(n) \backslash \{e\})\ if\ n \notin Q \wedge \\ \quad expectedEnd_R(nl_C, n, e) = \texttt{true} \wedge nl_C(n) \backslash \{e\} \neq \emptyset \\ (ns_C(n), nl_C(n)),\ otherwise. \end{cases}$$

Example 7.28 (Execution of ConsAbs nodes over an END event):
In Fig. 7.22, END event e_2, the expected END event of e_1, is processed in the second iteration. In m_2, Started ConsAbs node r_3 is executable over e_2. Therefore, application of ER8 and MR8 results in m_3 where r_3 is marked as Completed. However, if e_2 would not match r_3 (due to imposed end conditions), r_3 would be marked as Null when applying MR8.

7.3.4. End marking

In order to enable the evaluation of MarkStructures using Def. 7.5 (cf. Section 7.2.2), it is necessary to indicate that no further events will be observed when reaching the end of an execution trace (or the end node of a PEG). For this purpose, we introduce marking rule MR9 that finalizes MarkStructures upon reaching end of execution (cf. Algorithm 1). MR9 assigns (NotExecuted,$\emptyset$) to each not yet completed node.

Marking Rule MR9 (Marking MarkStructures when terminating execution):
Let $R = (A, C)$ be a CRG and let $ms = ((ns_A, nl_A), \{(ns_C^1, nl_C^1), \ldots, (ns_C^k, nl_C^k)\}) \in MS_R^*$ be a MarkStructure of R. Then,

$markEnd_R : MS_R^* \rightarrow MS_R^*$ is a function assigning a MarkStructure ms' of R to an original MarkStructure ms when end of trace is reached, with

$markEnd_R(ms) := ms' = ((ns'_A, nl'_A), \{(ns'^1_C, nl'^1_C), \ldots, (ns'^k_C, nl'^k_C)\})$ with

- $\forall n \in N_A$:

$$(ns'_A(n), nl'_A(n)) := \begin{cases} (\texttt{NotExecuted}, \emptyset)\ if\ ns_A(n) \in \{\texttt{Null}, \texttt{Started}\} \\ (ns_A(n), nl_A(n)),\ otherwise. \end{cases}$$

- $\forall n \in N_C$:

$$(ns'^i_C(n), nl'^i_C(n)) := \begin{cases} (\texttt{NotExecuted}, \emptyset)\ if\ ns^i_C(n) \in \{\texttt{Null}, \texttt{Started}\} \\ (ns^i_C(n), nl^i_C(n)),\ otherwise. \end{cases}$$

The application of MR9 renders all MarkStructures final when reaching the end of an execution trace. Thus, a MarkStructure obtained by applying MR9 will be either activated or deactivated according to Def. 7.5. If activated, it will be either satisfied or violated.

7.4. Application of compliance rule graph operationalization

In the following, the application of the execution and marking rules will be illustrated using examples. Recall that MarkStructures are utilized to capture a compliance state. In Algorithm 1 (cf. Section 7.3.1.8), execution rules (*executeStart* from Algorithm 2 and *executeEnd* from Algorithm 3) are applied depending on the observed event type in order to update the current MarkStructures. In Section 7.3.2 and 7.3.3, we formalized the execution and marking rules based on ExMarks instead of MarkStructures (for clarity reasons). Therefore, a MarkStructure is updated by applying the execution and marking rules to its ExMarks and aggregating the resulting ExMarks by the end of each iteration. For practical implementation, the rules can be applied directly to a MarkStructure.

As MarkStructures obtained from executing a CRG over an execution trace represent the behavior in the trace with regard to the CRG, analyzing these MarkStructures reveals whether the CRG is enforced in the trace. Clearly, if violated MarkStructures become manifest, the CRG is not enforced in the execution trace. If, on the contrary, no violated MarkStructures become manifest, the CRG is enforced in the execution trace.

7.4.1. Examples

Example 7.29:
The CRG shown in Fig. 7.26 necessitates the execution of C between each execution of A and the next execution of B. After processing e_2, the compliance state is reflected by ms_1 and ms_3. When e_3 is observed, both r_2 and r_3 become executable in ms_3 (i.e., a firing conflict between ordered nodes as described in Section 7.3.1.6). AnteOcc nodes are processed first in an iteration. This results in two MarkStructures, one in which r_2 is started (ms_4) and one in which r_2 is not started. In case r_2 is not started over e_3, r_3 remains executable in the same iteration. Thus, ER2 and MR2 can be applied which results in MarkStructure ms_5. After processing e_4, we can see that the CRG is activated in the trace as ms_6 is activated and violated. Note also that ms_7 is deactivated as an AnteAbs node is marked with Completed (cf. Def. 7.5). Hence, ms_7 cannot become activated through further events. This reflects the situation in the trace as no further B will lead to a new activation of the CRG unless a further A will be observed. Hence, deactivated MarkStructures do not have to be considered when processing new events. In Section 7.5, we discuss pruning strategies to increase the efficiency of CRG execution.

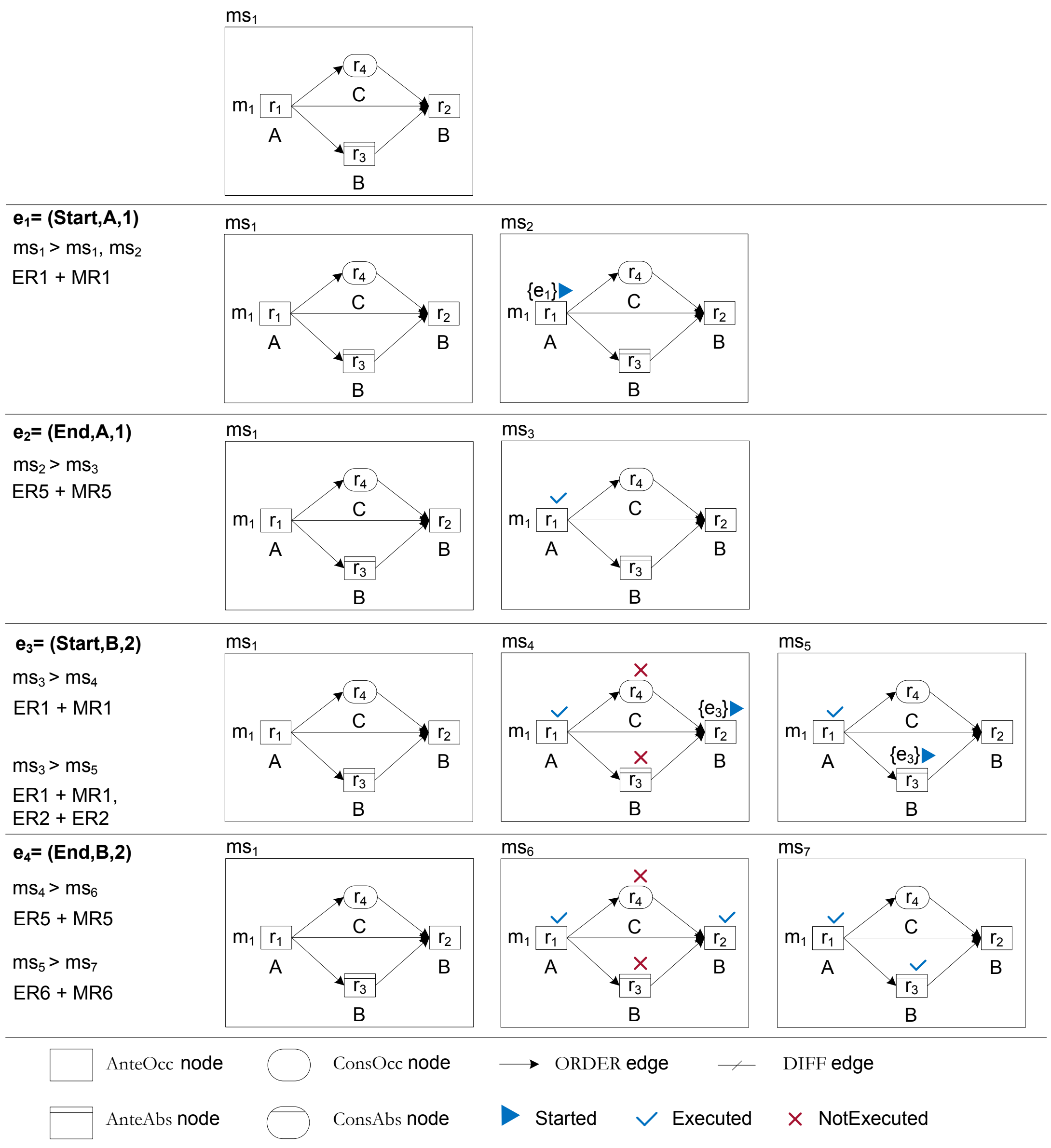


Figure 7.26.: Application of CRG operational semantics

Example 7.30:
The CRG shown in Fig. 7.27 requests that an execution of B is required after each execution of A. The trace in the figure contains two satisfied activations of the CRG. After processing e_8, these two activations are both represented by SATISFIED ms_4. According to Def. 7.4, ms_4 has a VIOLABLE (and PENDING) CONSEXMARKs and a SATISFIED CONSEXMARK. As a SATISFIED CONSEXMARK already suffices to satisfy a MARKSTRUCTURE, the VIOLABLE CONSEXMARK is *dominated* by the SATISFIED CONSEXMARK. In Section 7.5, we will discuss domination among CONSEXMARKS as a means to increase the efficiency of CRG execution.

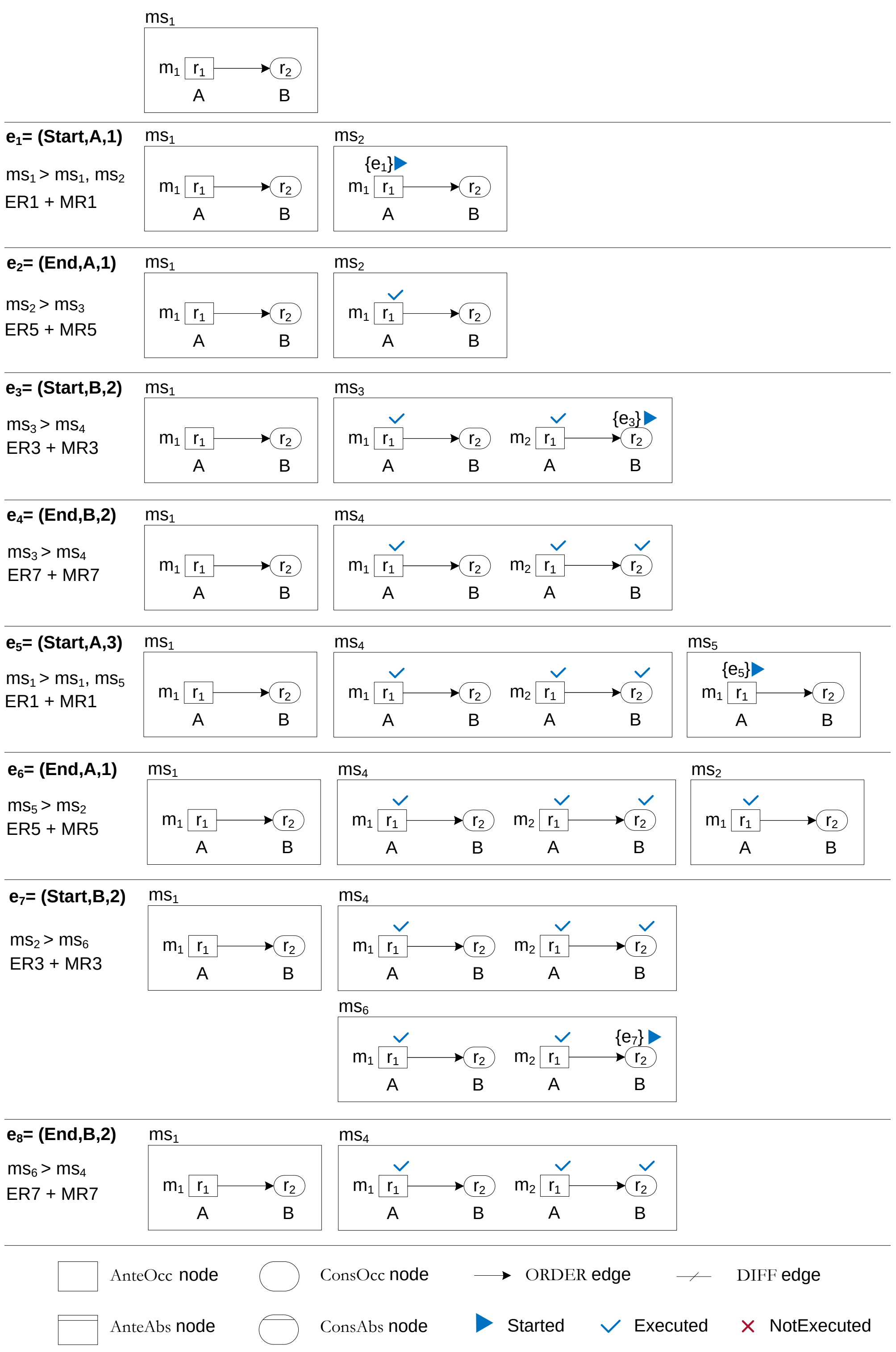


Figure 7.27.: Application of CRG operational semantics

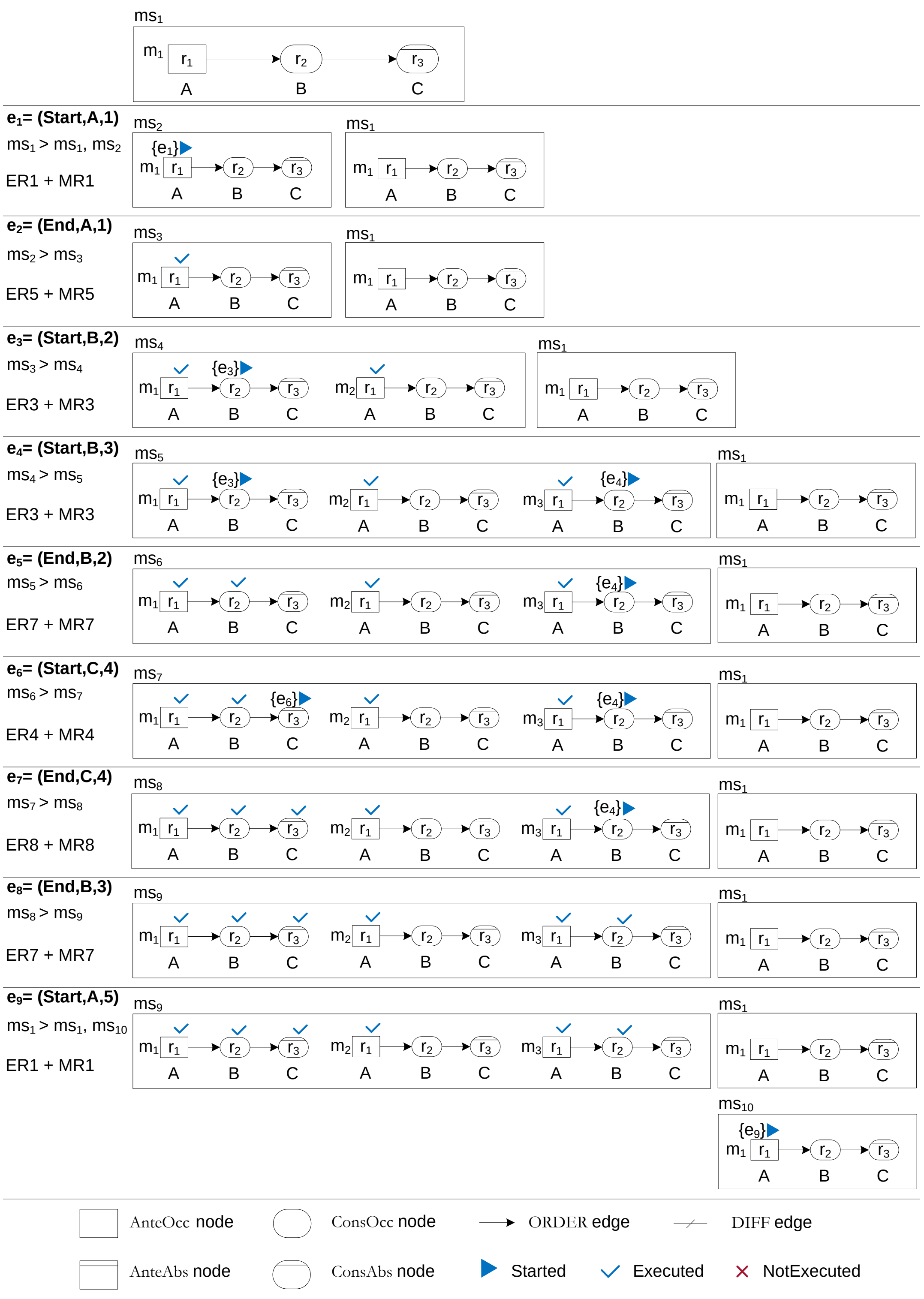


Figure 7.28.: Application of CRG operational semantics

Example 7.31:
Fig. 7.28 illustrates the execution of the CRG already taken as example in Fig. 7.11 (cf. Section 7.3.1.5). The CRG requests that each execution of A is followed by an execution of B without subsequent execution of C. The execution and marking rules applied in each iteration are given in Fig. 7.28. For example, for MarkStructure ms_1, ER1 and MR1 are applied to ExMark m_1 when e_1 is observed. After aggregation of the resulting ExMarks, we receive two MarkStructures ms_1 and ms_2 each consisting of a single ExMark.

After processing e_2, the compliance state is represented by ms_1 and ms_3. While ms_1 is still activatable and, thus, is not associated with an activation of the CRG, ms_3 is Activated but not yet satisfied as it does not have a satisfied ConsExMark (cf. Def. 7.5). If the process execution would stop after e_2, the end marking rule MR9 would be applied to ms_3, which would result in a violated MarkStructure.

In ms_8, the ConsExMark of m_1 is violated. This reflects that a B with subsequent C was observed. However, ms_8 is still violable (i.e., can still become satisfied) as it is associated with at least one violable ConsExMark (e.g., m_3).

After processing e_9, the compliance state is reflected by activatable ms_1, Activated and violable ms_9, and activatable ms_{10}. Thus, the CRG is activated over the processed execution trace. Immediate termination of the trace would result in ms_9 being rendered satisfied (as the ConsExMark of m_3 would become satisfied).

As these examples show, CRG execution can be applied to check compliance of an execution trace with imposed CRGs. The compliance state is reflected in the MarkStructures obtained from processing observed events at each stage. Using the compliance notions introduced in Section 7.2.2, it becomes possible to evaluate the compliance state of the individual MarkStructures and to aggregate these to an *overall compliance state*. This constitutes the fundament for compliance reports. By analyzing the violated and violable MarkStructures, the process supervisor can gain further insights into the compliance situation without having to analyze the execution trace, for example, during process execution. In Chapter 8, we will show how CRG execution can be applied to realize a compliance checking framework that addresses the complete process lifecycle.

7.4.2. Interpretation of MarkStructures

Based on the semantics of CRG nodes and their execution states, a MarkStructure can be interpreted to derive an intelligible and meaningful compliance diagnoses.

Violated MarkStructures As described by Def. 7.5, a violated MarkStructure contains solely violated ConsExMarks. This attests that all attempts to find a match for the consequence pattern of the respective CRG have failed. In that case, the violated ConsExMarks can provide valuable information for compliance diagnoses enabling to find the cause of the violation. For each violated ConsExMark, ConsOcc nodes marked with NotExecuted and ConsAbs nodes marked with COMPLETED constitute a cause for not satisfying the CRG's consequence as shown in Example 7.32.

Example 7.32 (Interpretation of VIOLATED MARKSTRUCTURES):
Consider the VIOLATED MARKSTRUCTURE ms_1 consisting of two EXMARKS depicted in Fig. 7.29. The CONSEXMARK of m_2, indicates that no execution of A without a subsequent execution of B could be detected before execution of C[14]. The CONSEXMARK of m_1, in turn, reveals that at least one execution of A followed by an execution of B before the execution of C was detected. From this, we can conclude that this violation can be resolved by either adding an A without subsequent B before C (to satisfy the CONSEXMARK of m_2) or removing B to ensure that no B is executed between A and C (to satisfy the CONSEXMARK of m_1).

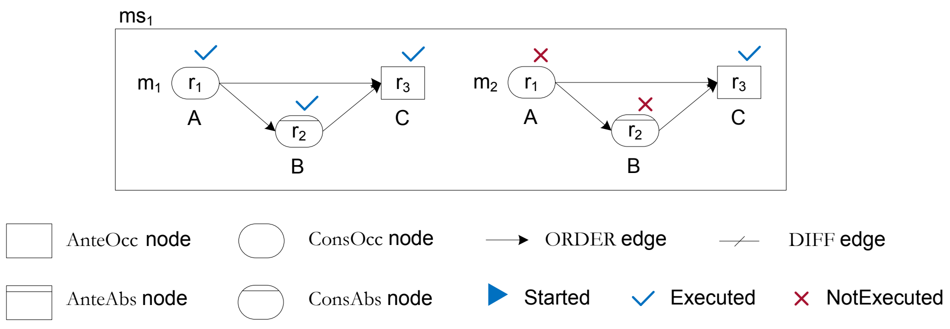


Figure 7.29.: Interpretation of VIOLATED MARKSTRUCTURES

Violable MARKSTRUCTURES As shown in our paper [LRMKD11], MARKSTRUCTURES can be used for compliance diagnosis even before a violation becomes manifest. In particular, VIOLABLE MARKSTRUCTURES can be exploited to derive measures required to render them SATISFIED. This is particularly interesting for compliance monitoring. Each VIOLABLE CONSEXMARK can become SATISFIED depending on the future events. Thus, for a VIOLABLE CONSEXMARK, each CONSOCC node not yet marked with COMPLETED represents still pending events. In turn, each CONSABS node that is not marked with NOTEXECUTED and does not have not yet COMPLETED CONSOCC predecessors represents an active absence constraint of the corresponding CONSEXMARK. The absence of the corresponding events is necessary in order for this CONSEXMARK to constitute an occurrence of the CRG's consequence pattern. Based on these considerations, we are able to derive event sequences for each VIOLABLE CONSEXMARK of a VIOLABLE MARKSTRUCTURE that lead to satisfaction of the CONSEXMARK and, thus, of the complete MARKSTRUCTURE[15]. This is illustrated in Example 7.33.

[14]Note that a MARKSTRUCTURE can be associated with multiple rule activations that belong to the same equivalence class. We will show in Section 7.4.3, how to enable per activation diagnoses.

[15]This can be done by parsing the CONSEXMARK. Then, the required event sequence can be derived based on the ordering of CRG nodes.

Example 7.33 (Interpretation of VIOLABLE MARKSTRUCTURES):
Consider the VIOLABLE MARKSTRUCTURE ms_1 from Fig. 7.30. As ms_1 is not PENDING according to Def. 7.5, no further events are necessary in order for it to become SATISFIED. This is reflected in EXMARK m_1 of ms_1. If no D is observed until the end of the trace, the CONSEXMARK of m_3 will become SATISFIED making ms_1 SATISFIED as a whole. However, we can also derive event sequences to render ms_1 SATISFIED from m_2 and m_3.

MARKSTRUCTURE ms_2 from Fig. 7.30 is also VIOLABLE. However, as it is PENDING, certain events are still required for its satisfaction. Which events in particular can be derived from its VIOLABLE CONSEXMARKS. For example, in order to render the CONSEXMARK of m_2 satisfied, we need an execution of C without a subsequent execution of D while an execution B followed by C and not followed by D is required to satisfy the CONSEXMARK of m_1.

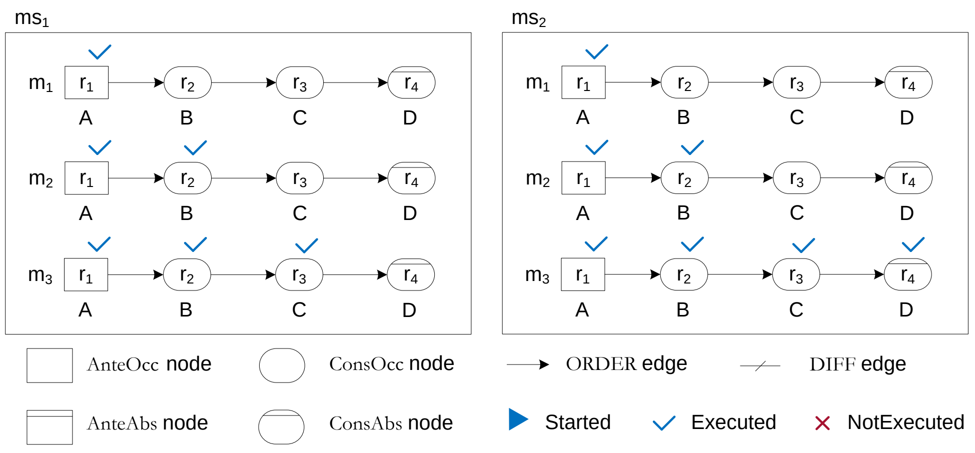


Figure 7.30.: Interpretation of VIOLABLE MARKSTRUCTURES

Even more detailed diagnoses at the granularity of rule activations can be provided when using event-specific MARKSTRUCTUREs as shown in Section 7.4.3.

7.4.3. Regulating compliance checking granularity

The marking rules introduced so far yield *event-independent* MARKSTRUCTUREs as COMPLETED nodes are not assigned any events (cf. Section 7.2.1). Thus, the event pattern captured by a MARKSTRUCTURE can actually occur multiple times in an execution trace as matching events can be caused by multiple activities. From such MARKSTRUCTUREs obtained from executing a CRG over an execution trace, we can deduce whether the trace contains noncompliance. However, since the MARKSTRUCTUREs do not refer to particular activities, they do not reveal the process nodes that are involved in, for example, a compliance violation (cf. Example 7.34). As discussed in Section 7.1.1, fine-grained reports become necessary in order to assist process designers and supervisors to localize and deal with compliance violations.

Example 7.34 (Event-independent MarkStructures):
Recall Example 7.30. Then, the CRG depicted in Fig. 7.27 is activated twice the execution trace. The two activations are both represented by MarkStructure ms_4 in Fig. 7.27. While it is possible to tell that the CRG is satisfied over the trace, we cannot tell from ms_4 which process nodes contribute to the activations. This is because the two activations can both be assigned the equivalence class captured by ms_4 (cf. Section 7.2.1).

Approach MarkStructures can not only be allocated in an event-independent manner but they can also be associated with specific observed events as illustrated by Example 7.35. In event-specific MarkStructures, Completed nodes are assigned events that are associated with the node's execution. As a result, MarkStructures represent patterns formed by specific observed events instead of general observed event patterns. In this case, equivalence classes of event sequences w.r.t. the CRG are unfolded into their particular members.

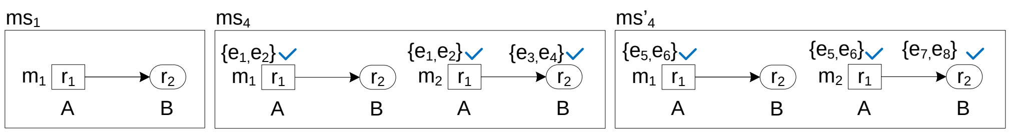


Figure 7.31.: MarkStructures with event assignments

Example 7.35 (Event-specific MarkStructures):
Fig. 7.31 shows the MarkStructures from Example 7.34 obtained when CRG nodes are assigned not only execution states but also events when being marked with Completed. From ms_4 and ms'_4, we can tell the process nodes involved in the rule activations. In case of violations, the source of the violation can be pinpointed, for example, directly in the process model.

In order to obtain event-specific MarkStructures from CRG execution, the marking rules have to adapted to assign not only node states but also events to CRG nodes. This only becomes necessary when a node becomes Completed. Since this is a rather minor modification, we abstain from providing all modified marking rules. Instead, marking rule MR10, the modification of MR5, serves as example. Marking rule MR10 is applied when executing AnteOcc nodes over END events. The modified part is shown in boldface. In particular, an AnteOcc node n will be assigned an EX event (that is obtained through composing the START event associated with n and the END event being processed) if n is marked with Completed.

Marking Rule MR10 (Event-specific marking of AnteOcc nodes over END events):
Let $R = (A, C)$ be a CRG. Then,

$markAnteOccEnd_R : M_R^* \times \mathcal{P}(N_R) \times E^{End} \rightarrow M_R^*$

is a function assigning an ExMark m' of R to an original ExMark $m = ((ns_A, nl_A), (ns_C, nl_C))$ of R , a set Q of AnteOcc nodes to be executed, and an END event e with

$markAnteOccEnd_R(m, Q, e) := m' = ((ns'_A, nl'_A), (ns'_C, nl'_C))$ with:

- $\forall n \in N_R$ with $nt_R(n) =$ AnteOcc:

$$(ns'_A(n), nl'_A(n)) := \begin{cases} (\text{Completed}, \{\mathbf{actExEvent(nl_A(n), e)}\})\ \mathbf{if\ n \in Q} \\ (\text{NotExecuted}, \emptyset)\ if\ n \notin Q \land expectedEnd_R(nl_A, n, e) = \texttt{true} \\ (ns_A(n), nl_A(n)),\ otherwise. \end{cases}$$

- $\forall n \in N_R$ with $nt_R(n) =$ AnteAbs:

  $(ns'_A(n), nl'_A(n)) := (ns_A(n), nl_A(n))$.

- $\forall n \in N_R$ with $nt_R(n) \in \{$ConsOcc, ConsAbs$\}$:

  $(ns'_C(n), nl'_C(n)) := (ns_C(n), nl_C(n))$.

The application of marking rules modified like MR10 results in event-specific MarkStructures. The latter enable insights into particular activations of a CRG and their individual compliance state as illustrated by Example 7.35. This, in turn, enables pinpointing the source of compliance violations, for example, for showing them in process models, and facilitates identifying activation-specific remedies.

7.5. Optimization strategies

As shown in the examples in Section 7.4, each execution iteration processing a START event may result in a growth of MarkStructures and ConsExMarks of MarkStructures due to non-deterministic start of AnteOcc and ConsOcc nodes. For each obtained MarkStructure, the execution and marking rules will be applied in the next execution iteration. The growth in an execution iteration is driven by the amount of AnteOcc and ConsOcc nodes executable over a START event. One strategy to more efficient CRG execution is to reduce the amount of resulting MarkStructures and ConsExMarks of MarkStructures. For this purpose, we introduce strategies to identify MarkStructures and ConsExMarks in Section 7.5.1 that are irrelevant to the further CRG execution and, hence, can be purged without falsifying the compliance state (w.r.t. the notions introduced in Def. 7.5). Purging these structures spares the effort of further processing them in later execution iterations. This can be considered an a posteriori optimization strategy as already created structures are discarded. In contrast to this, another strategy aims at avoiding unnecessary MarkStructures and ConsExMarks during CRG execution in the first place (a priori strategy). As unnecessary unfolding can be caused by starting AnteOcc and ConsOcc nodes over activity executions that do not lead to a match, this can be prevented by processing atomic execution events representing a complete activity execution instead of separate START and END events. Execution and marking rules for executing CRGs over EX events are discussed in Section 7.5.2. Finally, we discuss how the creation of unnecessary MarkStructures and ConsExMarks can be avoided by making use of a priori information on the process in Section 7.5.3.

7.5.1. Purging MarkStructures

In the following, we discuss strategies to purge MarkStructures and ConsExMarks that are not necessary for assessing the compliance with the imposed CRG. Section 7.5.1.1 introduces purging strategies based on the execution state of MarkStructures and ConsExMarks. Section 7.5.1.2 introduces strategies for reducing the ConsExMarks of a MarkStructure based on the notion of domination.

7.5.1.1. Execution state based purging

CRG execution automatically explores different alternatives to form the patterns specified in the CRG using observed events. In particular, each MarkStructure represents a matching attempt for the CRG's antecedent pattern and multiple matching attempts for the CRG's consequence pattern. At some point in the CRG execution, some of these attempts may have proved to not lead to an activated MarkStructure or to a satisfied ConsExMark. Even though these MarkStructures/ ConsExMarks might still be executable over future events, further execution of them will not affect the compliance state. Thus, a strategy to reduce execution cost is to spare the execution of such structures. In this section, we discuss how this can be done by exploiting the compliance notions.

Considerations for ConsExMarks Based on the compliance notion for MarkStructures (cf. Def. 7.5), it is obvious that a violated ConsExMark[16] will not contribute to rendering the associated MarkStructure satisfied. For that reason, violated ConsExMarks that are non-final can be spared from further execution. This can be done by rendering such ConsExMarks final by applying a designated marking rule that assigns NotExecuted to all consequence nodes that are not yet Completed[17] as shown in Example 7.36. This corresponds to pruning the search tree for the corresponding ConsExMarks.

Example 7.36 (Purging violated ConsExMarks):
Fig. 7.32 depicts MarkStructure ms_1 consisting of a violated and a violable ConsExMark. The violated ConsExMark of m_2 contains a ConsOcc node marked as NotExecuted (cf. Def. 7.4) and can be a result of an expected but not matching end event as illustrated in Fig. 7.20. As the ConsExMark of m_2 cannot become violable or satisfied in the further course of the execution, it cannot contribute to satisfying the CRG. Therefore, it can be discarded, for example, by

[16] Recall that a ConsExMark is violated if it contains at least a NotExecuted ConsOcc node or a Completed ConsAbs node (cf. Def. 7.4). A violated ConsExMark cannot be rendered satisfied or violable through observed events.

[17] In this context, it is notable that we can further refine execution state NotExecuted in order to distinguish between nodes not executed in the course of CRG execution and nodes that have been discarded due to other reasons. However, as this is not relevant to the compliance criteria, we abstain from further refining NotExecuted.

removing it from the MarkStructure or rendering it FINAL (as illustrated in Fig. 7.32) to spare unnecessary execution operations in future iterations.

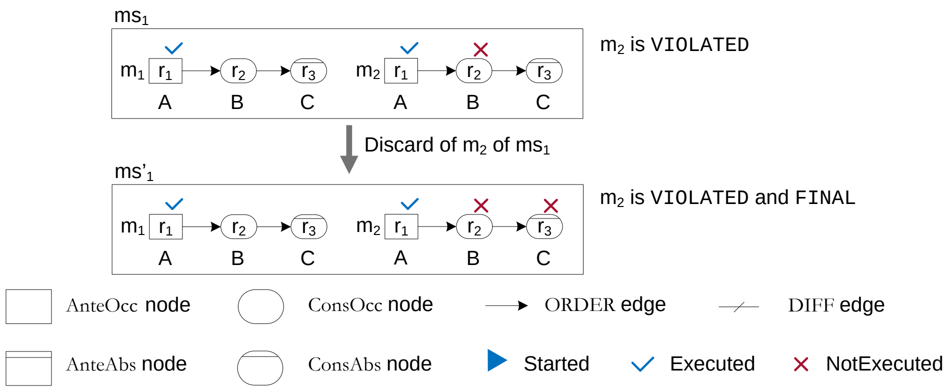


Figure 7.32.: Purging a VIOLATED ConsExMark within a MarkStructure

Considerations for MarkStructures As DEACTIVATED MarkStructures do not represent any rule activations, they do not affect the compliance state. Once a MarkStructure becomes DEACTIVATED (i.e., contains an AnteOcc node marked as NotExecuted or an AnteAbs node marked as Completed), it can no longer become ACTIVATED. For this reason, DEACTIVATED MarkStructures that are NON-FINAL can be excluded from further execution. This can be done by marking all not yet completed nodes of the MarkStructure as NotExecuted as shown in Example 7.37.

Example 7.37 (Purging MarkStructures):
Fig. 7.33 depicts DEACTIVATED MarkStructure ms_1. AnteOcc node r_1 marked as NotExecuted can result from an expected but not matching END event as shown in Fig. 7.25. Then, ms_1 can be purged by assigning execution state NotExecuted to all not yet completed nodes to render it FINAL. Note that the not yet completed nodes of ms_1 cannot become executable due to the necessary conditions (e.g., all AnteOcc predecessors must be Completed). However, a DEACTIVATED MarkStructure may still contain executable nodes (e.g., image that the CRG of ms_1 contains a further AnteOcc node that has no predecessors).

As shown in Example 7.37, a way to exclude a DEACTIVATED MarkStructure from further execution is just to render it FINAL. This can be done by applying a designated pruning rule.

7.5.1.2. Domination among ConsExMarks

As shown in Section 7.5.1.1, VIOLATED ConsExMarks and DEACTIVATED MarkStructures can be excluded from further CRG execution to reduce computational cost. In addition to that, we can provide purging strategies for ACTIVATED and ACTIVATABLE MarkStructures.

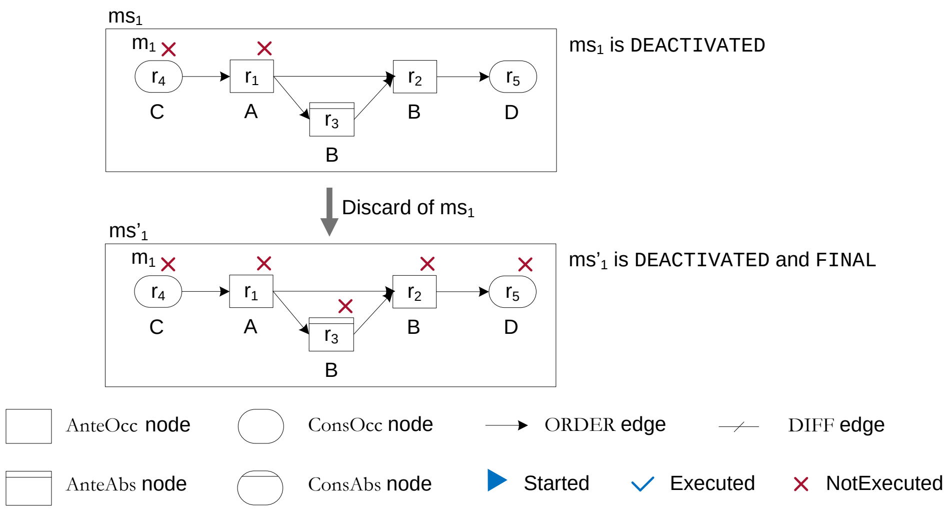


Figure 7.33.: Purging a DEACTIVATED MARKSTRUCTURE

A MARKSTRUCTURE can be associated with multiple CONSEXMARKS each of which corresponds to an attempt to spot the CRG's consequence pattern in the execution trace. Due to nondeterministic start of CONSOCC nodes, a MARKSTRUCTURE may contain both parent and child CONSEXMARKS (or EXMARKS) at the same time. As a result, a MARKSTRUCTURE may contain CONSEXMARKS, whose further execution can provide additional details on the compliance situation but is not necessary for assessing the compliance state of the MARKSTRUCTURE. Such a case is described in Example 7.38. The basic idea is to purge CONSEXMARKS that are *dominated* by other CONSEXMARKS and, hence, can be discarded without falsifying the compliance state of the overall MARKSTRUCTURE w.r.t. the compliance notions from Def. 7.5.

Example 7.38 (Domination among VIOLABLE CONSEXMARKS):
Fig. 7.34 depicts the execution of MARKSTRUCTURE ms_1 over a few events. Due to nondeterministic start of CONSOCC node r_2, the resulting MARKSTRUCTURE ms_2 contains CONSEXMARKS with r_2 being in different execution states.

In ms_3, m_1 is *dominated* by m_2 as the latter is closer to becoming SATISFIED. In particular, if further execution of m_1 leads to a SATISFIED CONSEXMARK, m_2 will also become SATISFIED as for satisfying m_2 only the execution of C is required. Hence, being dominated by m_2, m_1 can be purged from ms_3 as it is not necessary to correctly assess the compliance state. In contrast to ms_3, the purged MARKSTRUCTURE ms_3' is not executable over e_3.

Clearly, a SATISFIED CONSEXMARK dominates not SATISFIED CONSEXMARKS. Consequently, once a MARKSTRUCTURE is associated with at least one SATISFIED CONSEXMARKS, its not yet SATISFIED CONSEXMARKS can be discarded. In addition, it is obvious that VIOLATED CONSEXMARKS are dominated by non-VIOLATED CONSEXMARKS. These two cases are trivial and can be easily integrated into CRG execution and, therefore, will not be discussed in the following. Instead, we focus on domination among VIOLABLE CONSEXMARKS as illustrated by Example 7.38.

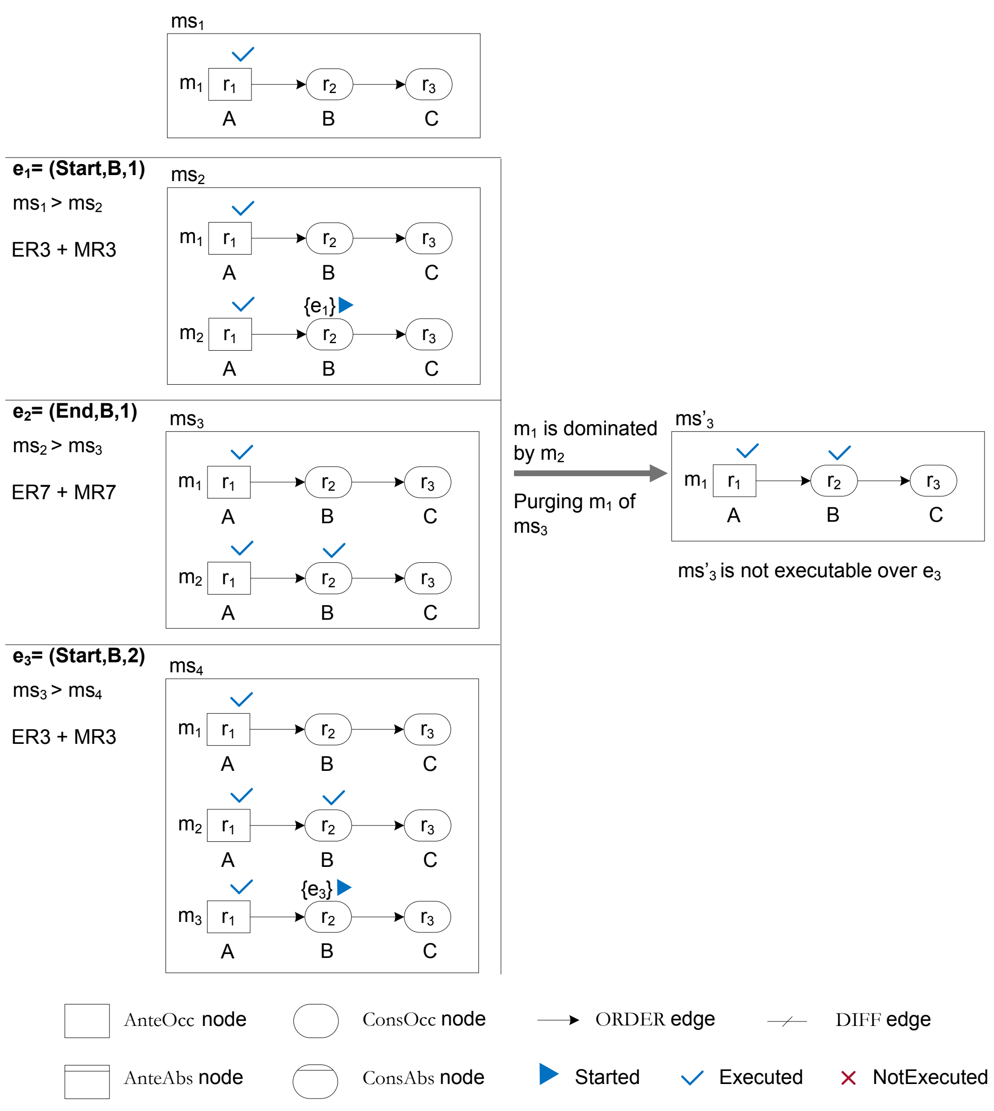


Figure 7.34.: Domination among VIOLABLE CONSEXMARKS

How to identify dominating CONSEXMARKS among VIOLABLE CONSEXMARKS? Intuitively, a VIOLABLE CONSEXMARK cm' is dominated by another VIOLABLE CONSEXMARK cm if cm constitutes a descendant CONSEXMARK of cm' that is "closer" to becoming SATISFIED. With respect to execution traces, this means that every trace (suffix) transforming cm' into a SATISFIED CONSEXMARK also transforms cm into a SATISFIED CONSEXMARK. This intuition formalized in Def. 7.11. It is based on three conditions ensuring the domination of cm over cm':

i) The CONSOCC nodes of cm that are not yet marked as COMPLETED constitute a subset of CONSOCC nodes not yet marked as COMPLETED of cm'. In other words, cm requires less activity executions to occur in order for all CONSOCC nodes to be marked as COMPLETED than cm'.

ii) This condition demands that cm does not contain any STARTED node that is not also

STARTED over the same START events in cm'. This ensures that cm cannot be invalidated (i.e., become VIOLATED) by an END event that would not also invalidate cm'. In addition, together with condition i), condition ii) ensures that cm is a descendant ConsExMark of cm'.

iii) A ConsExMark can be invalidated (i.e., become VIOLATED) not only through not matching END events but also through activity executions matching with ConsAbs nodes (i.e., a ConsAbs node marked as Completed). Condition iii), therefore, demands that cm cannot be invalidated by the occurrence of any activity execution that would not also invalidate cm'.

While condition i) demands that cm requires the execution of less activity executions than cm', the conditions ii) and iii) together ensure that cm cannot be invalidated by events that would not also invalidate cm'. Altogether, this ensures that cm will be satisfied over any trace suffix that also satisfies cm'.

Definition 7.11 (Domination among ConsExMarks)
Let R be a CRG and let $ms = ((ns_A, nl_A), \{(ns_C^1, nl_C^1), \ldots, (ns_C^k, nl_C^k)\}) \in MS_R^*$ be a MarkStructure of R obtained from execution of R over a (partial) trace. For two VIOLABLE ConsExMarks $cm = (ns_{cm}, nl_{cm})$ and $cm' = (ns_{cm'}, nl_{cm'})$ of ms, we say

cm dominates cm' $\Leftrightarrow$ conditions i) - iii) are satisfied with:

i) $\{n \in N_C \mid nt_R(n) = \texttt{ConsOcc} \wedge ns_{cm}(n) \neq \texttt{Completed}\} \subset \{n \in N_C \mid nt_R(n) = \texttt{ConsOcc} \wedge ns_{cm'}(n) \neq \texttt{Completed}\}$,

ii) $\forall n \in N_R$ with $ns_{cm}(n) = \texttt{Started}$ holds: $(ns_{cm}(n), nl_{cm}(n)) = (ns_{cm'}(n), nl_{cm'}(n))$, and

iii) $\neg(\exists n \in N_R :$
$(nt_R(n) = \texttt{ConsOcc} \wedge ns_{cm}(n) = \texttt{Completed} \wedge ns_{cm'}(n) \neq \texttt{Completed} \wedge$
$(\exists n' \in N_R : nt_R(n') = \texttt{ConsAbs} \wedge ns_{cm}(n') \neq \texttt{NotExecuted} \wedge n' \in succConsAbs_R(n))))$.

Example 7.39 (Domination among VIOLABLE ConsExMarks):
Consider MarkStructure ms_2 from Fig. 7.34. Then, the ConsExMark of m_1 is not dominated by the ConsExMark of m_2 although m_2 is a child ConsExMark of m_1. This is because the ConsExMark of m_2 can still become VIOLATED through r_2. In fact, it is easy to construct an execution trace satisfying the ConsExMark of m_1 but not the ConsExMark of m_2.

Further examples are depicted in Fig. 7.35. The ConsExMark of m_2 does not dominate the ConsExMark of m_1 due to condition iii). In particular, the ConsExMark of m_2 can become invalided by the next execution of D while this would not invalidate the ConsExMark of m_1. As the same applies to the ConsExMark of m_3, m_1 is also not dominated by m_3. However, m_2 is dominated by m_3 and both can be invalidated by D. The ConsExMark of m_4 cannot be invalidated by occurring events. It therefore dominates the ConsExMarks of m_1 and m_2 as m_4 only requires an execution of E to become SATISFIED, which is a subset of the activity executions required to render the other ConsExMarks SATISFIED. However, the ConsExMark of m_4 does not dominate the ConsExMark of m_3 as condition i) does not apply. The ConsExMark of m_5 neither dominates m_1 nor m_2 nor m_3 as it can be invalidated by an END event that would not invalidate the other ConsExMarks. Thus, condition ii) does not apply. Finally, the ConsExMark of m_6 dominates all other ConsExMarks shown in Fig. 7.35.

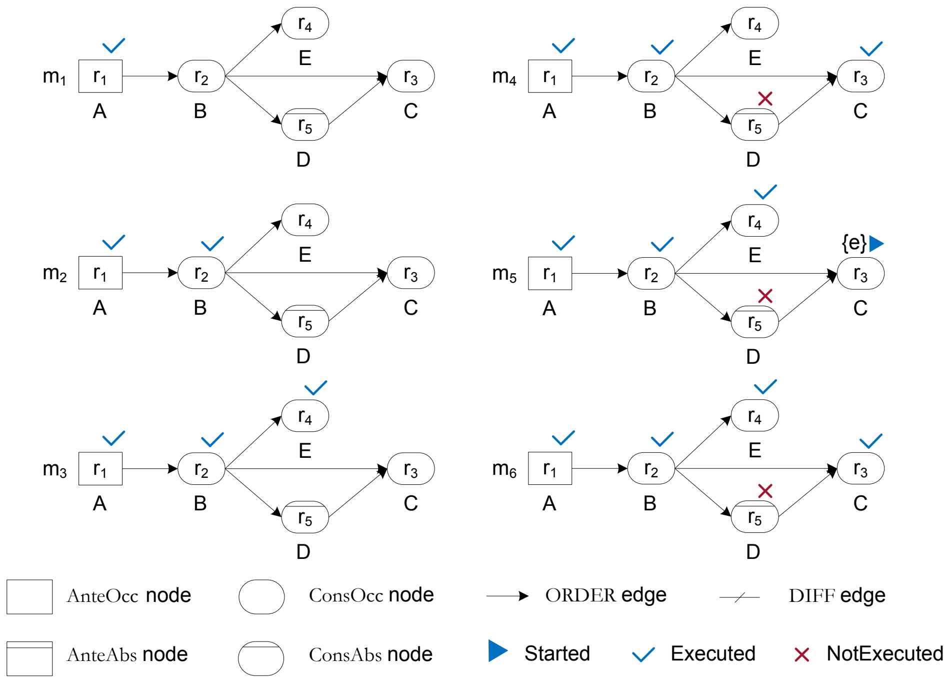


Figure 7.35.: MarkStructures of a CRG associated with different ConsExMarks

Based on the notion of domination among VIOLABLE ConsExMarks, it becomes possible to purge VIOLABLE ConsExMarks for MarkStructures obtained from CRG execution. In particular, only those ConsExMarks are retained that are not dominated by another ConsExMark of the MarkStructure.

Definition 7.12 (Purging ConsExMarks based on domination)
Let R be a CRG and let further $ms = ((ns_A, nl_A), CM_{ms}) \in MS_R^*$ with $CM_{ms} = \{(ns_C^1, nl_C^1), \ldots, (ns_C^k, nl_C^k)\}$ be a MarkStructure of R obtained from execution of R over a trace.

Then, $purge(ms) = ms' := ((ns_A, nl_A), CM_{ms'})$ where
$CM_{ms'} := \{cm \in CM_{ms} \mid \neg(\exists\ cm' \in CM_{ms} :\ cm' \text{ dominates } cm)\}$.

For applying Def. 7.12, pair-wise checking for domination among the ConsExMarks becomes necessary. Continuously purging ConsExMarks in the course of CRG execution confines the unfolding of paths caused by nondeterministic start of ConsOcc nodes.

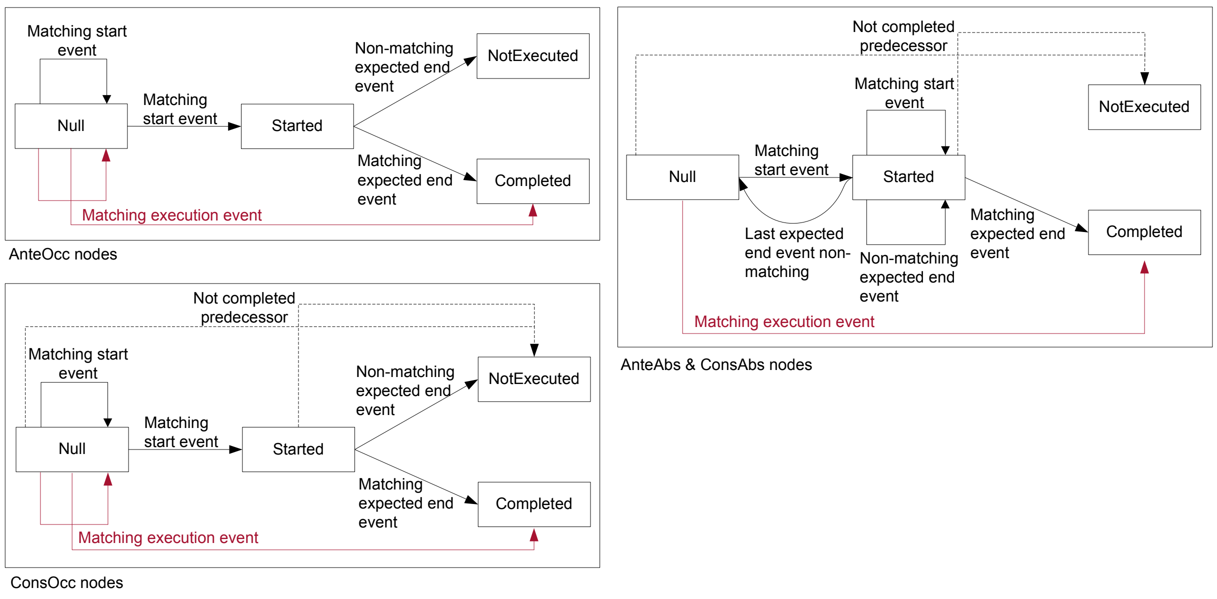


Figure 7.36.: Node state transitions when processing EX events

7.5.2. Execution of compliance rule graphs over EX events

So far, we introduced execution and marking rules for dealing with START and END events. The granularity of START and END events particularly becomes necessary when CRGs are executed over traces containing overlapping activity executions. The latter can occur, for example, due to parallel branches in the process model or when processes are executed concurrently, for example, in distributed scenarios.

In purely sequential scenarios (e.g., execution of a process model that is free of parallel branches), the granularity of START and END events is not necessary in order to reflect the ordering of activity executions. For such scenarios, the pair of START and corresponding END event can be subsumed by an atomic EX event (cf. Section 5.2.3). Processing START and END as an atomic unit has the advantage of preventing unnecessary start of AnteOcc and ConsOcc nodes over activity executions that finally turn out as not matching due to not matching end conditions. In addition, ConsOcc nodes without direct ConsAbs successors can be executed deterministically over a matching EX event based on the considerations on domination among ConsExMarks. As a result, processing EX events instead of separate START and END events can reduce the amount of MarkStructures and ConsExMarks produced in execution iterations.

The execution and marking rules to process EX events can be defined in similar manner as for START and END events and are given in Appendix A.2. Fig. 7.36 depicts the execution state transitions of CRG nodes enriched with the transitions caused by EX events based on the above considerations.

7.5.3. Exploiting a priori information

By exploiting a priori information about the process/the execution trace, unnecessary nondeterministic execution and, thus, unnecessary exploration can be avoided a priori.

Absence information Information on the absence of certain activity executions that are associated with nodes of an imposed CRG can be exploited to prevent unnecessary execution operations. If we are aware that no activity executions of an execution trace will match a particular AnteOcc node of a CRG R, we can spare executing R as R cannot become ACTIVATED over the execution trace. Similarly, if we know that a ConsOcc node will not match any activity execution contained in the execution trace, executing the corresponding consequence CRG will not lead to a SATISFIED ConsExMark. Thus, the corresponding consequence CRG can be excluded from execution. For absence nodes, we can exploit a priori information on the absence of activity executions to "refine" an imposed CRG. If we, for example, know that no activity execution matching an AnteAbs or a ConsAbs node is present in the execution trace, the respective node can be virtually removed from the CRG.

Occurrence information In a similar manner, a priori information on the frequency of an activity execution of a certain type can also be exploited to achieve more efficient CRG execution. Being aware that, for example, four activity executions of a particular type are contained in an execution trace, it becomes possible to tell during CRG execution that no such activity executions will occur after all four activity executions have already been observed. In addition, when executing an AnteOcc or a ConsOcc node over the fourth activity execution of that type, the corresponding nodes can be executed deterministically as we know that no further such activity execution will be observed in the remaining execution trace.

7.6. Correctness of compliance rule graph operationalization

In this chapter, we introduced MarkStructures to capture compliance states w.r.t. an imposed CRG. Moreover, we defined CRG operational semantics in the form of execution and marking rules. They allow for altering the MarkStructures in accordance to observed events such that the compliance state can be incrementally updated. Altogether, the compliance state of a CRG in an execution trace is reflected by a set of MarkStructures for which we provided notions for their evaluation. In Section 7.6.1, we argue why CRG operationalization as introduced in this chapter correctly implements the formal semantics of CRGs. Recall that we focused on CRGs instead of CRG composites so far. That is why we describe how the execution and marking rules can be adopted for operationalization of CRG composites in Section 7.6.2.

7.6.1. Operationalization of compliance rule graphs

Recall RFs and their interpretation over execution traces that capture the formal semantics of CRGs (and CRG composites) described in Section 6.3. For correctness, it has to be ensured that this semantics is reflected in the CRG operationalization. In particular, the execution of the CRG over the trace must reveal that the CRG is satisfied over the trace if and only if the formal interpretation of the RF of the CRG over an execution trace will reveal the same.

According to the formal semantics of CRGs, an execution trace violates a CRG iff there is at least one activation of the CRG that is not satisfied (i.e., if there is an occurrence of the antecedent pattern without the associated consequence pattern in the trace). In that case, the interpretation of the RF based on that trace would not yield a model for the RF. If there is no activation of a CRG in a trace at all, the RF will be satisfied over the trace due to its $\forall(\phi \rightarrow \psi)$ rule structure (cf. Def. 6.7). To implement this, it is, on the one hand, necessary that CRG execution is able to reveal if a compliance violation is contained in the trace. This means that the set of MarkStructures obtained from executing the CRG over the trace must contain a violated MarkStructure in that case. On the other hand, if no violation is contained, no violated MarkStructures must be yielded through CRG execution. Altogether, the structure of a RF can be summarized as shown below where the variables are yielded through CRG nodes of associated types and the conditions on these variables are yielded through relations between nodes and the node profiles:

$$\forall AOvars((Conds_{AOvars} \wedge \neg(\exists AAvars(Conds_{AAvars})))$$
$$\rightarrow \exists COvars(Conds_{COvars} \wedge \neg(\exists CAvars(Conds_{CAvars}))))$$

The domain of variables for the interpretation of a RF is constituted by the activity executions contained in an execution trace. For these, the formal interpretation derives actual relations and properties that are necessary to evaluate the truth of a RF. It is obvious that based on the activity executions contained in the trace, a search procedure can be implemented following the formal interpretation that checks the truth of a RF by finding all combinations of activity executions matching the antecedent pattern and checking the rule consequence for each such combination.

Instead of trying each combination of activity executions (as would be the case for a naive implementation of the formal interpretation), the CRG operationalization exploits the ordering relation of nodes to perform a search for activity executions matching the antecedent and consequence patterns in one pass of the execution trace. This is integrated into the procedure through start preconditions on predecessor nodes of certain types (e.g., a node can only be started if all AnteOcc predecessors are completed). Thus, less combinations have to be checked. diff relations are enforced by ensuring that no pair of nodes is started over the same activity execution in a trace.

Due to nondeterministic start of AnteOcc nodes, each occurrence of the antecedent pattern can be detected. To detect activity executions matching AnteAbs nodes that may prevent a rule activation, each potentially matching activity execution is checked. Thus, each occurrence of the antecedent pattern (corresponding to AnteOcc nodes and AnteOcc variables of the RF) will ultimately be reflected by an activated MarkStructure of the compliance state. For each occurrence of the antecedent pattern, it is also searched for a corresponding consequence pattern during CRG execution over the trace. Nondeterministic start of ConsOcc nodes ensures that activity executions matching the pattern specified by ConsOcc nodes will be found if they are contained in the trace. As each activity execution potentially matching a ConsAbs node is checked, it is ensured that the occurrence of a matching activity execution will detected. In order to identify occurrences of the antecedent and the consequence pattern in one pass over the execution trace, the execution of ConsOcc and ConsAbs nodes follows the ORDER relations. Thus, if an occurrence of the consequence pattern for a particular rule activation is contained in the trace, the corresponding MarkStructure will be associated with a SATISFIED ConsExMark (i.e., the MarkStructure will be SATISFIED). Otherwise, the MarkStructure will become VIOLATED by the end of CRG execution.

Altogether, CRG operationalization constitutes an implementation of the CRG formal semantics in one pass over the execution trace by exploiting both the ordering of events contained in the trace and the ordering of CRG nodes. To realize that, MarkStructures are used to store the information found in each step. Thus, the MarkStructures obtained from CRG execution can be used to derive meaningful information on the compliance situation, for example, in case of compliance violations.

7.6.2. Operationalization of compliance rule graph composites

So far, we focused on the operationalization of CRGs rather than CRG composites (i.e., compliance rules with multiple consequence patterns) for intelligibility reasons. Nevertheless, MarkStructures and the CRG operational semantics can be easily adapted for dealing with CRG composites. For that purpose, MarkStructures can be extended such that they can be associated with ConsExMarks of different consequence patterns as illustrated by Fig. 7.37.

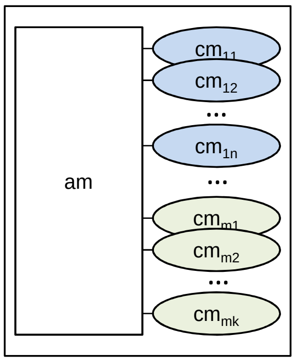


Figure 7.37.: Structure of a MarkStructure associated with ConsExMarks of multiple rule consequences

Due to the disjunctive semantics of the multiple consequence patterns of a CRG composite (i.e., one of them has to apply in order to satisfy the compliance rule), the different consequences can be executed separately using the introduced execution and marking rules. Then, the criteria to evaluate MarkStructures from Def. 7.5 can be adopted for dealing with ConsExMarks referring to different consequence CRGs. An ACTIVATED MarkStructure is considered SATISFIED, if it is associated with a SATISFIED ConsExMark.

7.7. Discussion

In this chapter, we described the operationalization of CRGs as a means for compliance verification. It enables to check compliance of an execution trace with an imposed CRG in a single pass. To capture reachable compliance states, we introduced so-called MarkStructures. The latter utilize execution state markings for CRG nodes, such as Running or Completed, to indicate behavior relevant to the respective CRG observed during verification. Thus, each compliance state with respect to a CRG can be represented by a set of MarkStructures. The execution and marking rules described in Section 7.3 enable to update the compliance state when a new event is processed. They particularly ensure that each activation of the CRG contained in the trace explored is represented by an ACTIVATED MarkStructure. Based on the semantics of CRG nodes and of node execution states, we provided notions to formally assess the MarkStructures in Section 7.2.2. Being represented by MarkStructures, a reached compliance state is constituted by the individual states of its MarkStructures. Hence, if the detected compliance state reveals a VIOLATED MarkStructure, the trace explored enables noncompliance.

As motivated in Section 7.1.1, key to the practical application of a compliance checking framework is its ability to provide explanations for compliance violations. By directly exploiting the structure of CRGs, MarkStructures allow for representing compliance states in an interpretable manner. This, in turn, enables the derivation of not only explanations for compliance violations but also valuable information that can be utilized to provide support to comply. As shown in Section 7.4.2, we can, for example, easily derive sequences of activities that become necessary in order to satisfy a compliance rule in an effective compliance state.

Using the approach proposed in this chapter, compliance checks can be conducted at different levels of granularity as described in Section 7.4.3. When being used in an event-independent manner, MarkStructures constitute an equivalence class with respect to the corresponding CRG for different event sequences contained in a trace. Thus, multiple activations of a CRG contained in a trace will be associated with the same MarkStructure when they are in the same state of enforcement. This, for example, suffices to provide compliance reports on the overall enforcement of compliance rules. Whenever awareness of the particular situations in which violations occur is necessary in order to adequately deal with violations, event-specific MarkStructures can be employed to encode compliance states. They enable the precise identification of all activations of a CRG within an execution trace. This together with the individual compliance state of these rule activations provide the basis for detailed compliance reports. The latter, in turn, enable process designers and supervisors to locate violations, evaluate their extent, and decide on adequate remedy strategies.

We further introduced different ideas to optimize the CRG execution with respect to efficiency in Section 7.5. These aim at purging MarkStructures, thus, pruning the search tree unfolded during CRG execution. At design time, the proposed approach can be applied to explore a process model and to detect which compliance states with respect to imposed CRGs a process model is able to yield. As the CRG operationalization is tailored towards incremental application, process instances can be monitored by updating the effective compliance state for each new event observed during process execution. In Chapter 8, we will describe design and runtime verification based on the proposed approach in more detail. In the following, we first discuss alternative approaches for verifying compliance in Section 7.7.1. Section 7.7.2 addresses the extension of CRG operationalization and further issues around CRG operationalization are addressed in Section 7.7.3.

7.7.1. Related work

In the following, we describe different approaches in literature that can be adopted to verify compliance with CRGs and elaborate on why the operationalization of CRGs as introduced in this chapter is more suitable.

Automaton-based approaches In literature, we often come across approaches using automatons for compliance checking. This goes back to research on enforcing transactional dependencies in workflows [Kle91, ASSR93, Sin96, Sin97]. As rules to be checked are typically not modeled by means of automatons, transformations become necessary. In case of compliance monitoring, the automaton has to reach an accepting state, if the compliance rule to be checked is satisfied [MMWA11, PA06, Pes08, SFV$^+$12]. Such an automaton can be used to monitor running processes based on the observed events. By analyzing whether an accepting state is still reachable from the current state, it can be determined whether the imposed rule can still be complied with. In case of process model verification, an automaton representing the negated property can be used to detect noncompliant traces. For example, in explicit LTL model checking, an automaton is generated from the negated LTL property that can be used to check whether the model and the automaton enable a common trace [CGP99, BBF$^+$01]. If so, a violation is detected and a counterexample can be provided.

While CRGs are not directly transformed into an automaton for conducting compliance checks, we can still draw parallels between automaton-based approaches and the operationalization of CRGs. As a reached compliance state w.r.t. a CRG can be represented by a set of MarkStructures, for a CRG R the state space is roughly given by $\mathcal{P}(MS_R^*)$ [18]. The transitions between the states are defined by the execution and marking rules. In our approach, the states are not black boxes but provide meaningful information as they refer to the structure of the CRG. Thus, when monitoring compliance at process runtime, not only violations can be detected but also measures to proactively satisfy rule activations can be derived as shown in [LRMKD11]. In particular, it can be easily derived for each MarkStructure whether it can still become satisfied without additional analysis. For compliance checks at design time, the CRG operational semantics can be applied in different ways to detect all violations contained in a process model as we will show in Chapter 8. Since no transformation into other representations become necessary with our approach, diagnoses can be derived and provided based on the specific modeled compliance rule. In addition, the flexibility of checking compliance at different granularity levels is a further benefit of our approach.

[18] This approximation holds for event-independent execution over EX events. It should be noted that the actual state space is smaller than that as not all MarkStructures in MS_R^* can be generated when executing a CRG due to execution preconditions and pruning rules.

Violation patterns In [AW09, AWW09], Awad et al. introduce an approach for querying the process model for patterns of compliance violations (referred to as anti-patterns). Motivated by the cumbersome transformation of the counterexample provided by a model checker back into a representation that can be utilized to visualize the detected violation, Awad et al. decided to base their visualization approach on anti-pattern queries. In [AW09], they focus on a set of rule patterns such as *before scope presence* or *response* based on the work of Dwyer and Corbett [DAC99]. For these rule patterns, they provide violation patterns in the form of BPMN-Q queries, a language for querying the process structure [Awa07]. If such a structure that leads to violation of the imposed rule is detected by BPMN-Q, it can be visualized within the process model. In [AWW09], the rule patterns are extended with data-aware conditions and violation patterns for them are provided.

At runtime, the (not yet complete) execution trace of a running process instance can be queried for violation patterns in order to detect noncompliance. For that purpose, existing frameworks and technologies such as *complex event processing* (CEP) [JML09] can be applied. In their work, Giblin et al. [GMP06] developed the REALM rule model (cf. Section 3.3.4). For REALM patterns, such as "y must occur within time t after x", they provide transformations into ACT correlation rules, which can be used for detecting relevant event patterns. Event processing technologies are further used by numerous compliance monitoring frameworks to detect violations, e.g., in the COMPAS project [HMZD11, BDL+10] (cf. Section 3.3.3). In addition, commercial *security incident and event monitoring* (SIEM) tools exploit event processing to detect patterns that indicate critical incidents.

To adopt this approach for checking compliance with CRGs at design and runtime, violation patterns have to be provided for CRGs. Following [ASW09, AWW09, WZM+11, WPDM10, WPD+11], one option is to provide violation patterns for a particular set of CRGs. However, this would not suffice to check compliance with arbitrary CRGs. As complex CRGs can be violated in multiple ways, automatic derivation of all violation patterns of arbitrary CRGs becomes necessary. This is not required when using CRG operational semantics as violation patterns are indirectly encoded by VIOLATED MARKSTRUCTURES.

Further approaches In [MMC+11], the event calculus is utilized to formalize ConDec / DECLARE [PA06] compliance rules. In [ACG+08, MPA+10], $\mathcal{S}$CIFF, an extension of logic programming, is used to formalize and reason about compliance rules. The operational specification of $\mathcal{S}$CIFF is constituted by a proof procedure, which can be used to implement compliance checks [CMMS07a] (cf. Section 3.3.1). Adopting these approaches for checking CRGs, however, does not suggest itself as CRG operationalization is tailored towards the graph-based nature of CRGs and can exploit the latter to enable compliance diagnosis.

Model checkers, such as SPIN, are used for compliance verification of process models by a multitude of approaches in literature, for example [FPR06, FUMK06, FESS06, FESS07, LMX07, GK07, ADW08, KLRM+10, FS10, KGE11]. In order to enable the verification of CRGs through a model checker, CRGs first have to be transformed into a representation that can be processed by the model checker, for example, a temporal logic specification in LTL. In addition, the output of the model checker, in many cases a single counterexample, would have to be transformed back in order to derive a diagnosis [KLRM+10, AW09]. In addition to that, model checkers are only suitable for verifying process models against imposed rules. In order to support compliance monitoring (when the process model is unknown), additional mechanisms for enabling incremental compliance checks are required.

7.7.2. Extensions of compliance rule graph operational semantics

As described in Section 6.4.2, the CRG language can be extended in order to enable the specification of more sophisticated compliance rules, for example, containing data or metric time relations. To support these, the operational semantics will have to be extended accordingly.

Data relations In order to verify such CRGs enriched with data relations as described in Section 6.4.2, the operational semantics has to be adapted. Generally, a data relation cannot be evaluated until the nodes related are associated with an activity execution. Consider, for example, the compliance rule requesting that the settlement of an account has to be approved beforehand by an authorized personnel, who is not the beneficiary of the settlement. Then, this compliance rule can be captured by a CRG consisting of an AnteOcc node r_1 (for the settlement of the account) with a predecessor ConsOcc node r_2 (for the approval by the authorized personnel) and a relation between these two nodes ensuring that the agent performing r_2 is not associated with r_1. Thus, when observing an approval activity in the trace, it is not yet possible to tell whether this activity execution will satisfy an activation of the above compliance rule as the data relation cannot be assessed until the rule becomes activated. Such cases can be dealt with by adopting nondeterministic node execution as already used in CRG. In that manner, CRG operational semantics can be extended to deal with data relations. While this is not in the focus of this thesis, we implemented support for certain data relations in the SeaFlows Toolset, our proof-of-concept prototype, that will be described in Chapter 10.

Metric time conditions Compliance rules often involve time conditions [MPA$^+$10]. Besides conditions on the duration of activity executions, deadlines, or other metric time properties that can be evaluated the same way as data conditions (cf. Section 6.2.1.2), we often encounter temporal relations in compliance rules involving quantitative time. Consider, for example, a guideline from the medical domain requesting that after an invasive treatment, it is not only required that a aftercare for the patient takes place but the aftercare should also take place within one day after the treatment to ensure proper care. While qualitative temporal relations (such as predecessor and successor relations) are an integral part of CRGs, quantitative time is not yet dealt with. Frequent time patterns in process-aware information systems are collected in [LWR09, LWR10].

As mentioned in Section 6.2.1.2, dealing with time conditions necessitates an extension of the logical model, particularly of the notion of execution traces, events, and activity executions. These have to be extended by a notion of time (e.g., by introducing timestamps) in order to enable formal interpretation of time conditions. Then, time relations can be supported by using timer events. In particular, timer events (e.g., one day after X) can be assigned to CRG nodes like activity executions, which enables complex time relations. In [LRMKD11], we showed how this can look like. Time relations may also be introduced in analogy to data relations. To evaluate the time relations, we envision a designated time component as, for example, developed in the ATAPIS project [Lan08, LWR09, LWR10] interacting with the compliance checking component. Such a time component may not only evaluate the truth of time relations but may also handle forecasts (for example, for deadlines). As this is beyond the scope of our work, we leave this to future research.

7.7.3. Further issues

The operational semantics of CRGs is tailored towards dealing with activity executions matching with CRG nodes. However, as previously mentioned, CRGs may also be used to define rules on events rather than on activity executions. This enables using CRG on a different level of granularity. The execution and marking rules for EX events described in Section 7.5.2 (cf. Appendix A.2) can be directly adopted for dealing with CRGs on events.

For implementation of the CRG operational semantics, it can be useful to store the event assignment and the execution state assignment of CRG nodes of MarkStructures in separate data structures. This enables to merge different event-specific MarkStructures with the same node execution states that only differ in the event assignments. Generally, we can imagine sophisticated data structures for the practical implementation of CRG operationalization.

8 A compliance checking framework based on compliance rule graphs

In Chapter 6, we introduced the CRG language as a means to capture compliance requirements imposed on business processes. Based on their formal semantics, we provided execution and marking rules for operationalizing CRGs in Chapter 7. They enable the incremental verification of execution traces against CRGs. This provides the means for exploring process models and for monitoring running process instances with respect to compliance. Key to this approach is the exploitation of the graph structure of CRGs for representing compliance states in a transparent and interpretable manner. Thus, it becomes possible to derive not only explanations for compliance violations from detected compliance states but also valuable information that can be utilized to provide support to comply.

As described in Chapter 2, a comprehensive compliance checking framework is supposed to support the complete process lifecycle by providing adequate mechanisms to verify and foster compliance. In particular, the framework has to support compliance checks of the process model at process design and compliance monitoring of running process instances at runtime. With the CRG language and its operationalization introduced so far, we have provided the fundament for such a compliance checking framework. In this chapter, we show how the latter can be realized based on the CRG approach. In Section 8.1, we recall the requirements on comprehensive compliance checking discussed in Chapter 2 and describe the basics of the SeaFlows compliance checking framework. Section 8.2 elaborates on the application of the proposed concepts for checking a general model specification against imposed CRGs. This, in turn, constitutes the fundament for verifying process models as will be described in Section 8.3. Section 8.4 addresses compliance monitoring of process instances. Finally, the chapter closes with a discussion of related work and further issues and applications of the proposed compliance checking framework in Section 8.5.

8.1. Introduction

Recall the vision of a comprehensive compliance checking framework covering both process design and process runtime that we described in Section 2.2.1. In the following, we describe briefly how the concepts introduced in this thesis can be used to realize design and runtime compliance checks before going into details.

8.1.1. Process design time

A process model typically enables multiple different process executions. Compliance verification of process models aims at the detection of process executions violating imposed rules. For that the process model has to be explored to analyze the behavior encoded by the model with respect to compliance. The exploration of the process model and the compliance verification can be conducted in one step as the CRG execution can be applied on-the-fly when simulating the execution of a process model. To illustrate the application of CRG execution for compliance verification in an intelligible manner, however, we opted for showing the verification in two separate steps as shown in Fig. 8.1.

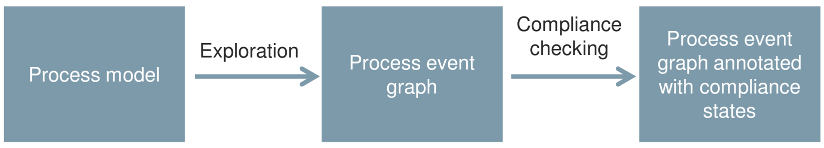


Figure 8.1.: Overall compliance checking procedure at process design time

The process model to be verified is first explored. We use *process event graphs* (PEGs) as a simple and general representation for capturing the behavior enabled by the process model independently from the process description language employed. A PEG is an automaton-like structure in which nodes are assigned events of the process execution (cf. Section 5.2.6). In this chapter, not the exploration of the process model is in focus but the verification of the process given an equivalent state space representation (in this case an equivalent PEG)[1]. In the second step, the equivalent PEG is checked for compliance. This is done by executing the imposed CRGs over the paths of the PEG. This results in a PEG annotated with reachable compliance states. How this is accomplished is described in Section 8.2.

Based the compliance states associated with the PEG, we are able to reveal compliance violations. By using event-specific MarkStructures for representing compliance states, the compliance checking framework can provide diagnoses for the particular activations of a CRG contained in a process and their individual compliance state. Seizing this information, the involved activities can be localized directly in the process model making it easier to pinpoint and assess the extent of compliance violations.

[1]We will discuss a variety of abstraction strategies that aim at reducing the size of the PEG by abstracting from irrelevant behavior and structures in Chapter 9. For that we will relax the notion of equivalence between process model and PEG.

8.1.2. Process runtime

Compliance monitoring refers to the observation of the execution of process instances for timely detection of compliance violations. By incrementally executing imposed CRGs over the events observed during process execution, it becomes possible to be constantly aware of the effective compliance state with respect to the CRGs. Thus, compliance violations becoming manifest in a process instance's execution trace can be spotted in a timely fashion. Technically, the monitoring procedure is comparable to the execution of CRGs over an (evolving) execution trace. As this only relies on observed execution events, the approach can be applied to process instances that are not executed based on a predefined process model (e.g., ad-hoc process instances).

When monitoring process instances that are based on a predefined process model, not only its past but also its future behavior can be anticipated. As the execution proceeds, only a subset of the predefined future activities may still be executable in the running process instance. Due to that, CRGs that are not enforced in the underlying process model may turn out to be enforced or to be surely violated in the process instance in the meantime. Being aware of that in time can be valuable as it allows for timely remedies. To support this, the behavior encoded in the model can be exploited for updating compliance predictions made at the process model level.

Altogether, a crucial basis to the compliance checking framework is the verification of a `PEG` against imposed CRGs. Therefore, we first provide fundamentals on checking a `PEG` for compliance with a CRG in Section 8.2. Based on that, we further show how process model verification and process instance monitoring can be supported.

8.2. Compliance checking of process event graphs

In this thesis, we use `PEG`s as a general representation of the possible behavior encoded by a process model or a process instance. A `PEG` typically captures multiple execution traces as paths from the `PEG`'s start to a `PEG` end node (cf. Section 5.2.6). Obviously, a CRG is not enforced in a `PEG` if the latter contains at least one noncompliant trace. There mere detection of compliance violations is, however, not sufficient. In order to support the requirements on design time compliance checks described in Section 8.1.1, we have to provide mechanisms to answer the following queries for a given `PEG`:

- Does the `PEG` (and, hence, the corresponding process model / process instance) enable noncompliant traces and if so, which traces are affected?
- Will a CRG always be violated or do violations only occur in specific cases[2]?
- Which activities are involved in a violation and what are the possible causes?

As described in Section 7.1.2, for checking compliance of an execution trace with a CRG, we execute the CRG over the events of the trace. Each observed event may lead to a change of the effective compliance state w.r.t. the CRG. Recall that a compliance state is represented by a set of MarkStructures. Then, the CRG is enforced in the execution trace if the compliance state

[2]This, for example, facilitates to assess the risk and the extent of a violation.

reached after completing CRG execution is not associated with any VIOLATED MarkStructures. Following this, we execute the CRG along the paths of the PEG for checking compliance of a PEG with the CRG. If a compliance state associated with VIOLATED MarkStructures is reachable, the PEG contains noncompliant traces.

Instead of checking the PEG trace by trace for compliance (naive approach), we employ a more efficient strategy that annotates the nodes of the PEG with reachable compliance states. The basic idea is illustrated in Fig. 8.2. The CRG to be checked is executed along the paths of the PEG starting with the start node. In this procedure, each node of the PEG is annotated with the compliance states (represented by a set of MarkStructures) that are yielded on the paths to this node (including the node). Assuming the transitions of compliance states as given in Fig. 8.2, PEG node s_7 is, for example, annotated with the compliance states CS_1 and CS_3. CS_1 results from the lower path of the PEG while CS_3 is yielded on the upper path (note that CS_2 is transformed into CS_3 when observing e_7).

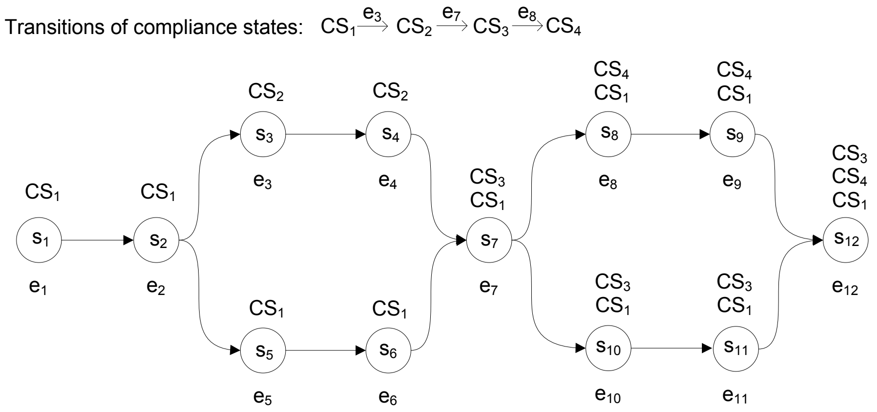


Figure 8.2.: A PEG annotated with reachable compliance states

By propagating reachable compliance states through the PEG[3], the compliance states associated with the end nodes of the PEG will reveal whether the CRG to be checked is enforced. Apparently, if an end node is associated with a compliance state consisting of at least one VIOLATED MarkStructure, the CRG is not enforced in the PEG. When being enriched with further information (e.g., transitions of compliance states), a thus marked PEG can also serve as basis for further analyses to localize the detected noncompliance. For example, assuming that CS_4 is associated with VIOLATED MarkStructures and attests noncompliance, tracing the noncompliance reveals that $< e_1, e_2, e_3, e_4, e_7, e_8, e_9 >$ is the execution trace leading to CS_4. The trace can be replayed in a process model or the process instance for visualization[4]. In order to generate explanations for detected violations and to identify the activities involved, the VIOLATED (event-specific) MarkStructures can be analyzed as described in Section 7.4.2.

[3]Note that is is also possible to build a product automaton of the model and the compliance states in a similar manner to explicit LTL model checking [CGP99, BBF$^+$01]. This basically corresponds to unfolding the PEG along the set of reachable compliance states and can result in a large structure. However, this approach can be advisable when a violation is detected and a detailed model unfolded w.r.t. the reachable compliance states for further analysis is required.

[4]In Section 9.3, we will address the visualization of compliance violations.

In the following, we first describe the general approach propagate reachable compliance states through a `PEG` in Section 8.2.1. Based on that, Section 8.2.2 addresses the question how noncompliance can be detected and further analyses, for example to localize detected compliance violations, can be conducted.

8.2.1. Propagating compliance states through a `PEG`

In order to detect noncompliance, we annotate the nodes of a `PEG` with all compliance states that can be reached when exploring the process from the start to the respective nodes. Def. 8.1 provides the fundament for that by introducing a function to execute a `MarkStructure` over a given START, END, or EX event. It is based on Algorithm 2, Algorithm 3, Algorithm 5, and MR9 (cf. Sections 7.3.2, 7.3.3, A.2.2, and 7.3.4, respectively). This function will be utilized for incrementally updating compliance states.

Definition 8.1 (Executing a `MarkStructure` over an event (*execute*))
Let $R = (A, C)$ be a CRG and $ms \in MS^*_R$ be a `MarkStructure` of R. Then,

$execute_R : MS^*_R \times E^* \rightarrow 2^{MS^*_R}$

is a function assigning child `MarkStructure`s to a given `MarkStructure` ms and event e with:

$$execute_R(ms, e) := \begin{cases} executeStart_R(ms, e)\ if\ e = (\text{START}, \ldots, \ldots, \ldots) \\ executeEnd_R(ms, e)\ if\ e = (\text{END}, \ldots, \ldots, \ldots) \\ executeAE_R(ms, e)\ if\ e = (\text{EX}, \ldots, \ldots, \ldots) \\ markEnd_R(ms)\ if\ e = \texttt{EOT}. \end{cases}$$

In the following, we provide marking algorithms for acyclic and cyclic `PEG`s. It should be noted that they can be applied using both *event-specific* (cf. Section 7.4.3) and *event-independent* `MarkStructure`s in order to provide diagnoses at the suitable granularity level (cf. Section 7.4.3).

8.2.1.1. Dealing with acyclic `PEG`s

Generally, the behavior specified by an acyclic process model or process instance can be represented by an acyclic `PEG`. Listing 6 in Appendix A.3 describes an algorithm for marking an acyclic `PEG` with reachable compliance states. Starting with the start node, the compliance states are propagated through the `PEG` until the end nodes are reached. A `PEG` node that has been processed propagates its reachable compliance states to all its outgoing edges. A not yet processed `PEG` node, in turn, will be processed and marked if all its predecessors and, thus, its incoming edges are marked with respective sets of compliance states. Each `PEG` node is processed exactly one time, which makes Algorithm 6 linear in the size of the `PEG`. Algorithm 6 is applied in Example 8.1.

Example 8.1 (Marking an acyclic PEG with reachable compliance states):
Fig. 8.3 depicts CRG R_1 and event-independent MARKSTRUCTUREs of it. The latter are employed to represent compliance states when applying Algorithm 6 to the verification of the PEG shown in Fig. 8.4. For compactness, we used EX events in the PEG and applied the execution and marking rules for EX events (cf. Section 7.5.2).

After the application of Algorithm 6, s_{20}, the end node of the PEG, is associated with two different compliance states, CS_1 and CS_2 each represented by a set of MARKSTRUCTUREs obtained from CRG execution. CS_1 is associated with the VIOLATED MARKSTRUCTURE ms_3 and, therefore, attests the presence of a compliance violation in the trace(s) that yield CS_1. In contrast, CS_2 is associated with a SATISFIED MARKSTRUCTURE ms_5 indicating that R_1 becomes activated and is satisfied in trace(s) that yield CS_2. The other MARKSTRUCTUREs associated with CS_1 and CS_2 are DEACTIVATED and, hence, do not affect the compliance state.

When using event-specific MARKSTRUCTUREs, the compliance states will enable detailed insights into the three activations of R_1 in the process.

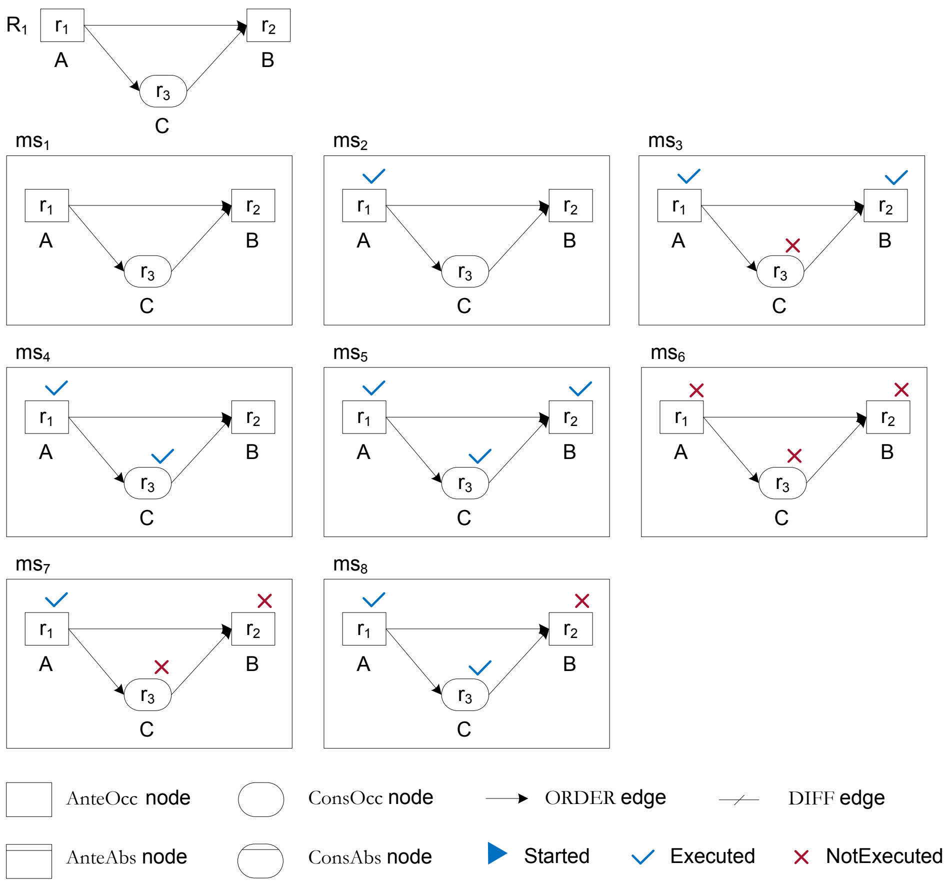


Figure 8.3.: CRG R_1 and corresponding MARKSTRUCTUREs

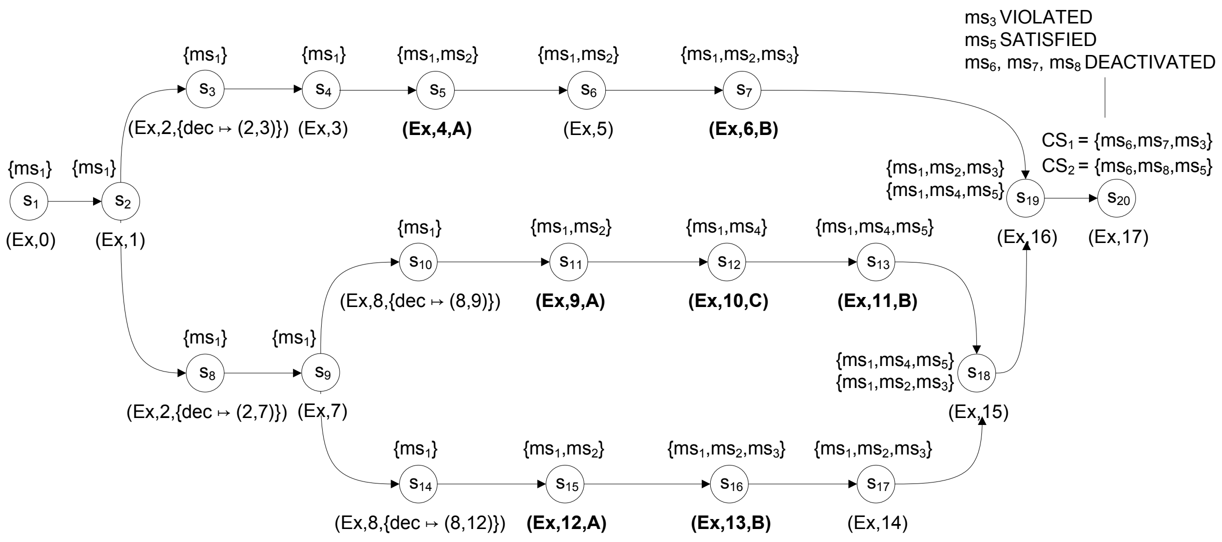

ms3 VIOLATED
ms5 SATISFIED
ms6, ms7, ms8 DEACTIVATED
CS1 = {ms6,ms7,ms3}
CS2 = {ms6,ms8,ms5}
s1 (Ex,0) {ms1}
s2 (Ex,1) {ms1}
s3 (Ex,2,{dec ↦ (2,3)}) {ms1}
s4 (Ex,3) {ms1}
s5 (Ex,4,A) {ms1,ms2}
s6 (Ex,5) {ms1,ms2}
s7 (Ex,6,B) {ms1,ms2,ms3}
s8 (Ex,2,{dec ↦ (2,7)}) {ms1}
s9 (Ex,7) {ms1}
s10 (Ex,8,{dec ↦ (8,9)}) {ms1}
s11 (Ex,9,A) {ms1,ms2}
s12 (Ex,10,C) {ms1,ms4}
s13 (Ex,11,B) {ms1,ms4,ms5}
s14 (Ex,8,{dec ↦ (8,12)}) {ms1}
s15 (Ex,12,A) {ms1,ms2}
s16 (Ex,13,B) {ms1,ms2,ms3}
s17 (Ex,14) {ms1,ms2,ms3}
s18 (Ex,15) {ms1,ms4,ms5} {ms1,ms2,ms3}
s19 (Ex,16) {ms1,ms2,ms3} {ms1,ms4,ms5}
s20 (Ex,17)

Figure 8.4.: Application of Algorithm 6 using event-independent MarkStructures

8.2.1.2. Dealing with cyclic PEGs

Cyclic PEGs become necessary in order to capture the behavior specified by cyclic process models and process instances. A new pass through a loop of a cyclic PEG may lead to new compliance states. To deal with this, a loop of the PEG is passed and reachable compliance states are propagated until no new compliance states become reachable. This means that a fixed point with regard to the compliance states of the corresponding CRG is reached.

Fixed point approach Listing 7 in Appendix A.3 constitutes a label correcting algorithm for marking PEGs. As such, starting with the annotations of the start node of the PEG and its outgoing edges, the reached compliance states are propagated through the PEG and the annotations of PEG nodes and edges are corrected until no further changes occur. If new compliance states are propagated to a PEG node s through one of its incoming edges, the CRG operational semantics is applied to these compliance states. If new compliance states are yielded by this operation, they will be propagated to successor nodes. Thus, CRG operational semantics is applied only to new compliance states. The maximum frequency a PEG node is processed by the algorithm, therefore, corresponds to the number of compliance states associated with that node.

Note that a fixed point will be reached ultimately, i.e., Algorithm 7 will always terminate, as the set of events associated with a PEG is finite[5] and so is the amount of MarkStructures of a CRG. Algorithm 7 can be applied using both event-specific as well as event-independent MarkStructures (cf. Section 7.4.3).

Example 8.2 (Marking a cyclic PEG):
Fig. 8.6 shows the application of Algorithm 7 to verify CRG R_2 from Fig. 8.5. The compliance states associated with PEG nodes obtained from the first / the second pass of the loop edge are

[5]Note that we assume finite domains for activity parameters (cf. Section 5.2.1).

shown in red / blue, respectively. We can see that the algorithm halts after the second pass of the loop as no new compliance states can be propagated to successor nodes of s_4 and s_6.

Each compliance state associated with the end node of the PEG is associated with an ACTIVATED MARKSTRUCTURE (i.e., ms_4, ms_6, and ms_7). This indicates that R_2 becomes activated in all paths of the PEG. Moreover, we can tell that the process enables both executions satisfying and executions violating R_2 as CS_1 reveals the satisfaction while CS_2 and CS_3 reveal the violation of R_2.

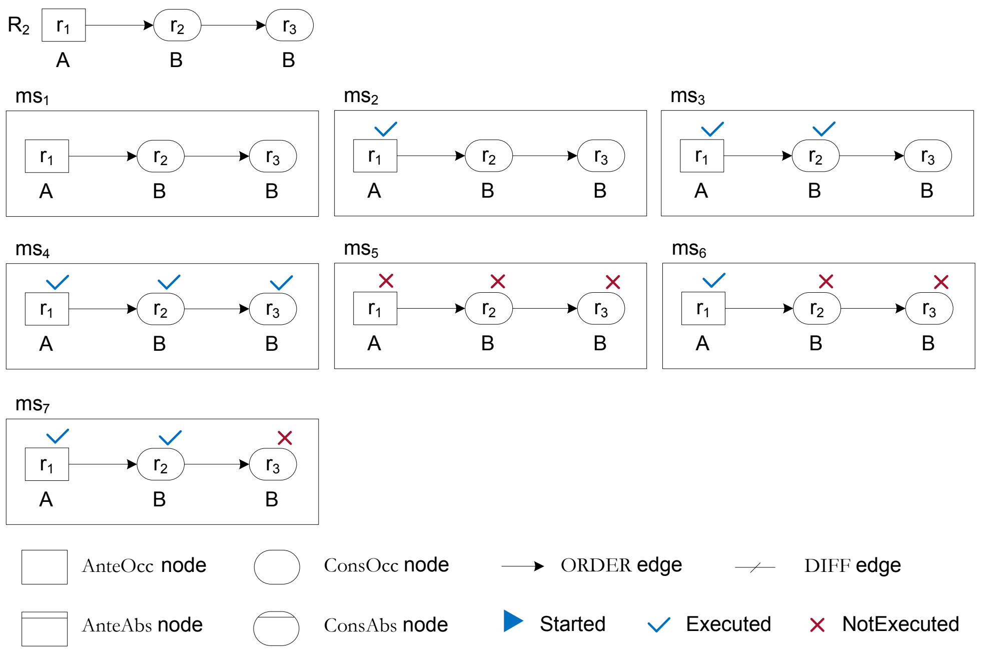


Figure 8.5.: A CRG and associated MARKSTRUCTURES

Bounded approach Process models and process instances containing cycles can also be verified by unrolling the loops up to a certain upper bound. Consequently, the resulting PEG is free of cycles. This enables verifying the process for a predefined number of loop iterations. This can be helpful to process designers to test designed process models, particularly when the interesting number of iterations is known.

8.2.2. Evaluation of marked process event graphs

Based on a PEG annotated with reachable compliance states and their transitions, we will be able to derive answers to the questions for comprehensive compliance diagnoses mentioned earlier in Section 8.2. In the following, Section 8.2.2.1 describes how to identify compliance violations

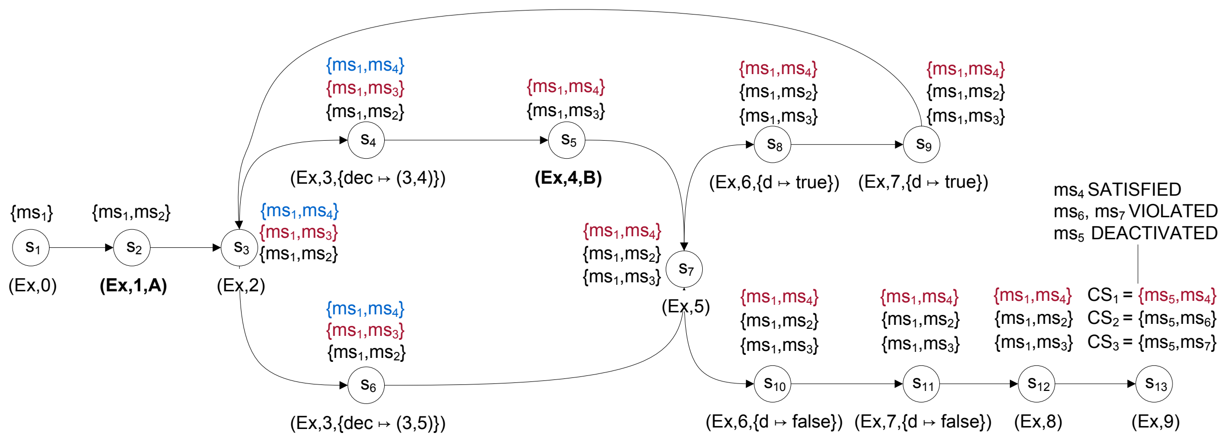


Figure 8.6.: Application of Algorithm 7 using event-independent MarkStructures

based on annotated PEGs. Section 8.2.2.2 addresses the identification of traces that bear noncompliance and the explanation of compliance violations.

8.2.2.1. Identification of compliance violations

As intuitively applied in the Examples 8.1 and 8.2, a CRG is not enforced in a PEG if a compliance state associated with an end node of the PEG reveals at least one VIOLATED MarkStructure. Such VIOLATED MarkStructures reflect predicted violations of the process. We will refer to the compliance states associated with the end nodes of the PEG as *final compliance states* in the following. As final compliance states together reflect the behavior of the traces of the PEG w.r.t. the imposed CRG, we can draw further conclusions from them. If all traces of the PEG contain noncompliance, all final compliance states will be bearing VIOLATED MarkStructures. If the PEG, however, enables both compliant and noncompliant traces, there will be both final compliance states with and without VIOLATED MarkStructures. Def. 8.2 formalizes this intuition[6].

Definition 8.2 (Compliance of a PEG with a CRG)
Let R be a CRG and let further $X = (S, s_0, S_E, T, el)$ be a PEG. Let further CS^*_R be the set of possible compliance states of R. As a compliance state is represented by a set of MarkStructures, $CS^*_R := \mathcal{P}(MS^*_R)$. Let further $pr_{X,R} : S \rightarrow 2^{CS^*_R}$ be the function assigning a set of reachable compliance states to each node of X. Then, we say

- R is ENFORCED over X iff holds:
  $\neg(\exists s_e \in S_E$ with $\exists CS \in pr_{X,R}(s_e)$ with CS consists of at least one VIOLATED MarkStructure$)$,

[6]Note that we assume that the end marking rule MR9 is applied when a PEG end node is reached (cf. Section 7.3.1.8). Thus, each assigned ACTIVATED MarkStructure will be either SATISFIED or VIOLATED, which enables the application of Def. 8.2.

- R is IGNORED over X iff holds:
  $\neg(\exists s_e \in S_E$ with $\exists CS \in pr_{X,R}(s_e)$ with CS consists of no VIOLATED MarkStructures), and
- R is ENFORCEABLE over X iff holds:
  R is neither ENFORCED nor IGNORED over X.

Example 8.3 (Identification of compliance violations within a PEG):
Consider again the marked PEG depicted in Fig. 8.6 based on the MarkStructures shown in Fig. 8.5. Then, we can tell from the three final compliance states associated with s_{13} that CRG R_2 is neither ENFORCED nor IGNORED but is ENFORCEABLE in the PEG.

It should be noted that Def. 8.2 is applicable regardless of whether event-independent or event-specific MarkStructures are employed. Depending on the choice of MarkStructures, the marked PEG can provide insights into the compliance situation at different levels of granularity. The compliance states associated with PEG end nodes further reveal whether a CRG always becomes activated in the process. This is the case when all final compliance states are associated with an activation of the CRG. In a similar manner, it is possible to detect whether a CRG will become violated each time it is activated. Altogether, a marked PEG not only enables the detection of compliance violations but also provides the basis for more advanced compliance diagnosis.

8.2.2.2. Tracing compliance violations

As mentioned earlier in Section 8.2, it can become necessary not only to detect that a CRG is not enforced but also to identify in which cases (i.e., in which traces) a violation occurs in order to better assess risks and apply adequate remedies. This corresponds to querying a marked PEG for paths that lead to final compliance states bearing VIOLATED MarkStructures.

By making use of information on the transitions of compliance states or by unfolding a PEG along the reachable compliance states (cf. Section 8.2.1), it is possible to track execution traces that resulted in a certain compliance state (e.g., compliance states revealing violations). These execution traces, in turn, can be visualized in the PEG. Based on the events and particularly the process node identifiers assigned to each PEG node, these traces can also be replayed a process model or a process instance. The visualization of compliance violations in process models will be addressed in Section 9.3.

Example 8.4 (Tracing MarkStructures):
Consider again the PEG depicted in Fig. 8.4. Then, by tracking CS_1, which is associated with a VIOLATED MarkStructure, we can identify the topmost and the lowermost path of the PEG as the ones bearing noncompliance.

8.2.2.3. Compliance diagnosis

In order to provide explanations for detected compliance violations, the MarkStructures associated with final compliance states of a PEG can be interpreted as described in Section 7.4.2. In particular, for a VIOLATED MarkStructure, we can derive potential causes of the violation based on the CRG node semantics and their execution states as shown in Example 8.5.

Example 8.5 (Identification of violations using event-independent MarkStructures): Consider again the PEG depicted in Fig. 8.6 annotated with compliance states of R_2 based on the MarkStructures shown in Fig. 8.5. The final compliance states CS_2 and CS_3 associated with s_{13} both bear compliance violations as indicated by the VIOLATED MarkStructures ms_6 and ms_7, respectively. Based on the considerations on interpreting MarkStructures described in Section 7.4.2, ms_6 reflects the case of no execution while ms_7 reflects the case of only one execution of B after A. As two executions of B after the occurrence of A are requested by CRG R_2, both these MarkStructures represent violation patterns of R_2. From that we can conclude that the PEG enables noncompliance in two possible ways.

Clearly, event-specific MarkStructures enable even more detailed diagnoses revealing the events associated with a MarkStructure (cf. Section 7.4.3). Using them, the events causing an activation of a CRG can be detected. This is helpful to pinpoint compliance violations, which is the first step to apply suitable remedies. A rule activation can occur in multiple traces and can be violated in one trace, but be satisfied in other traces at the same time. Such behavior can be detected by grouping MarkStructures of final compliance states whose antecedent pattern is associated with the same events. Altogether, the combination of showing the execution traces bearing noncompliance and deriving explanations from the obtained MarkStructures provides the fundament for comprehensive and meaningful compliance reports.

8.3. Compliance checking of process models

As described in Section 8.1.1, a process model can be verified against imposed CRGs by checking an equivalent PEG against these CRGs (cf. Fig. 8.1). The fundamentals for doing this were provided in Section 8.2. Based on these, Section 8.3.1 defines compliance notions for process models. Section 8.3.2 discusses how detected violations can be dealt with at the process model level. Finally, Section 8.3.3 discusses further issues and contrasts our approach against existing work.

8.3.1. Detection of compliance violations

As depicted in Fig. 8.1, compliance violations within a process model can be detected by checking an equivalent PEG capturing the behavior of the model for compliance[7]. Using PEGs consisting of

[7]Instead of transforming a process model into a PEG to conduct compliance checking, compliance can also be checked on-the-fly while exploring the process model.

execution traces to represent the behavior encoded by process models, our compliance checking framework remains independent from the specific process description language employed[8]. Note that in practice, a process model can be subject to multiple CRGs. We assume that these CRGs are compatible (i.e., their conjunction is satisfiable, cf. Section 6.3.3). In case multiple CRGs are imposed on a process model, each such CRG can still be verified individually.

Before conducting compliance checks, the original process model can be reduced to those parts that are relevant to the CRG to be verified to increase compliance checking efficiency. For now, we assume that to a process model, an equivalent PEG (cf. Def. 5.7) is provided. In Chapter 9, we will show how to automatically derive PEGs from WSM net process models [Rin04, Rei00] and will discuss abstraction strategies for more efficient compliance checking.

By verifying an equivalent PEG against a CRG, we are able to identify whether the corresponding process model enables noncompliant process executions. Clearly, a CRG is enforced over a process model if and only if it is also enforced over the equivalent PEG. This is formalized in Def. 8.3. In case a CRG R is not ENFORCED in a process model P, we will say that R is violated in P or P bears a violation of R.

Definition 8.3 (Compliance of a process model with a CRG)
Let P be a process model, R be a CRG, and X be a PEG equivalent to P. Then,

- R is ENFORCED over $P \leftrightarrow R$ is ENFORCED over X,
- R is ENFORCEABLE over $P \leftrightarrow R$ is ENFORCEABLE over X, and
- R is IGNORED over $P \leftrightarrow R$ is IGNORED over X.

Example 8.6 (Compliance checking of process models):
Recall Example 8.1. Then, Fig. 8.7 depicts process model P_1, which is equivalent to the PEG depicted in Fig. 8.4 with respect to execution traces consisting of EX events (i.e., P_1 and the PEG describe the same traces consisting of EX events). Hence, P_1 can be verified against R_1 from Fig. 8.4 by checking compliance of the PEG with R_1. As shown in Example 8.1, R_1 is not ENFORCED in the PEG. Therefore, R_1 is also not ENFORCED in P_1. As the end node of the PEG is not only associated with compliance states bearing violations, R_1 is ENFORCEABLE in P_2 (i.e., CS_2 is not associated with a compliance violation). Note that as P_1 neither involves data flows nor has correlations between split nodes, the graph structure of P_1 can in fact serve as an equivalent PEG. Thus, we can directly annotate P_1 with reachable compliance states.

Fig. 8.7 further depicts process model P_2. Example 8.2 showed how the PEG equivalent to P_2 w.r.t. traces consisting of EX events (cf. Fig. 8.6) is checked for compliance with R_2 depicted in Fig. 8.5. As shown, R_2 is not ENFORCED in P_2. However, as the end node of the PEG is also associated with a compliance state not bearing violations, R_2 is ENFORCEABLE in P_2.

[8]Interestingly, for certain process models, the underlying process graph already corresponds very much to a PEG that would result from the transformation. This is true for process models without data flows and without correlations between split nodes. Such process models can be directly marked with reachable compliance states.

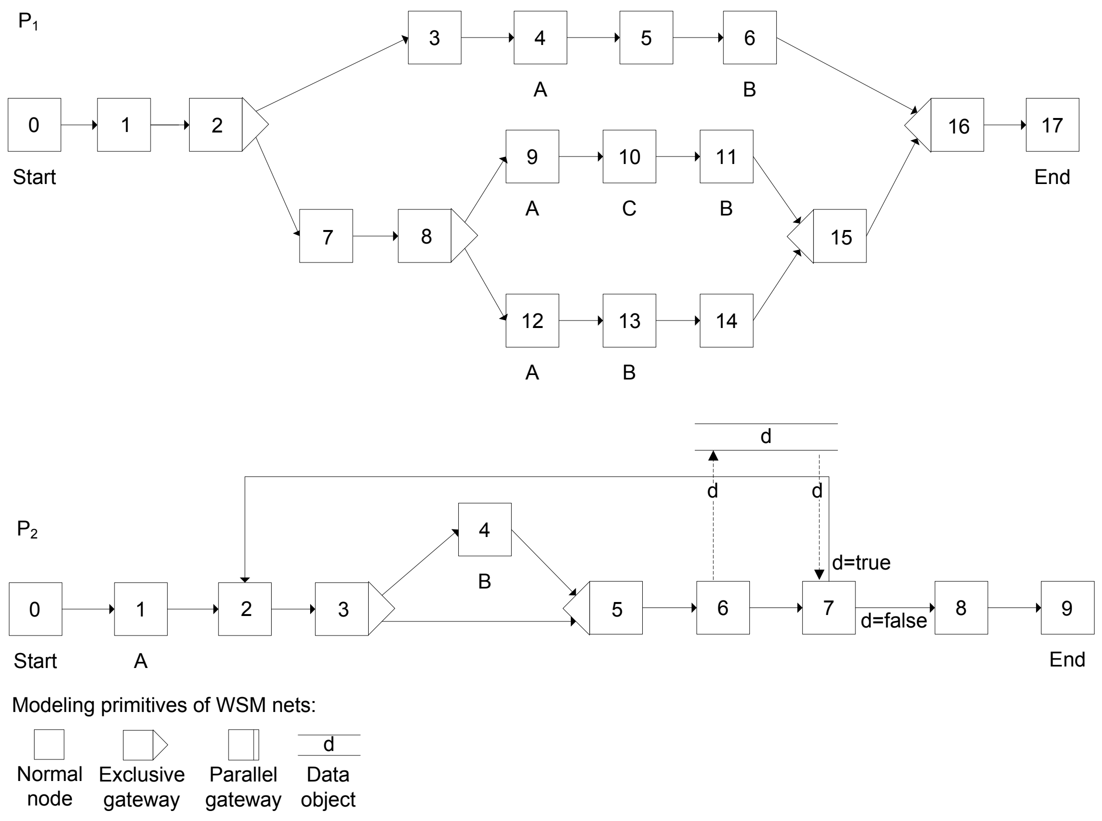


Figure 8.7.: Process models

8.3.2. Dealing with compliance violations at process model level

Detected compliance violations within process models can be dealt with in different ways:

i) A process designer may modify the process model such that not yet ENFORCED CRGs become ENFORCED.

ii) It is further possible to override the CRG for the particular process model.

iii) It is also possible to deploy a process model still containing violations and to monitor compliance with the CRG at runtime for the created process instances.

Modifying the process model In order to enforce the compliance with an imposed CRG at the process model level, the violations in the process model have to be resolved. For that purpose, the VIOLATED MARKSTRUCTUREs obtained from compliance checking can be exploited for root-cause analysis and to derive helpful information on how to resolve the violations. In this context, event-specific MARKSTRUCTUREs are particularly interesting as they shed light on the particular activations of a CRG. For a VIOLATED (event-specific) MARKSTRUCTURE, its VIOLATED CONSEXMARKs can provide explanations why the MARKSTRUCTURE became VIOLATED in the first place (cf. Section 7.4.2).

Overriding the CRG Depending on the enforcement level of the CRG (cf. Appendix A.4), the process designer may decide to override a not ENFORCED CRG. Overriding may be effected at different levels. Generally, the overall CRG may be overridden causing the CRG to be no longer imposed on the particular process model. Alternatively, the process designer may also override selected activations of the CRG that are not ENFORCED in the process model. A use case for overriding selected activations is given when CRGs are defined using abstract activity types (e.g., from a domain model). Then, multiple activations of the CRG can be contained in a process model whose enforcement level may even vary depending on the particular activity types involved.

Monitoring compliance at runtime This option becomes particularly interesting for CRGs that are rather of recommendation nature (i.e., low enforcement level). Then, it may not be desirable to enforce the CRG already at the process model level. In this case, monitoring the compliance with the CRG for process instances created from the process model enables more flexibility as the CRG can be dealt with at a process instance basis. In addition, monitoring compliance at runtime can also become advisable if the modifications necessary to enforce the CRG would yield an overly complex process model.

8.3.3. Discussion

In Section 8.3, we showed how compliance checking of process models can be realized using CRGs and the corresponding execution mechanisms. In the following, we discuss alternative approaches to realize process model verification in Section 8.3.3.1. Finally, further issues beyond the scope of this thesis are outlined in Section 8.3.3.2.

8.3.3.1. Related work

Our framework verifies a process model by applying CRG operational semantics to the process model's state space representation (i.e., to the execution traces encoded by the process model's PEG). By doing so, we identify which compliance states of an imposed CRG (expressed as sets of MarkStructure) a state of the process model is associated with. Depending on the desired granularity level, the compliance state may be represented through event-independent or through event-specific MarkStructures. In addition, alternative approaches to assign compliance states to PEG nodes are available, for example, with or without unfolding the PEG.

Some model checking approaches employ a similar strategy to verify a system specification against a property. The basic idea of explicit LTL model checking is to produce the product automaton of an automaton representing the system's behavior (i.e., the PEG in our case) and an automaton representing the negated property [BBF+01]. As discussed in Section 3.1, a variety of approaches in literature suggests model checking to verify compliance, for example [FPR06, YMHJ06, LMX07, FESS06, FESS07, FS10, KGE11]. In the SeaFlows project, we also experimented with model checking. This resulted in the implementation of a compliance checking component of the SeaFlows Toolset [LKRM+10, KLRM+10] (cf. Chapter 11). For this approach, we used the model checker SAL. One major reason we also investigated on alternatives to model checking was that a LTL model checker typically provides a single counterexample in the form of an execution trace in case a violation is detected. In addition, model checking provides no inherent support for identifying and verifying single rule activations. Due to the parallels, our approach to check compliance of a PEG with an imposed CRG can be considered a model checking approach that aims at enabling comprehensive and fine-grained compliance diagnoses.

In the SeaFlows project, we also investigated on analyzing the graph structure of the process model to efficiently identify noncompliant structures [LKRM$^+$10]. The developed approach exploits the block structure of WSM net process models to analyze the process graphs. However, the approach is limited to simple compliance rules without data conditions. In case of data-aware compliance rules and process models containing data flows, analysis of the mere graph structure of the process model is no longer sufficient in order to correctly identify noncompliance.

As described in Section 2.1.2.1, being able to provide explanations for compliance violations is an important requirement for the practical application of a compliance checking framework. A few approaches in literature address this issue in the context of design time compliance verification [ETHP10b, AWW09]. In [ETHP10b], Elgammal et al. present an approach for the property specification patterns introduced by Dwyer et al [DAC99] (cf. Section 2.1.1.1) that is based on providing a so-called *current reality tree* for each pattern. The roots of the tree are associated with undesirable effects, i.e., violations of a certain pattern. By traversing the tree answering questions associated with the violation, the user is guided to the root-cause of the violation and is further provided a remedy strategy. In [AWW09], Awad et al. propose an approach for visualizing and explaining compliance violations that utilizes *temporal logic querying* to identify paths in which a (data-aware) compliance rule is violated. Similar to the work by Elgammal et al., Awad et al. also focus on specific compliance rule patterns that are based on the property specification patterns. As shown in Section 8.2.2, our approach enables not only the explanations for compliance violations of arbitrary CRGs but also the identification of traces violating CRGs. In Section 9.3.1, we will show how this information is seized for visualizing compliance violations.

8.3.3.2. Further issues

In this section, we assume that a PEG equivalent to a process model to be checked with regard to compliance with the imposed CRG is provided in order to conduct compliance checking. Being based on PEGs as general process representation, our framework is independent of particular process description languages. In Chapter 9, an approach to transform WSM Net process models into PEGs is provided. As already discussed in Section 8.3, the transformation of a process model into a PEG *before* conducting compliance is not necessary. In fact, the CRG operational semantics can also be applied on-the-fly while exploring the possible process executions encoded by the process model. By doing so, a state space representation of the process model, for example, a reachability graph for Petri net process models, marked with reachable MarkStructures can be derived.

As compliance checking of process models requires the exploration of the process executions enabled by the process model, state explosion can become a challenge. As CRGs are typically small in practice, the complexity of the compliance checks is rather driven by the size of the model. Thus, reduction of the latter constitutes a valid strategy to tackle the state explosion problem [Awa07, KLRM$^+$10, RWMR13]. To achieve this, abstraction strategies also applied for model checking can be employed. Structural abstraction, for example, can be applied to reduce the process model to those parts relevant to the imposed CRG. Here, existing approaches, such as [Awa07], can be applied. As typically only a relatively small part of a process model is affected by a compliance rule, such structural abstraction can help to substantially reduce the complexity of compliance checks. Data domain abstraction can further be used to prevent the state explosion problem caused by data exploration. In [KLRM$^+$10], we propose an abstract interpretation approach to tackle this. In Chapter 9, we will describe abstraction strategies in more detail.

Our framework enables the identification of violations and even to pinpoint violations in a PEG. This can serve as input to generate feedback for users, for example visualization of the noncompliant traces in the process model. Though very interesting, concepts to generate suitable compliance reports and to visualize compliance violations to process designers and process users are beyond the scope of this thesis. Results obtained from a first study on this are described in Section 9.3.

The application of suitable remedies to resolve noncompliance is considered a manual task so far. Especially when multiple CRGs affecting the same activities are imposed on a process, manual modification of the process such that all CRGs are satisfied can become challenging. Thus, the (semi)automatic derivation of suitable remedies, for example, process adaptations, constitutes an interesting research challenge. In this context, the few existing approaches such as [GK07] should be investigated in future work.

8.4. Compliance monitoring during process execution

The objective of compliance monitoring is to detect noncompliance in a prompt manner. In the best case, noncompliance can even be prevented when being detected in time. Otherwise, detected compliance violations can be documented, which facilitates a posteriori analysis.

Recall the requirements on the compliance checking framework described in Section 2.1.2. The two essential aspects of compliance monitoring are illustrated in Fig. 8.8. Essentially, it must be possible to check whether a running process instance violates an imposed CRG. Obviously, compliance violations can be detected by checking the execution trace of the process instance. Monitoring this can detect emerging violations in a timely manner. We refer to this as compliance state monitoring (cf. Fig. 8.8). Section 8.4.1 addresses this in more detail.

Even when process instances are executed based on a known process model that has already been verified against imposed rules, compliance monitoring for running process instances can still become necessary. As mentioned in Section 8.3.2, it is not always viable to enforce compliance with all imposed CRGs at the process model level (for example, as this can lead to overly complex process models or due to the low enforcement level of the imposed CRG). In scenarios in which process instances are executed based on a known process model, the latter can be exploited in

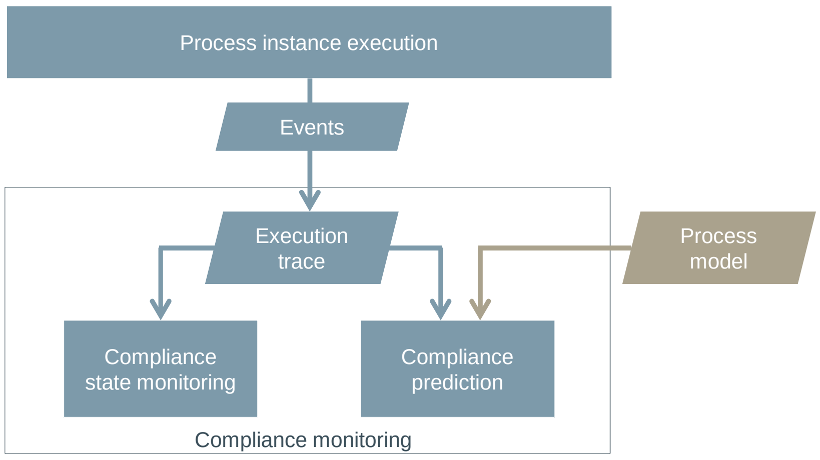


Figure 8.8.: Different aspects of compliance monitoring

order to update the compliance predictions made at the model level (cf. Fig. 8.8). How this can be realized is discussed in Section 8.4.2.

8.4.1. Execution trace based compliance monitoring

By verifying the execution trace of the process instance against an imposed CRG, violations occurred so far can be revealed. The compliance state yielded by the execution trace of the process instance is referred to as *effective compliance state* (ECS) of the process instance as formalized in Def. 8.4. For each new execution event observed during process execution, the CRG operationalization can be applied to update the ECS. As the particular procedure of verifying an execution trace against a CRG corresponds to the incremental application of CRG operational semantics and has already been exemplified earlier in Section 7.4, we abstain from detailing this in the following.

Definition 8.4 (Effective compliance state of a process instance)
Let R be a CRG, P be a process model, and $I = (P, \sigma)$ be a process instance of P. Then, the *effective compliance state* (ECS) of I with regard to R is represented through a set of MarkStructures MS_I with $MS_I := verify(R, \sigma)$ (cf. Algorithm 1).

If R is clear from the context, we will just say ECS of I.

If the ECS of a process instance reveals violations (represented by VIOLATED MarkStructures), these violations are already manifest in the execution history and, thus, permanent. Then, analysis of the respective MarkStructures can help to identify the root-cause of the violations. Moreover, detected violations can be reported to supervisors and documented by respective personnel (e.g., why noncompliance happened) in a timely manner. This can provide valuable information for process optimization [LRMGD12].

Before a violation becomes manifest in the trace, analysis of the VIOLABLE MarkStructures of the ECS enables the derivation of measures to render the respective CRG or selected activations of it satisfied (*proactive prevention of violations*). We described this in Section 7.4.2. This functionality can be exploited for providing support to comply. Further ideas on proactive prevention of noncompliance are described in [LRMKD11].

8.4.2. Compliance prediction at process instance level

Generally, compliance monitoring will only become necessary for process instances executed based on an already verified process model if the respective CRG is ENFORCEABLE in the underlying model. The reason is that ENFORCED/ IGNORED CRGs will become satisfied / violated in any instance of the model, respectively. It is further possible to monitor not the original CRG but only selected activations of it (for example, activations that are ENFORCEABLE) by refining the CRG accordingly. In addition, monitoring may also be conducted for only selected process instances, for example, process instances with priority.

Based on Def. 8.3 for process models, Def. 8.5 formalizes the notions ENFORCEABLE, ENFORCED, and IGNORED for process instances.

Definition 8.5 (Compliance of a process instance with a CRG)
Let $I = (P, \sigma)$ be a process instance and R be a CRG. Let further X be a PEG equivalent to I with regard to compliance with R. Then,

- R is ENFORCED over $I \leftrightarrow R$ is ENFORCED over X,
- R is ENFORCEABLE over $I \leftrightarrow R$ is ENFORCEABLE over X, and
- R is IGNORED over $I \leftrightarrow R$ is IGNORED over X.

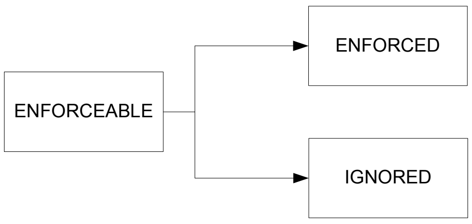


Figure 8.9.: Relations between the compliance notions ENFORCEABLE, ENFORCED, and IGNORED

As the execution of a process instance proceeds, a CRG that is ENFORCEABLE in a process instance can become ENFORCED or IGNORED depending on the activities executed in the instance as illustrated in Fig. 8.9. Once an ENFORCEABLE CRG becomes ENFORCED in the process instance according to Def. 8.5, it can no longer be violated (unless the process instance deviates from the predefined model). If an ENFORCEABLE CRG becomes IGNORED during the process instance execution, however, the respective CRG will be violated (unless a violation is already manifest in the process instance' ECS). Keeping such compliance predictions for a process instance current enables prompt reaction on changes as illustrated by Example 8.7.

Example 8.7 (Updating the compliance prediction for a process instance):
Fig. 8.10 depicts process instance I_1 running on process model P_1 from Fig. 8.7 at two different points in time. Recall the PEG equivalent to P_1 annotated with reachable compliance states of R_1 shown in Fig. 8.4. At time t_1, R_1 is still **ENFORCEABLE** in I_1 as R_1 can still become both violated or satisfied depending on which path is chosen after completion of process node 8. This is also reflected in the PEG from Fig 8.4. At time t_2, R_1 is **IGNORED** in I_1 as at this point, R_1 is going to be violated when the execution of I_1 proceeds. However, the predicted violation is not yet manifest in the ECS of I_1 and, therefore, can still be averted, for example, by modifying the process instance accordingly.

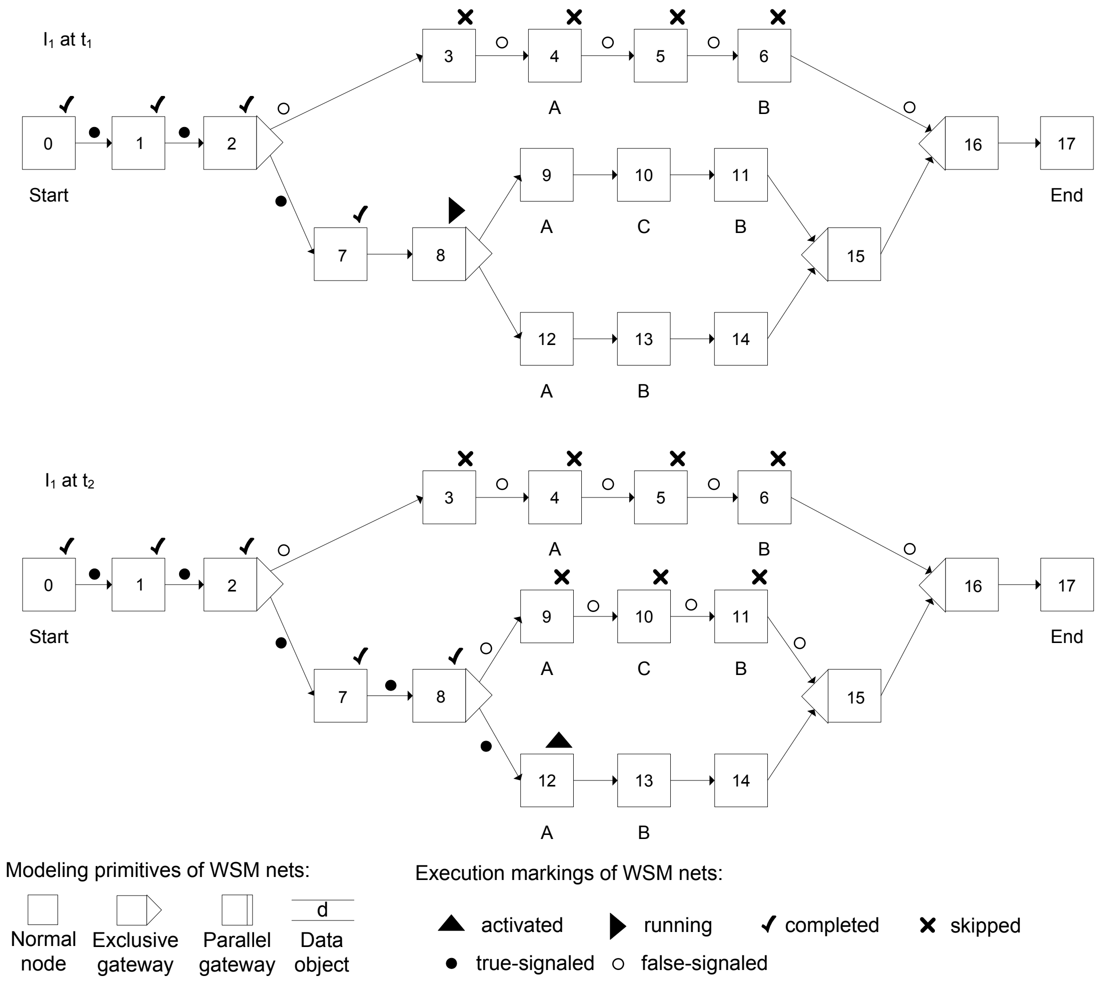


Figure 8.10.: A process instance at two different points in time

How to update the compliance prediction for a process instance? Generally, the compliance prediction for a process instance can be derived the same way a process model is verified against imposed rules (cf. Section 8.3). In particular, the process instance can be transformed into a `PEG` for compliance checking or the exploration of the process instance and the CRG execution can be combined in one step. When a `PEG` of the instance' process model is already given, we can use this `PEG` for compliance checks as the process model `PEG` captures the superset of traces that can be produced from a running process instance I. Thus, by replaying I's execution history in the process model's `PEG`, we can precisely identify the future execution trace suffixes that still can be generated from I in the future. These are exactly the trace suffixes reachable from the `PEG` node reached when replaying I's execution history. Based on this, the propagation approaches (cf. Section 8.2.1) can be applied to find out whether the process instance enables noncompliant execution traces.

When to update the compliance prediction? The compliance prediction may be updated upon reaching predefined check points in the process model or the update may be automated. Obviously, not all events occurring during process execution necessitate updating the compliance prediction as some events do not affect the possible future behavior. Independently of the imposed CRG, an `ENFORCEABLE` CRG can only become `IGNORED` or `ENFORCED` when an observed event leads to discard of execution trace suffixes in an equivalent `PEG`. The latter will only apply if the observed event corresponds to the selection of an outgoing branch of a split node in the `PEG`. Updating the compliance prediction only in such cases, therefore, helps to avoid unnecessary operations.

It should be noted that there are also other ways to keep the compliance prediction of a process instance current. In scenarios where emphasis is put on this, we may also unfold the `PEG` of the process model with regard to the reachable compliance states as discussed in Section 8.2 when verifying the process model. The unfolded `PEG` can then be used to keep track of still reachable trace suffixes and still reachable compliance states at runtime.

8.4.3. Discussion

In Section 8.4, we showed how compliance monitoring can be realized based on the proposed CRG approach. Our framework enables to monitor the effective compliance state of a process instance based on events observed during process execution. The possible future behavior encoded in a process instance' process model can be taken into account to detect changes of compliance predictions made at process model level. In Section 8.4.3.1, we discuss alternative approaches to compliance monitoring. In Section 8.4.3.2, we discuss further issues that are not addressed by our approach.

8.4.3.1. Related work

In literature, a variety of approaches for compliance monitoring were proposed. Most of these focus on data consistency or process performance (e.g., KPI monitoring or business activity monitoring). However, there are also approaches that address compliance monitoring based on observed process execution events. As discussed earlier in Section 7.7.1, we can distinguish between different types of such monitoring approaches. Some approaches use automatons to monitor process executions [MMWA11, PA06, Pes08, SFV$^+$12]. Typically, these automatons are generated from LTL properties. By producing an automaton representing the conjunction of the imposed rules, it becomes possible to detect whether conflicts between them arise during execution making it impossible to satisfy all imposed rules [MMWA11]. This is not yet addressed by our approach. In Section 8.5.3, we describe ideas how this can be realized. In our approach, the automaton is yielded by the possible compliance states of a CRG to be checked and the transitions between them. Instead of explicitly creating an automaton, the reachable compliance states are unfolded incrementally during monitoring. Other approaches monitor compliance by querying the execution trace for patterns of compliance violations or of relevant incidents [GMP06, HMZD11, BDL$^+$10]. This can be realized using event processing technologies such as *complex event processing* (CEP) [JML09]. Further alternatives involve formalizing compliance rules for example using the event calculus [MMC$^+$11] or using logic programming [ACG$^+$08, MPA$^+$10]. For details on the techniques applied, we refer to the discussion in Section 7.7.1. The mentioned approaches all focus on execution trace based or event based monitoring in general. To our best knowledge, the compliance monitoring approaches in literature do not exploit the process model in order to update compliance predictions for process instances.

Being based on CRG execution, our compliance checking approach is applicable to both design and runtime compliance checking. The beauty of our approach is that a transformation of the CRGs into other representations such as automatons or alternative formalizations is not necessary in order to conduct monitoring. Incremental CRG execution enables dealing with evolving execution traces. Even after a violation has occurred, monitoring can still be continued. The latter can be a problem for approaches relating compliance violations to a notion of logical inconsistency. As the graph structure of the CRGs is exploited for compliance monitoring, feedback can be given based on the specific rule structure. Since compliance states are represented by MarkStructures, compliance states are transparent and can be analyzed, for example, in order to derive strategies to prevent compliance violations [LRMKD11]. In addition, similar to compliance checking at design time, compliance can be monitored using both event-specific and event-independent MarkStructures. This gives the process supervisor the flexibility to decide on the suitable granularity for the respective application scenario.

Conformance checking investigates whether a process model and process logs are conform to each other and is, hence, related to compliance monitoring. Generally, the conformance can be tested, for example, by replaying the log over the process model. To tackle this, several approaches were proposed [AM05, RA08, WZM$^+$11, WPDM10, WPD$^+$11] that introduce techniques and notions, such as fitness and appropriateness, to also quantify conformance. Implementations of these approaches (e.g., the Conformance Checker) are available in the process mining framework ProM [VBDA10]. Conformance checking and compliance rule monitoring exhibit major differences that require different techniques. For example, compliance rules are typically declarative while process models are mostly procedural.

8.4.3.2. Further issues

In this work, we assume that execution events are observed after the fact. Thus, violations are revealed in the execution trace after they happened. However, one can also think of scenarios in which the compliance monitoring tier is queried by the process execution tier to find out whether the commit of an event leads to compliance violations. This enables declining respective events a priori.

8.5. Summary and further issues

As pointed out previously in Chapter 3, the bulk of existing work focuses on isolated scenarios of the business process lifecycle rather than on providing integrated support for process design and process runtime. The combination of these isolated approaches still lacks support with respect to comprehensiveness and granularity of compliance diagnoses. Thus, there is still need for a compliance checking framework that not only supports the verification of process models and process instances but is also able to provide intelligible compliance reports at the level of particular rule activations. In this chapter, we showed how a compliance checking framework can be realized based on the CRG concepts introduced in Chapter 6 and 7. As envisioned (cf. Section 2.2.1), the described compliance framework addresses both design and runtime compliance verification using the same mechanisms. Thus, transformations of compliance rules modeled as CRGs into other languages or representations that target design or runtime compliance checks are not necessary.

Aiming at enabling detailed compliance diagnoses, we exploit the structure of CRGs to represent reachable compliance states. As the latter can be represented through both event-specific as well as event-independent MarkStructures, the granularity of compliance checks can be adjusted to suit the needs of the respective stakeholders and application scenarios.

A process model is verified against imposed CRGs by determining the compliance states that its executions are able to yield. While this can be conducted in one step, we showed the verification of process models in two steps in this chapter for better illustration. First, the process model is explored to derive a specification capturing the behavior encoded in the model. In this work, we utilize PEGs, an automaton-like structure whose nodes are associated with execution events, to represent the behavior of processes independently from the process description language employed. The PEG is then verified by marking its nodes with reachable compliance states. The detected compliance states not only reveal compliance violations enabled by a process model but also enable the derivation of meaningful compliance diagnoses. For example, it can be detected in which traces a violation occurs and whether a CRG becomes activated or is violated in all or only some specific executions of the process. This information and the MarkStructures bearing compliance violations can be utilized for explaining and visualizing detected noncompliance within the process model to be checked. Some ideas on the visualization of compliance violations will be described in Chapter 9.

The compliance state of a running process instance is monitored based on its evolving execution trace. The incremental nature of CRG operational semantics enables to derive the effective compliance state after observing a new event based on the last compliance state. When a compliance violation becomes evident, explanations for its root-cause can be derived from the MarkStructures that constitute the respective compliance state. Even before a compliance violation becomes manifest, useful information can be derived from the compliance state that can, for example, be utilized to schedule activities in order to satisfy an imposed compliance rule [LRMKD11] (i.e., support to comply). Moreover, the possible future behavior of a process instance predefined in the underlying process model can be exploited for updating compliance predictions. Together with monitoring based on the execution trace of a process instance, this can help to identify potential compliance threats in a timely manner.

The compliance checking framework presented in this thesis is not restricted to a particular process description language as it is based on a general notion of execution traces. It can be complemented with further concepts such as abstraction to increase the efficiency of compliance checks as we will show in Chapter 9. As stated earlier, we can think of embedding it into an overall compliance management framework such as envisioned in the COMPAS project [The11]. The SeaFlows compliance checking framework would be complemented by existing functionalities for managing and tracing compliance requirements while the compliance management framework could benefit from the compliance checking functionality of our framework.

8.5.1. Process adaptations

During process execution, it can become necessary to deviate from the predefined process model, for example, in order to deal with an exceptional situation [EKR95, RD98, RWRW05, WSR09, KR11a]. To deal with such cases, flexible process management systems such as ADEPT [RD98, DRR+08] enable to change the process, for example, to add additional activities or remove existing ones. Changes can be applied to particular process instances or to the process model. Obviously, process changes can introduce noncompliance that can be detected by rechecking the process instances and process models against imposed CRGs. In order to avoid unnecessary checking operations, it is desirable to only reverify the process against those CRGs that are possibly affected by the process changes. In [LRD06, LRD08], we described a first approach to identify such rules that is based on exploiting the semantics of the change operation. This approach is limited to basic rule structures. This can be extended exploiting further information such as identified rule activations and their individual compliance state.

8.5.2. Declarative process modeling and execution

The declarative approach to process modeling and execution aims at enabling more flexible processes and, thus, at overcoming limitations that many traditional imperative workflow approaches suffer from [WLB03, PA06, Pes08, FLM+09]. The basic idea is to specify what should be done in a process without specifying how this should be accomplished. A declarative process model is, therefore, constituted by a set of constraints that each process execution has to comply with. After a process instance is started from such a constraint-based process model, an agent may execute activities without being guided by a predefined process. However, at the end of the execution, the executed process instance must satisfy all constraints.

Existing declarative approaches propose different constraint specification approaches. The Tucupi approach introduced by Wainer et al. [WLB03, WLBB04] uses constraints based on pre- and postconditions for executing particular activities. Pesic et al. introduce a declarative approach based on linear temporal logic (LTL) called DECLARE in [Pes08, PA06]. In DECLARE, the constraints are constituted by a set of LTL formulas. To hide the complexity of LTL from the user, Pesic et al. introduce a set of patterns each of which is assigned a graphical notation. To execute a declarative process instance, an automaton is created from the LTL formulas of the process model. By checking the activities initiated by users against the automaton, it can be determined whether the process instance is in an accepting state or whether the satisfaction of all constraints can still be achieved.

CRGs can be applied to design declarative process models. Then, a process model is constituted by a set of CRGs. When a process instance is created from the model, its state is represented by a set of MarkStructures. Similar to monitoring a process instance (cf. Section 8.4.1), the effective compliance state of a declarative process instance can be monitored using the CRG execution mechanisms. Thus, activities that render a MarkStructure `violated` can be declined by the system as such a violation is permanent. The process instance can only be terminated, if all `activated` MarkStructures are `satisfied`. From the `violable` MarkStructures, we can further derive information that can help users to complete the process instance (i.e., support to comply). For example, a `pending` MarkStructure indicates that activities are still required in order to comply with the respective CRG. Further research on the application of CRGs to support the declarative process paradigm seems to be interesting and promising.

8.5.3. Conflicts among rules

Clearly, multiple compliance rules imposed on a process can only be satisfied when they are not conflicting meaning that their conjunction is not satisfiable due to logical inconsistency (cf. Section 6.3.3). However, even with rules that are not conflicting in general, conflicts can still arise. Consider, for example, two compliance rules, one requesting that C is executed after A (rule R_1) and the other requesting that C must not be executed after B (rule R_2). Now imagine a process model defining that B will be executed prior to A. Then, despite not being conflicting in general, R_1 and R_2 are still conflicting in that particular case as it is not possible to satisfy both rules unless A or B are removed or arranged differently. The detection of such conflicts is not yet addressed by our work. However, several approaches in literature address this issue. Awad et al. propose an approach for checking the consistency of compliance rules (specified in LTL) based on Büchi automaton that can be enriched with information from the process domain in [AWW10]. Maggi et al. [MMWA11] utilize a global automaton involving all imposed rules (specified in LTL) to detect conflicts arising during process execution. Situations in which there is no possible continuation satisfying all rules can be detected. Further research becomes necessary in order to adapt these promising approaches for CRGs.

8.5.4. Rule refinement

As CRGs can be modeled using abstract activity types (e.g., from a domain model), there may be multiple activations of such an abstract CRG in a process that are associated with different concrete activities. Depending on the particular activities involved in the rule activation, different enforcement levels may become relevant. For example, certain rule activations may require more stringent enforcement than others or specific remedies. Against this background, it is notable that we can exploit the identified rule activations to refine the original CRG such that it applies only to a particular rule activation. This can be accomplished by allocating the node identifier property and data conditions of CRG nodes accordingly. Such event- or activity-specific CRGs can be used, for example, to perform checks for a particular previously identified rule activation or to monitor a particular rule activation at runtime.

9 The compliance checking process: pre- and post-processing

In order to verify a process model against imposed compliance rules, we have to explore the behavior encoded by the model. In Section 8.1.1, we showed process model verification in two steps. First, the process model is explored and a model specification capturing the possible process behavior is obtained. In this thesis, we utilize process event graphs (PEGs, cf. Sect. 5.2.6), an automaton-like structure whose nodes are associated with execution events, to describe the behavior enabled by a process model. The PEG is verified against imposed rules in the second step. In Chapter 8, we described how this is conducted using the operational semantics of CRGs. Clearly, this constitutes the central part of the compliance checking process as illustrated in Fig. 9.1. In this chapter, we address pre- and post-processing aspects of the overall compliance checking process to complete the picture. Specifically, this chapter addresses two questions:

- Pre-processing: How to obtain a model specification capturing the process behavior (like a PEG) for compliance checking of process models and process instances?
- Post-processing: How to visualize and explain the results of compliance checks to stakeholders?

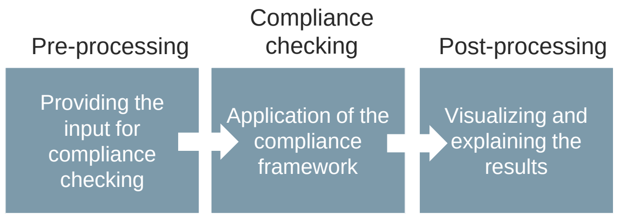

Pre-processing
Compliance checking
Post-processing
Providing the input for compliance checking
Application of the compliance framework
Visualizing and explaining the results

Figure 9.1.: Simplified compliance checking process

The first question is addressed in Section 9.1 for the example of WSM nets [Rei00, Rin04]. Section 9.2 then describes abstraction strategies that can be applied to yield more compact PEGs and, thus, can help to reduce the computational costs of compliance checks. The second question is addressed in Section 9.3 where we describe approaches on visualizing and explaining compliance checking results from literature and developed in the SeaFlows project.

9.1. Transforming process models into process event graphs

Generally, a model specification capturing the behavior encoded by a process model like a PEG can be obtained by simulating the process execution[1]. This can be accomplished by applying the operational semantics of the respective process description language and corresponds to a reachability analysis as known from Petri nets [May81]. The general procedure is to start with an initial process execution state and check which activities can be started or completed in that state. By expanding these enabled events, further execution states of the process can be explored until no further events are enabled. By doing so, we are able to build a PEG containing all event sequences that can be generated by the process. In the following, we outline a transformation algorithm for the example of WSM net process models and process instances.

9.1.1. Basic transformation algorithm

WSM nets [Rei00, Rin04] have well-defined operational semantics. When executing a WSM net process instance, execution markings are assigned to nodes and edges of the process and values are assigned to data objects (referred to as PROCESS MARKING in the following) that reflect the current execution state and also enable to retrace the execution path taken so far (cf. Sect. 5.1.2). Listing 4 sketches a general algorithm for transforming a WSM net process model into a PEG. Algorithm 4 can be applied to transform process instances into PEGs as well (only minor modifications become necessary).

- Starting with an initial state z_0 representing the PROCESS MARKING after starting the START node of the process model[2], it is first checked for events that can be fired under the present PROCESS MARKING. This is encapsulated in the function getNextEvents, which returns a set of events that can be fired under a particular PROCESS MARKING pm_z. It should be noted that multiple events may be enabled under a certain PROCESS MARKING (cf. Sect. 5.1.2).
- The firing is simulated for each such identified event. We assume that the WSM net operational semantics is applied for simulating the firing (e.g., start an activity), which results in a succeeding execution state for each fired event. This is encapsulated in the function fireEvent, which returns a tuple of a PROCESS MARKING $pm_{z'}$ and an event e.

[1]While the exploration of a process model and the compliance verification may also be combined in a single step, we opted for showing these two steps separately in Section 8.1.1 for clarity reasons. Thus, a process model is first transformed into a PEG based on which compliance checks are conducted. However, the considerations and concepts presented in this chapter are also applicable when exploring and verifying a process in one single step.

[2]Note that WSM nets have a single start and a single end node [Rei00, Rin04].

- It is then checked whether the obtained state encoded through the `PROCESS MARKING` and the event has already been reached previously. If this is not the case, the novel state will be put into a queue of states to be further explored. In addition, the precedence relation between the original and the novel state will be established. Elsewise, only the novel precedence relation is established by adding a corresponding tuple to E^*.

- From the structure Z^* and E^* obtained from Algorithm 4, we can easily derive a `PEG` $X = (S, s_0, S_E, T, el)$ (cf. Def. 5.6): For each $z \in Z^*$, we create a `PEG` node $s \in S$. The start node s_0 corresponds to the initial state z_0. The end nodes S_E are those $z \in Z^*$ without any outgoing edges. The set of precedence relations T corresponds to E^*. Finally, for each $s \in S$ with $z = (pm_z, e_z)$ being the structure corresponding to s, we assign the event associated with z to s (i.e., $el(s) := e_z$).

We assume that the functions `getNextEvents` and `fireEvent` as described above can be implemented based on the operational semantics of WSM nets [Rei00, Rin04]. As the informal description above suffices for explaining the basic ideas of transforming a process model into a `PEG`, we abstain from defining these functions.

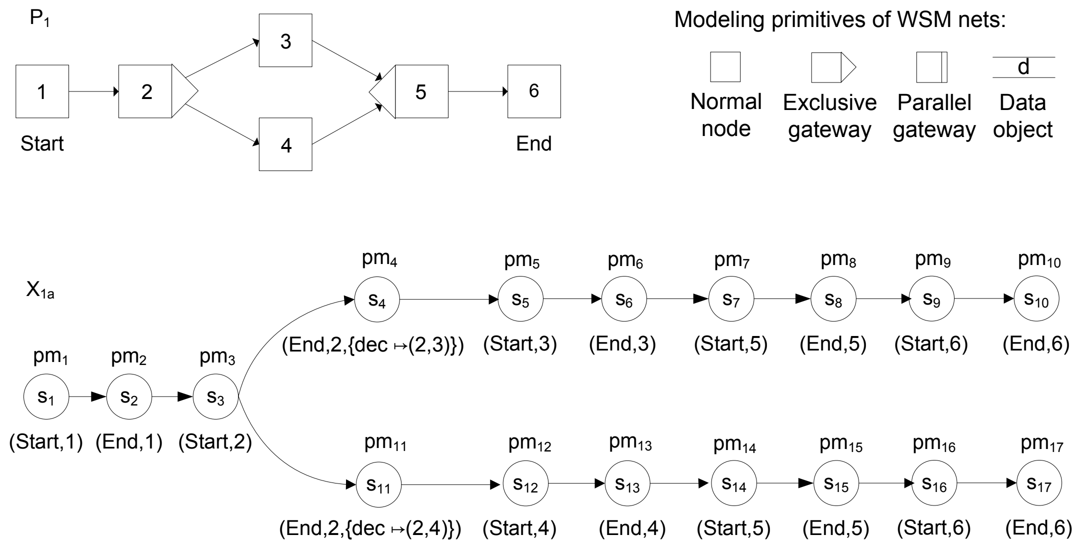


Figure 9.2.: A process model and the `PEG` obtained when applying Algorithm 4

Example 9.1 (Transformation of a WSM net process model into a `PEG`):
Fig. 9.2 depicts process model P_1 and `PEG` X_{1a} capturing all execution traces of P_1. X_{1a}'s upper path represents the execution of the upper while X_{1a}'s lower path represents the execution of the lower alternative branch. Each node of X_{1a} is associated with a START or an END event and a `PROCESS MARKING` of P_1 (cf. Algorithm 4). Recall that we assume that the decision on the outgoing branch of a split node is reflected in the designated parameter *dec* (cf. Section 5.2.3). The `PROCESS MARKINGS` of P_1 yielded during the exploration are depicted in Fig. 9.3.

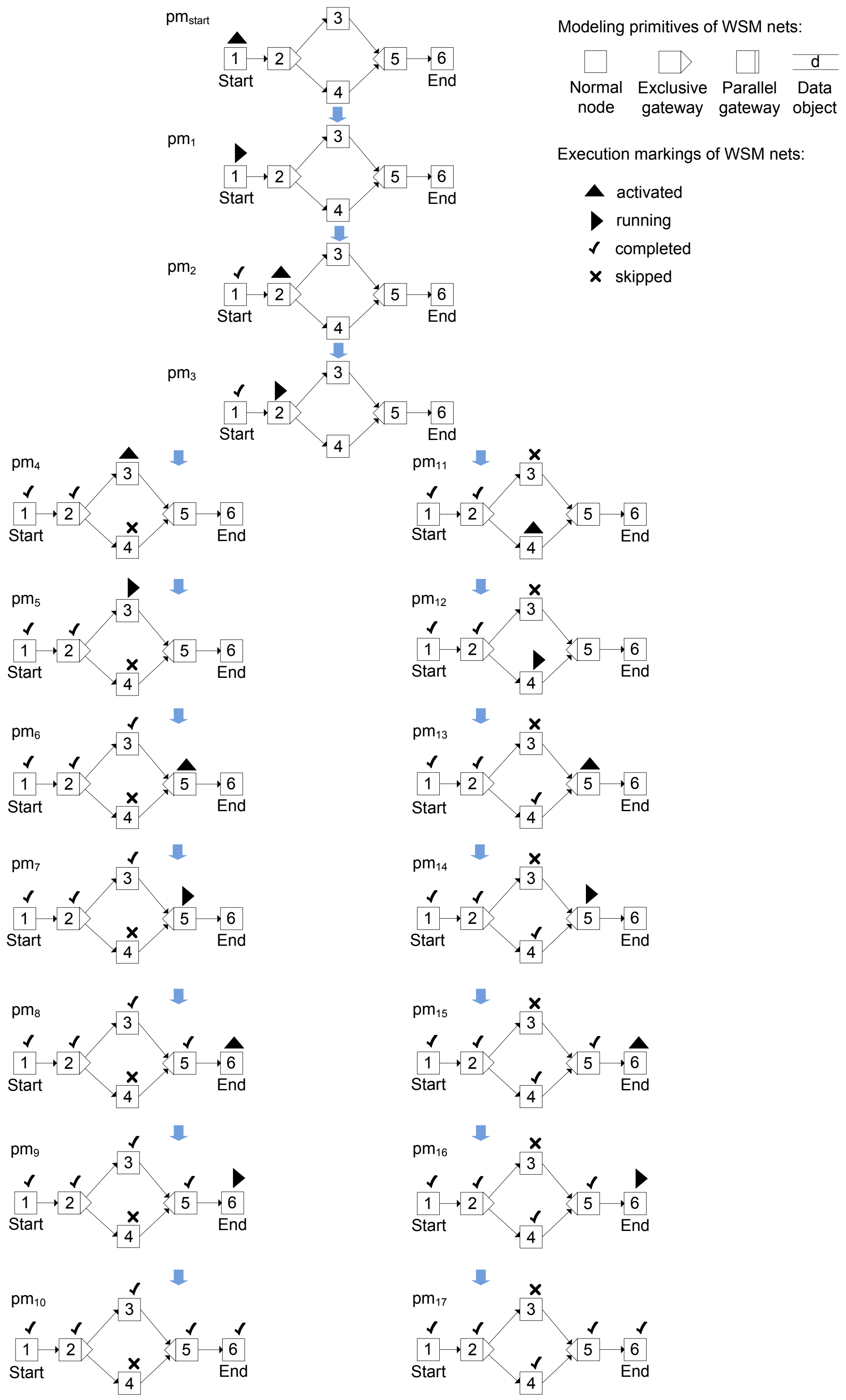


Figure 9.3.: Exploration of P_1 from Fig. 9.2

Algorithm 4 Transforming a WSM net into a PEG

```
1: $P$ is a WSM net;
2: $Z^*$ is set of $z = (pm_z, e_z)$ where $pm_z$ is a PROCESS MARKING of $P$ (cf. [Rei00, Rin04]) and $e_z$
   is an event;
3: $E^* \subset Z^* \times Z^*$ is a set of precedence relations;
4: $pm_0$ is the initial PROCESS MARKING of $P$;

   {INITIALIZATION}
5: $Z^* = \emptyset$;
6: $E^* = \emptyset$;
7: $z_0 = (pm_{start}, e_{start})$ where $pm_{start}$ is obtained from $pm_0$ after starting the start node of $P$
   and $e_{start}$ is the START event of $P$'s start node;
8: $Q = \{z_0\}$;

   {ITERATION}
9: while $Q \neq \emptyset$ do
10:   $z = (pm_z, e_z) = Q[1]$;
11:   $Q = Q \backslash Q[1]$;
12:   $TE_z = getNextEvents(pm_z)$;
13:   for all $e \in TE_z$ do
14:     $pm_{z'} = fireEvent(pm_z, e)$;
15:     $z' = (pm_{z'}, e)$;
16:     if $z' \notin Z^*$ then
17:       $Z^* = Z^* \cup \{z'\}$;
18:       $Q = Q \cup \{z'\}$;
19:     end if
20:     $E^* = E^* \cup \{(z, z')\}$;
21:   end for
22: end while
```

Algorithm 4 relies on the operational semantics of WSM nets to simulate the process execution. For each PROCESS MARKING, it is checked which events can be fired based on the node and data states. Thus, it is ensured that the resulting PEG and the original process model both exhibit the same execution traces. Hence, the resulting PEG and the process are *equivalent* according to Def. 5.7. While Algorithm 4 is tailored towards WSM nets, it can be adapted for other process description languages as well.

9.1.2. Optimizations

Evidently, the size of the PEG significantly affects the computational costs of compliance checks. Complex processes with extensive data flows typically result in very voluminous PEGs [KLRM+10, RWMR13] as illustrated in Example 9.2.

Example 9.2 (State explosion):
Consider the relatively small process model P_2 in Fig. 9.4. Then, PEG X_2 in Fig. 9.4 obtained by straight-forwardly exploring the model as for example by applying Algorithm 4 contains a considerable amount of nodes due to the exploration of the data dimension.

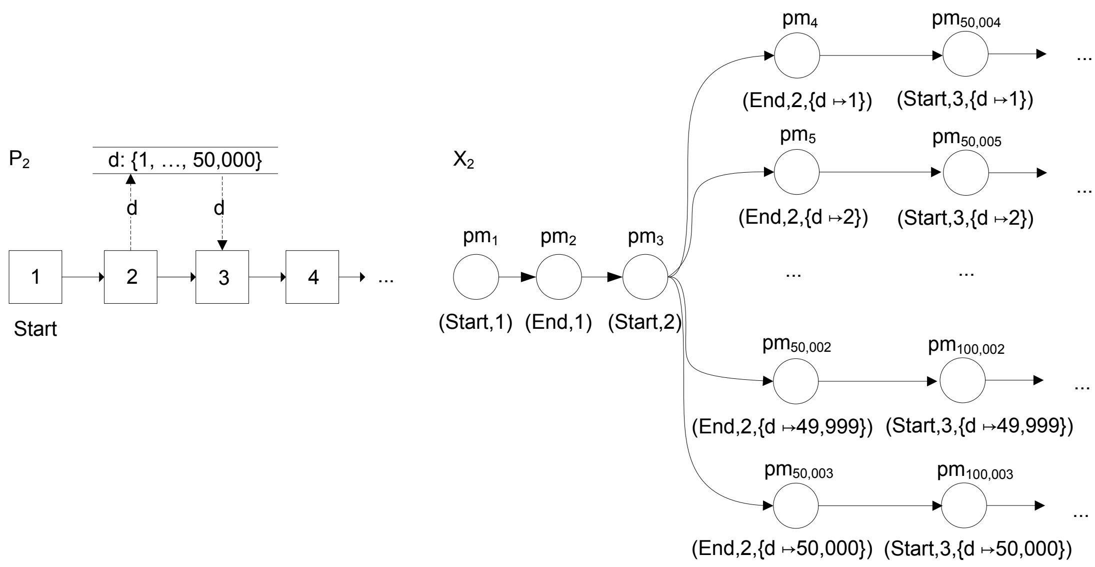


Figure 9.4.: State explosion due to exploration of the data dimension

This state explosion problem is also known from other verification settings (e.g., software verification or verification of reactive systems) [CGP99, Val96]. As mentioned in Sect. 5.2.6, equivalence between the process and the PEG is not necessary in order to correctly verify compliance. In practice, it is often desirable to conduct compliance checks on more compact model specifications. Against the background that compliance rules often only affect parts of a process, optimizations are possible. As state explosion is a well-known issue, we can benefit from extensive research. A general approach to tame the state explosion is *abstraction*. Abstraction aims at reducing the complexity by focusing on only the relevant parts of the system to be verified [BBF+01]. Apparently, this can be adopted for the verification of business processes. In the following, we describe different abstraction strategies.

9.2. Abstraction strategies

Abstraction aims at reducing the state space to be searched. The basic idea underlying all abstraction strategies is to focus on relevant parts of a system and to abstract from aspects that do not affect the properties to be checked [BBF+01, CGP99]. The objective is to obtain a smaller system model M' such that $M' \models \phi \Leftrightarrow M \models \phi$ where M denotes the original system model and ϕ denotes the property to be checked (i.e., the abstraction is conservative [Das03]). Ideally, M' is constructed without having to generate M.

In the context of business process compliance checking, this means that *instead of generating an equivalent* PEG (cf. Def. 5.7 in Sect. 5.2.6) from a process, a smaller PEG (i.e., containing less nodes / edges) is generated based on which compliance checking is conducted[3]. This improves the efficiency of compliance checks. In order to obtain a smaller PEG, we abstract from aspects that are indistinguishable to the compliance rules. In the following, we discuss two fundamentals types of abstraction strategies: rule-independent and rule-specific strategies. The former can be applied independently of the imposed CRGs and result in PEGs that can be utilized to verify compliance with arbitrary imposed CRGs. Rule-specific strategies exploit specific knowledge on the imposed CRG to be checked and abstract from aspects that are irrelevant to the specific CRG. While the resulting PEGs can be used to verify compliance with the specific CRGs, they will often be no longer suitable to verify the compliance with arbitrary CRGs. Though the considerations on abstraction discussed here are based on CRGs, they can also be adapted for other compliance rule languages.

In the following, we first provide relaxed equivalence notions for processes and PEGs to assess the effect of abstraction in Section 9.2.1. Then, we describe *node state abstraction*, a modification of the marking rules as defined by WSM nets, in Section 9.2.2. This modification enables us to reconsolidate execution paths induced by alternative branches. Then, *data state abstraction* is introduced in Section 9.2.3. This strategy aims at abstracting from data states once a data object is no longer relevant to the further execution. Applying this strategy together with node state abstraction enables us to avoid tree structures when generating PEGs. Moreover, we introduce *event type abstraction* in Section 9.2.4. Following the observation that CRG operational semantics can be applied to EX events in a more efficient manner, this abstraction strategy aims at favoring EX events over START and END events when creating PEGs. Section 9.2.5 addresses *structural abstraction* for reducing the process to the parts potentially affected by a compliance rule. *Data domain abstraction* discussed in Section 9.2.6 aims at reducing the domains of data objects, thus, reducing the overall state space. Finally, we discuss the adoption of *partial order reduction* from model checking in Section 9.2.7. Section 9.2.8 summarizes the abstraction strategies.

9.2.1. Relaxation of the equivalence notion for abstraction

In Sect. 5.2.6, we introduced the notion of equivalence between a process model and a PEG (cf. Def. 5.7). Informally, a PEG and a process model are considered equivalent if they convey exactly the same execution traces. As abstraction aims at producing smaller PEGs with less nodes and edges, the resulting PEGs are typically no longer equivalent to the original process models. However, the smaller PEGs are still supposed to preserve the compliance property to be checked. With respect to CRGs, there are two cases to be distinguished:

compliance-equivalent A PEG X is considered compliance-equivalent to a process model P iff X complies with all CRGs P complies with and vice versa.

rule-equivalent A PEG X is further considered rule-compliance-equivalent to a process model P with regard to a specific CRG R if X complies with R iff P complies with R.

[3]Here, we assume that the PEG is explicitly constructed before compliance checks are applied. However, this may also be conducted in a single step.

These relaxed equivalence notions are formalized in Def. 9.1.

Definition 9.1 (Relaxed equivalence notions)
Let P be a process model and X be an equivalent PEG of P (cf. Def. 5.7). Let further X' be a smaller PEG (i.e., containing less nodes / edges) that is not equivalent to X. Then, P and X' are called

- compliance-equivalent if for all CRGs R holds:
  - R is ENFORCED over $X' \leftrightarrow R$ is ENFORCED over X and
- rule-equivalent for a specific CRG R if holds:
  - R is ENFORCED over $X' \leftrightarrow R$ is ENFORCED over X.

Clearly, a PEG that is equivalent to a process model is also compliance- and rule-equivalent to this model. It should be noted that Def. 8.3 in Sect. 8.3 provides three notions to characterize compliance with an imposed CRG, namely ENFORCED, ENFORCEABLE, and IGNORED. The above-defined relaxed equivalence notions do not distinguish between ENFORCEABLE and IGNORED. This, however, suffices for the purpose of describing the abstraction strategies in Section 9.2 as we do not intend to formalize the abstraction.

9.2.2. Node state abstraction

Some process description languages assigning execution states to process nodes for process execution (such as WSM nets [Rei00, Rin04] or the process graphs used in the WASA project [Wes98]) perform dead path elimination after processing an alternative split gateway. As a result, nodes of the branches not selected are marked as SKIPPED (for WSM nets). Thus, for each selected branch a different set of nodes will be marked as SKIPPED in the resulting PROCESS MARKINGS.

Example 9.3 (Potential for abstracting from process node states):
A closer look at X_{1a} from Fig. 9.2 reveals that after the completion of node 3/4, respectively, both paths encoded through $<s_7, s_8, s_9, s_{10}>$ and $<s_{14}, s_{15}, s_{16}, s_{17}>$ are associated with the same execution trace suffix (i.e., the same event sequence). However, the paths are associated with different PROCESS MARKINGS as for the upper path, node 4 is marked as SKIPPED while for the lower path, node 3 is marked as SKIPPED.

As illustrated by Example 9.3, when executing the alternative join gateway, the particular state of predecessor nodes (i.e., whether they are SKIPPED or COMPLETED) does not affect the process nodes that can be executed in the future. Thus, at this point, the event trace suffixes (i.e., the firing sequences leading from the current state to the final state [EKR95]) that can be produced by the process are independent of whether previous nodes are marked as SKIPPED or COMPLETED. Hence, we can abstract from these node states when exploring the states of the process.

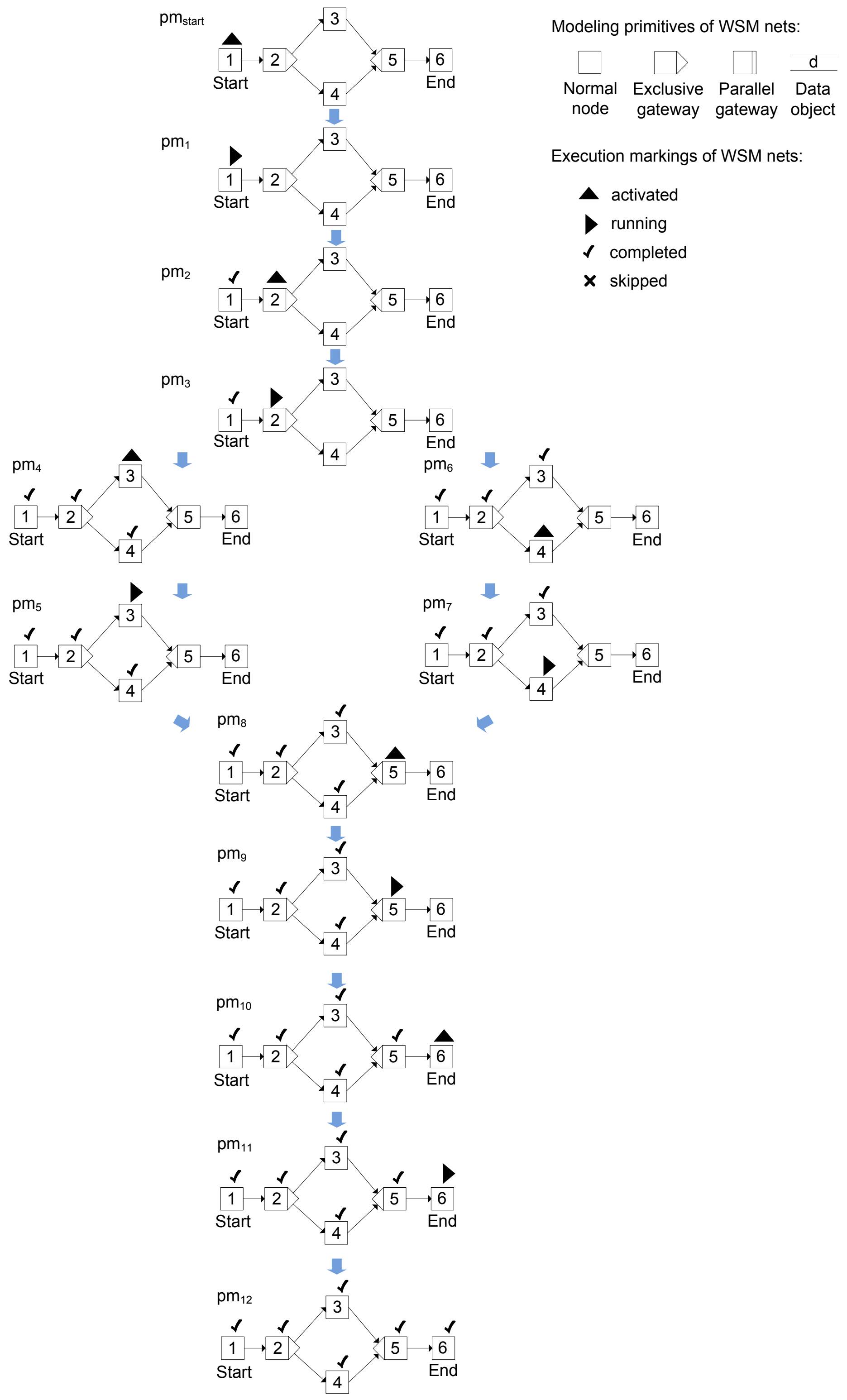


Figure 9.5.: Application of node state abstraction to the transformation of P_1 from Fig. 9.2

Example 9.4 illustrates how abstracting from node states reduces the states to be explored as the different paths yielded through an alternative split gateway can be consolidated when executing the join gateway. More information on node state abstraction can be found in the master thesis of Knuplesch [Knu08] conducted in the SeaFlows project.

Example 9.4 (Node state abstraction):
Fig. 9.5 shows the transformation procedure of P_1 from Fig. 9.2 when applying node state abstraction. In contrast to the original exploration illustrated in Fig. 9.3, the node state `SKIPPED` is replaced by `COMPLETED`. This, in turn, enables us to merge the two execution paths after completing the nodes 3 and 4. The resulting `PEG` X_{1b} depicted in Fig. 9.6 is smaller then `PEG` X_{1a} obtained by applying the original WSM net operational semantics.

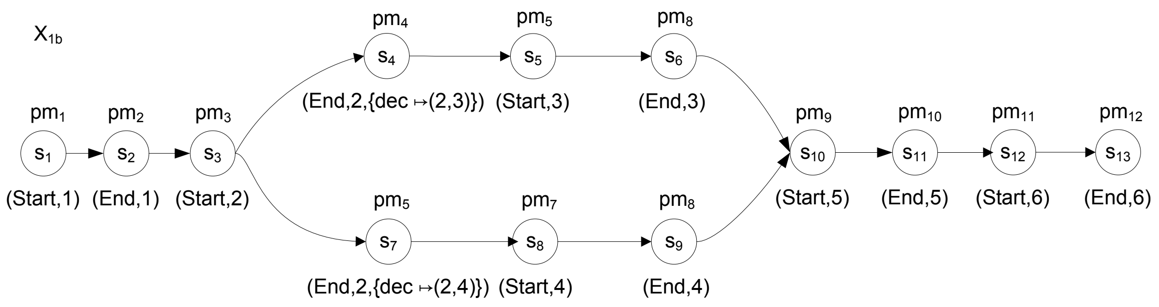


Figure 9.6.: The `PEG` resulting from the exploration as depicted in Fig. 9.5

As illustrated by Example 9.4, this approach leads to equivalent `PEG`s according to Def. 5.7 as the event traces are not altered by the abstraction.

9.2.3. Data state abstraction

Data flows are a major cause for state explosion. For example, n data objects with binary domains yield 2^n different data allocations for each marking of process nodes and edges. This bears the potential for optimization through abstraction as data assignments in `PROCESS MARKINGS` often do not add any extra information as illustrated by Example 9.5.

Example 9.5 (Exploitation of process data flows for abstraction):
Consider process model P_3 and the `PROCESS MARKINGS` yielded when exploring P_3 depicted in Fig. 9.7. Then, we can observe that after execution of process node 2, data object d is no longer read from. This indicates that once 2 is executed, d will neither provide data context to events to come nor will it have impact on any future gateway decisions and, thus, on the process control flow. This is reflected in the `PEG` X_{3a} obtained when exploring P_3 using the original WSM net operational semantics. The upper path after s_4 and the lower path after s_5 are both associated with the same execution trace suffix. This observation enables us to abstract from the data assignment in the `PROCESS MARKINGS` once the particular data object is no longer relevant to the further process execution.

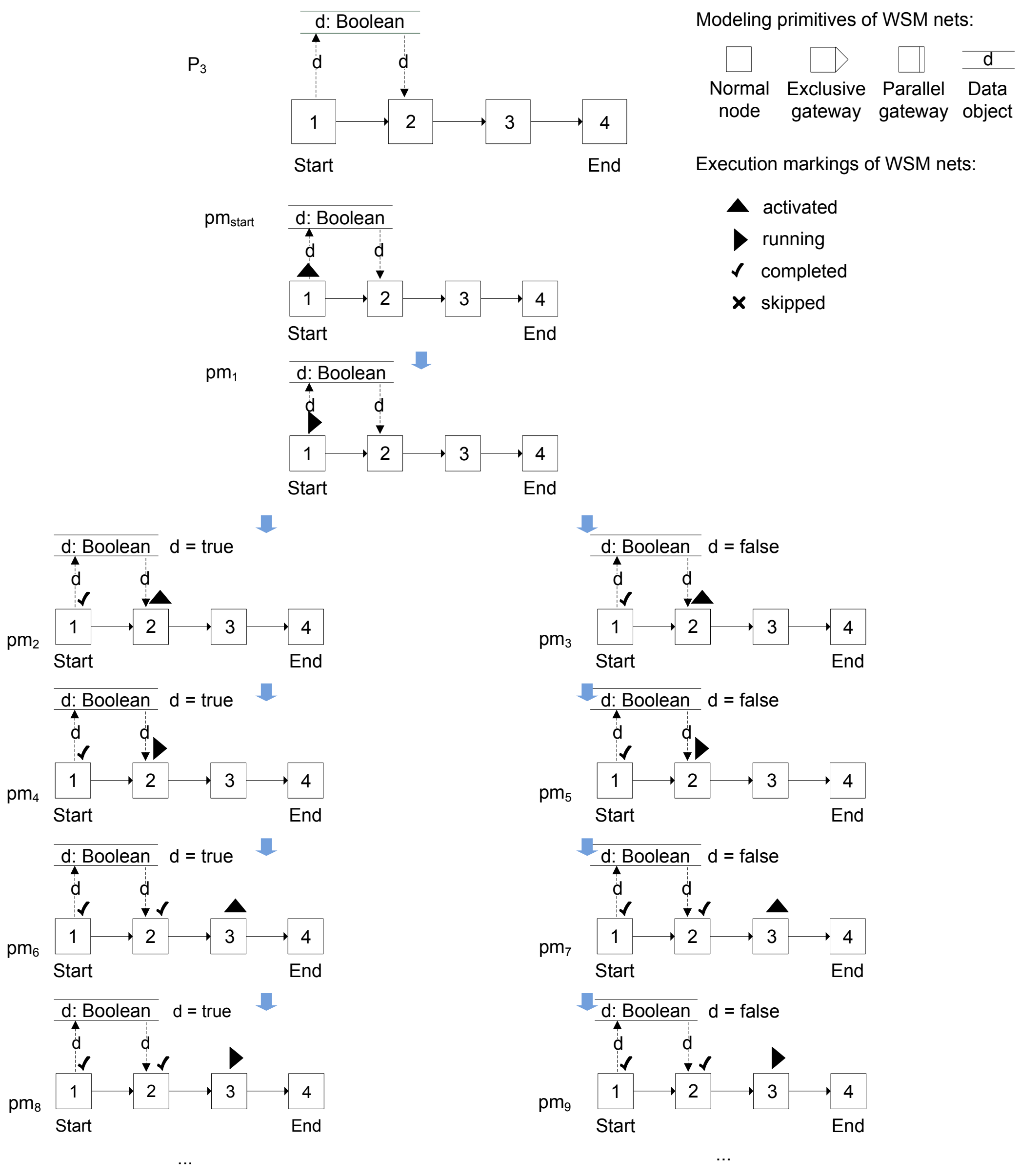


Figure 9.7.: Exploration of process model P_3 based on the original WSM net operational semantics

As shown by Example 9.5, once the states of a data object become indistinguishable with respect to the potential future states, the state space (in particular the PROCESS MARKINGS) can be reduced by abstracting from the state of that data object. Assuming explicitly modeled data flows, it is possible to identify when data objects will no longer be read from in a process. In WSM nets, for example, this information can be derived from the data reading edges. By exploiting this information, we can easily devise a modification of Algorithm 4 that performs data state abstraction to reduce the state space to be searched. If a data object is accessed from a node that is part of a loop, data state abstraction can be applied after being sure that the node is not going to be reexecuted.

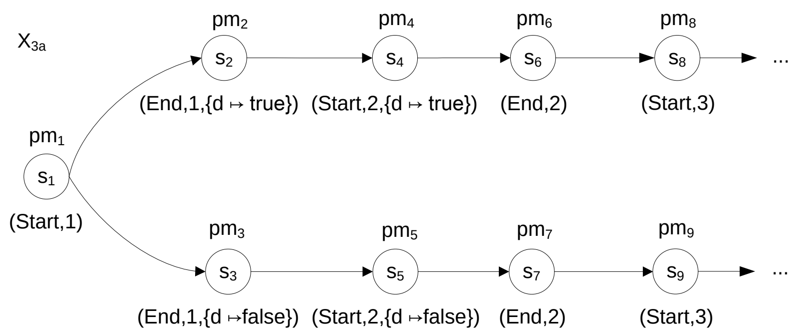


Figure 9.8.: The PEG resulting from the exploration depicted in Fig. 9.7

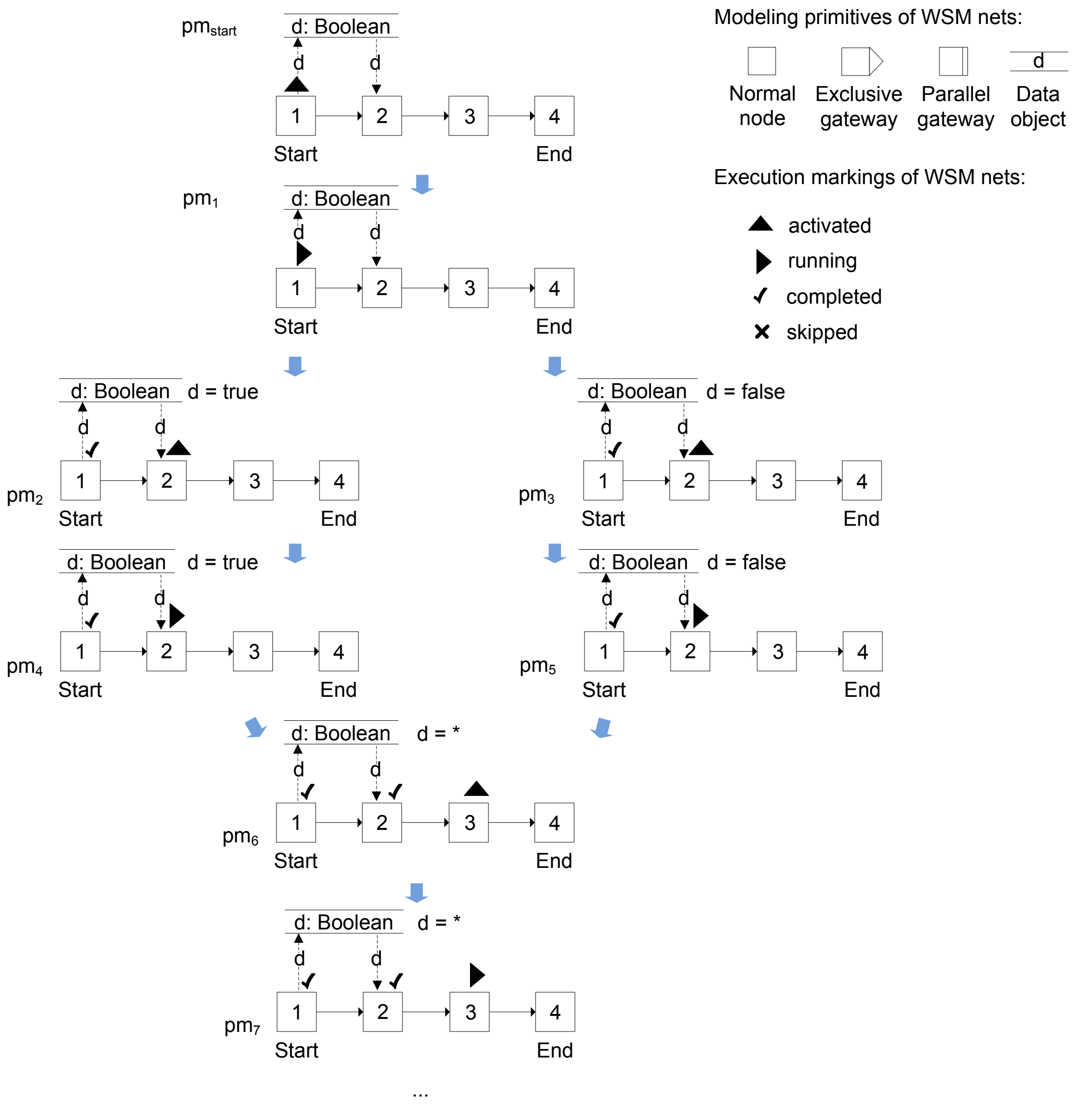


Figure 9.9.: Application of data state abstraction to the exploration of P_3 from Fig. 9.7

Example 9.6 (Data state abstraction):
Fig. 9.9 depicts the exploration process when transforming process model P_3 from Fig. 9.7 applying data state abstraction. The PEG obtained is shown in Fig. 9.10. In contrast to the original WSM net operational semantics, the state of d is abstracted from after completion of node 2 by assigning a wildcard to d. Apparently, data state abstraction enables us to merge the original execution paths to a single execution path.

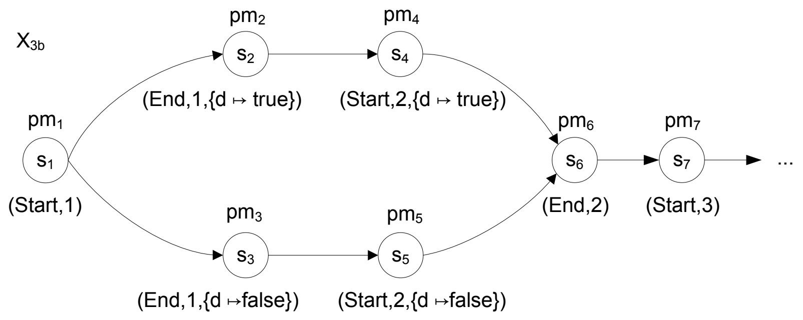


Figure 9.10.: The PEG obtained when applying data state abstraction to P_3 from Fig. 9.7

As data state abstraction is only applied if a data object is no longer read from, it affects the reachable PROCESS MARKINGS but not the resulting event traces. Therefore, data state abstraction leads to equivalent PEGs (cf. Def. 5.7).

9.2.4. Event type abstraction

As formalized in Def. 5.6, a PEG node is assigned a START or an END event. Thus, execution traces of PEGs consist of START and END events. While START and END events are particularly necessary to capture overlapping activities within an execution trace, they are rather superfluous when overlapped executions are not possible (as in strictly sequential processes). In such cases, the aggregation of START and corresponding END events to more compact EX events representing the atomic execution of an activity (cf. Def. 5.1 in Sect. 5.2.3) is advantageous as less states and transitions will have to be explored. Moreover, as discussed in Section 7.5.2, the operational semantics for executing CRGs over EX events is more efficient than over START and END events. For practical application, we, therefore, abstract from START and END events and utilize EX events when exploring a process without concurrently executable activities. For processes with concurrently executable parts, START and END events only have to be used for these parts.

Due to the block-structuring of WSM nets, concurrently executable process blocks can be easily identified a priori. By slightly modifying the operational semantics of WSM nets enabling the direct transition from the node execution state ACTIVATED to the execution state COMPLETED in the exploration procedure, we can easily apply event type abstraction to WSM nets.

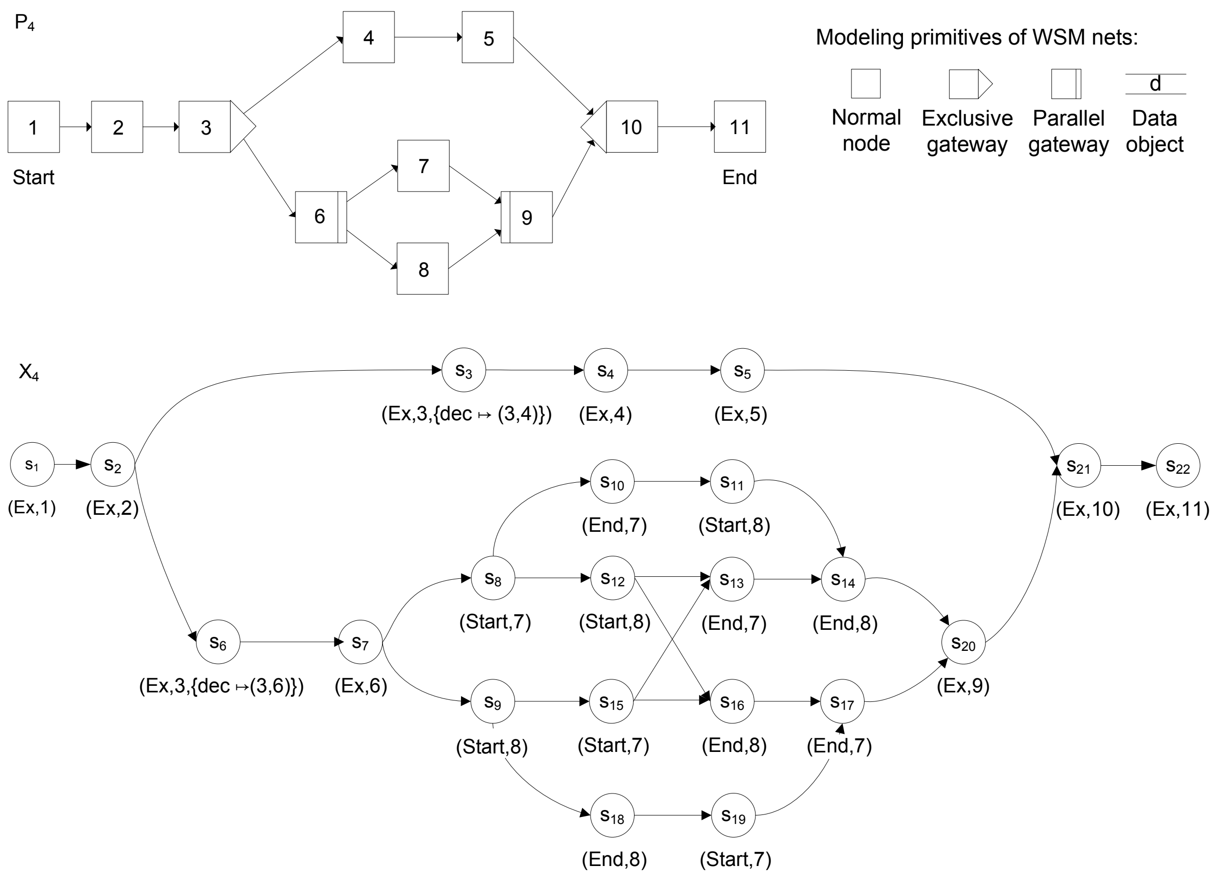


Figure 9.11.: A process model and the PEG obtained when applying event type and node state abstraction

Example 9.7 (Event type abstraction):
Fig. 9.11 depicts PEG X_4 obtained from process model P_4 when applying both event type and node state abstraction. Event type abstraction is only applied to nodes not located on a parallel branch. For the nodes 7 and 8, which can be executed in parallel, events of types START and END are generated.

The use of EX events leads to PEGs that can still be considered equivalent to the original process according to Def. 5.7. This is because no information that is relevant to CRGs (i.e., that can be distinguished by CRGs) is left out through the aggregation of START and END events to EX events and EX events can be disaggregated into pairs of START and END events. Note that event type abstraction as proposed here can also be applied for other compliance rule languages. Even compliance rules containing time conditions can be supported provided that the time information is still preserved in the aggregated EX event (e.g., through attributes for start and end time of the execution).

9.2.5. Structural abstraction

In realistic settings, process models can become quite huge containing up to hundreds or even thousands of artifacts [BRB07, RKBB12]. The verification of such processes can become intricate and intractable due to the state explosion. An exhaustive exploration of the state space results in a `PEG` that can be used for checking the compliance with arbitrary CRGs as it captures all potential behaviors. A specific CRG, however, typically only refers to particular activities conducted in a process. Therefore, the compliance with a specific CRG is often determined by only a relatively small part of the process model. This bears potential for optimization through abstraction from the parts of the process that are not affected by the CRG to be checked. An effective strategy to tame the state explosion is, therefore, to structurally reduce the original process by exploiting knowledge on the CRG to be checked. In this process, individual activities (i.e., process nodes) and data objects that are not relevant to the imposed CRG to complete process blocks may be removed from the model (cf. Fig. 9.12). The reduction can be conducted automatically. In particular, structural reduction without considering data conditions can be accomplished in linear time (w.r.t. the size of the model). The thus abstracted process model is then used to verify compliance as illustrated in Fig. 9.12. This strategy can lead to much smaller process models and, thus, induce a considerable reduction of the state space to be searched.

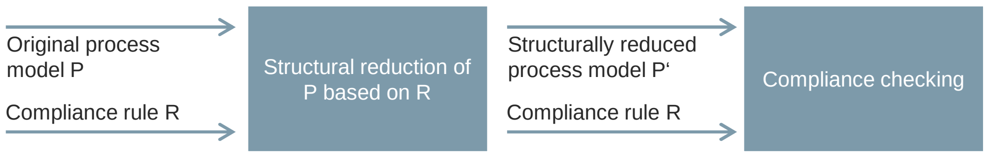


Figure 9.12.: Structural abstraction

In contrast to rule-independent abstraction approaches like data state abstraction, the structural abstraction utilizes knowledge on the particular CRG to be verified. As a result, the abstracted process model may not be suitable for checking compliance with other CRGs.

In the following, we discuss existing structural reduction approaches. Clearly, the particular reduction approach is specific to the process description language employed. Awad et al. introduce in [ADW08] an approach for reducing BPMN process models based on BPMN-Q [Awa07], a language for querying BPMN processes by matching the process with a BPMN-Q query graph. This reduction approach is conducted in two steps. For the compliance rule specified in BPMN-Q notation, BPMN-Q queries are generated to be evaluated against the process model repository. The BPMN-Q queries are supposed to identify those process models that are relevant to the compliance rule. Only the selected process models will have to be verified against the compliance rule in the second step. The first step is supposed to reduce the effort for manually identifying process models affected by a compliance rule. It is, for example, checked whether all activities of the compliance rule are contained in the process model. It should be noted that this does not correspond to the semantics of CRGs. For CRGs, it is important to distinguish between whether activities of the rule's antecedent or of the rule's consequence are not contained in the process model. In particular, if activities matching `AnteOcc` nodes are not contained in a process model, the corresponding CRG would be satisfied as the rule does not apply. If, however, activities matching `ConsOcc` nodes are not contained in the process model, the CRG may be violated. This distinction is considered by the approach proposed in [RMM11], which clusters process models according to the compliance rules that are potentially activated over the models. For the actual structural abstraction, Awad et al. introduce nine reduction rules that can be applied to dwindle the process model to the parts that are relevant to the compliance rule to be checked. Some of the reduction rules are adapted from approaches that use graph reduction to verify correctness [Men07, DAV05, SO00]. As far as we know, the reduction rules introduced by Awad et al. [ADW08] do not yet consider data flows.

Rabbi et al. propose a model reduction approach (also referred to as *model slicing*) in [RWMR13], which is inspired by *program slicing* [Tip95]. The approach is able to deal with data flows. It is implemented in the NOVA workflow framework and is designed for process models in CWML (compensable workflow modeling language). For a LTL_X[4] formula ϕ, the approach generates a reduced process model M' from a model M such that the truth value of ϕ is preserved in M' (M and M' are stuttering equivalent with respect to ϕ). In order to reduce the process model, the approach parses the model, generates a syntax tree for which the reduction is applied. It should be noted that the compliance rules addressed by Rabbi et al. are imposed on the states of the process execution while compliance rules addressed in our work impose constraints on the events of a process. Thus, the stuttering equivalence criteria is not directly applicable for the verification of CRGs. However, we believe that the approach can be adapted to consider compliance rules on events as well.

Structural reduction can be conducted as a preprocessing step prior to exploring the process (as suggested by [ADW08]). Generally, we believe that structural reduction will be essential for the practical application of compliance checking as it can considerably reduce the cost of compliance checks.

9.2.6. Data domain abstraction

Not only the amount of data objects but also the size of their domains can considerably amplify the state explosion. For n data objects with domain D, the state space of the data objects is D^n for each execution state of process nodes and edges. Data domain abstraction aims at reducing the domain of data objects, thus, reducing the states to be explored for compliance verification. Data domain abstraction is based on abstraction through representatives [Pel96]. The basic idea is to exploit knowledge on the imposed CRG and the data conditions used as branching conditions (for alternative gateways) to identify suitable equivalence classes whose members are indistinguishable to the imposed CRG. Then, the data domain of the corresponding data object can be reduced to a set of representatives of the equivalence classes. The reduction from concrete to abstract states is also known as *abstract interpretation* in literature [CC77].

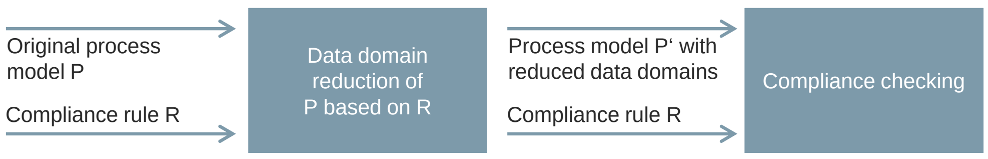

Original process model P
Compliance rule R
Data domain reduction of P based on R
Process model P‘ with reduced data domains
Compliance rule R
Compliance checking

Figure 9.13.: Data domain abstraction

[4]LTL_X refers to LTL without the nexttime operator.

The general procedure is illustrated in Fig. 9.13[5]. Based on the imposed CRG and the process model, an abstracted process model and an abstracted CRG can be automatically derived. Thus, compliance checking can be conducted for the reduced process model. As the data domains are reduced by exploiting the compliance rule to be checked, the resulting process may not be suitable for checking compliance with arbitrary compliance rules. Example 9.8 illustrates the idea of data domain abstraction. Further details are described in [KLRM$^+$10, Knu08].

Example 9.8 (Data domain abstraction):
Consider, for example, process model P_5 from Fig. 9.14. Let us further assume that the compliance rule "B must occur after A" is imposed on P_5. Then, for the correct verification of P_5 against the compliance rule with respect to Def. 8.3, it suffices to consider two cases, namely a) when A is executed with d exceeding 5000 and b) when A is executed with d not exceeding 5000. For the first case, the compliance rule will be satisfied. For the latter case, the compliance rule will be violated. Hence, the compliance rule is ENFORCEABLE over P_5 (cf. Def. 8.3 and Def. 8.2). We can, therefore, reduce the domain of d to the abstract states $\{d > 5000, d \leq 5000\}$. This way, only two different states instead of $\{1, 2, \dots, 5001, \dots\}$ for $d \in \mathbb{N}$ will have to be considered for d when exploring P_5.

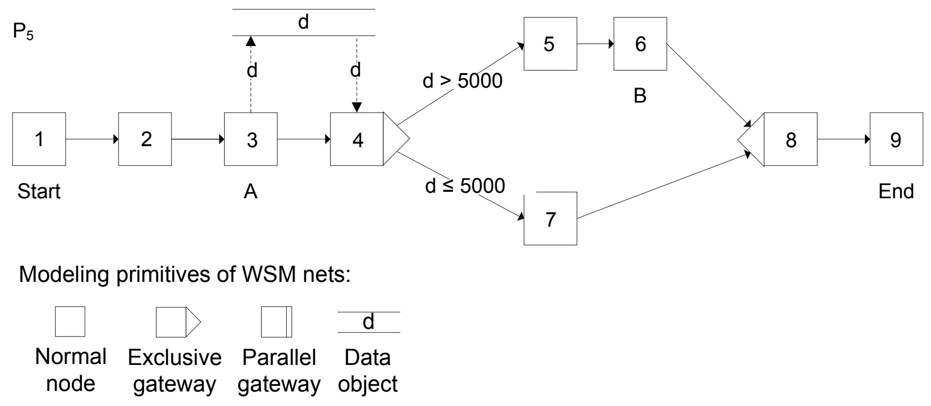


Figure 9.14.: A process model containing data-based branching conditions

Similar to structural abstraction, data domain abstraction can be conducted as a preprocessing step prior to the exploration [KLRM$^+$10]. It is noteworthy that the combination of data domain abstraction with structural abstraction (cf. Section 9.2.5) is most effective. In particular, structural abstraction reduces the process model to the parts that are truly relevant to the compliance rule to be checked. In the structural abstraction process, irrelevant data objects and irrelevant data-based gateways can be removed. For the remaining relevant data objects, data domain abstraction can be applied to further reduce the state space.

[5]Note that for compliance rules containing data conditions, it can become necessary to adapt the compliance rule to the reduced domain of the data objects. This is neglected here as it is a minor rewriting task.

9.2.7. Partial order reduction

The representation of concurrently executable transitions as interleavings of events is a potential cause of state explosion and a well-known issue in the verification of concurrent reactive systems [CGMP99, God94]. *Partial order reduction* aims at reducing the state space by abstracting from interleavings of concurrently executable transitions when the orderings are indistinguishable to the property to be checked [God94, Pel96, CGP99, Sch03]. Thus, partial order reduction implements abstraction through representatives [Pel96, Pel98]. The basic idea is to not build the complete state graph but a reduced state graph. With respect to the exploration of the state space, this means that only a subset of enabled transitions is expanded. Thus, given a property, only a subset of the state space will be explored [God94]. There are different approaches for selecting the transitions to be expanded (e.g., [Pel96, God94]). Partial order reduction can be implemented such that specific properties, e.g., deadlock-freeness or LTL properties are preserved [God94, Pel98, CGP99].

In business process compliance verification, the exploration of all possible orderings of concurrently executable activities is a potential source of state explosion. If the variants of orderings of events from overlapped activity executions are indistinguishable to the compliance rule to be checked (as is the case for CRGs), reducing the variants is advisable.

Example 9.9 (Partial order reduction):
Consider PEG X_4 depicted in Fig. 9.11. Then, we notice that X_4 contains all interleavings of the nodes 7 and 8, which can be executed concurrently. Hence, X_4 contains four variants for the overlapped execution of the nodes 7 and 8 (i.e., 7 is started before 8 is started and completed before 8 is completed, 7 is started before 8 is started and completed after 8 is completed, and so on) depicted in Fig. 9.15. However, these variants of overlapped executions are indistinguishable to CRGs and also to a bulk of other compliance rule languages proposed in literature (e.g. [ASW09, AWW09]). Unless a compliance rule specifically constraints overlapped executions (e.g., through constraints on START and END events), such overlapping variants are not distinguishable. Therefore, a PEG containing all these variants contains no additional information over a PEG containing only one overlapped execution with respect to CRGs.

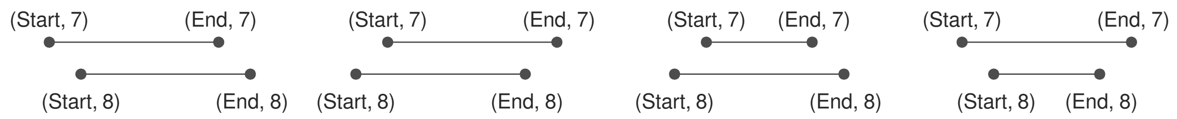


Figure 9.15.: Variants of the overlapped execution of the nodes 7 and 8

Variants of overlapped executions as illustrated by Example 9.9 are typically indistinguishable to compliance rules that treat the execution of activities as an unit (e.g., CRGs and many approaches in literature such as [ASW09, AWW09]). Note that this does not prohibit time conditions (e.g., on the time distance or deadline of activities). It is noteworthy that for compliance rules employing an interval-based model [All83] or constraining the overlapped execution of activities (e.g., A must be started one day after the start of B and completed one day before completing B) the overlapping variants can become distinguishable to the compliance rules.

The partial order methods for the verification of reactive concurrent systems as described in [God94, Pel98] are not directly applicable to business process compliance verification as these approaches rely on the commutativity of concurrently enabled transitions (i.e., different orderings of commutative transitions that lead to the same states). In particular, different orderings of transitions will still yield similar paths with respect to the propositions that hold in the states along the paths. In [Pel98, CGMP99], for example, orderings of concurrently executable transitions that lead to so-called stuttering equivalent paths are reduced as these paths are indistinguishable to LTL_X[6] properties. However, against the background that compliance rules impose constraints on the occurrence, absence, and ordering of business process *events* (i.e., the transitions in this case), it is clear that their ordering can be subject to the compliance property. Specifically, the ordering of the events directly affects the propositions that hold in each state as the state labels are constituted by the event through which the state is reached.

Nevertheless, as Example 9.9 showed, compliance checking obviously bears potential for partial order reduction. As illustrated in the example, CRGs and also a plethora of other compliance rule specification formalisms do not distinguish between variants of overlapped executions. This can be exploited for partial order reduction. In addition to that, we can also think of strategies that exploit the particular compliance rule to be checked to reduce the orderings to be explored. For example, if a compliance rule does not impose direct or indirect constraints on the ordering of two concurrently executable activities, any interleaving of these activities will suffice for compliance verification. We believe that the existing partial order reduction approaches can be adopted and extended to support these cases. This is left to future research.

9.2.8. Summary

The abstraction strategies described in this chapter aim at reducing the state space that must be searched in order to verify business process compliance. They exploit properties of business process models such as the explicit data flows or properties of the compliance rule to be verified. Clearly, their applicability depends on the business process description language employed. For example, node state abstraction only becomes effective for process description languages whose operationalization foresees the explicit marking of discarded nodes such as WSM nets [Rei00]. Other strategies, such as structural abstraction, are rather generic and not restricted to specific process description languages. While some abstraction strategies can be incorporated into the exploration almost without additional costs (i.e., node state abstraction, data state abstraction, and event type abstraction), other strategies are suitable for preprocessing (e.g., structural abstraction). In conclusion, we believe that the proposed strategies from our research and from literature can contribute to making compliance checks even of huge models tractable. The strategies were discussed with respect to CRGs and related compliance rule languages focusing on constraining occurrence and ordering of activities. For compliance rules with more sophisticated conditions (e.g., on overlapping intervals), however, similar considerations with respect to differentiability can be applied to identify suitable abstraction strategies.

Extensive research in the model checking community has resulted in abstraction strategies, which can be adopted for business process compliance verification. Some of the abstraction approaches described are based on existing abstraction strategies (e.g., abstraction through

[6]LTL_X refers to LTL without the nexttime operator.

representatives [CGP99, Pel96] or merging states [BBF^{+}01]). In addition, the adoption of further existing approaches, such as the exploitation of symmetries [ID96, CGP99], for compliance verification can be investigated. We leave this to future research.

9.3. Conveying compliance violations

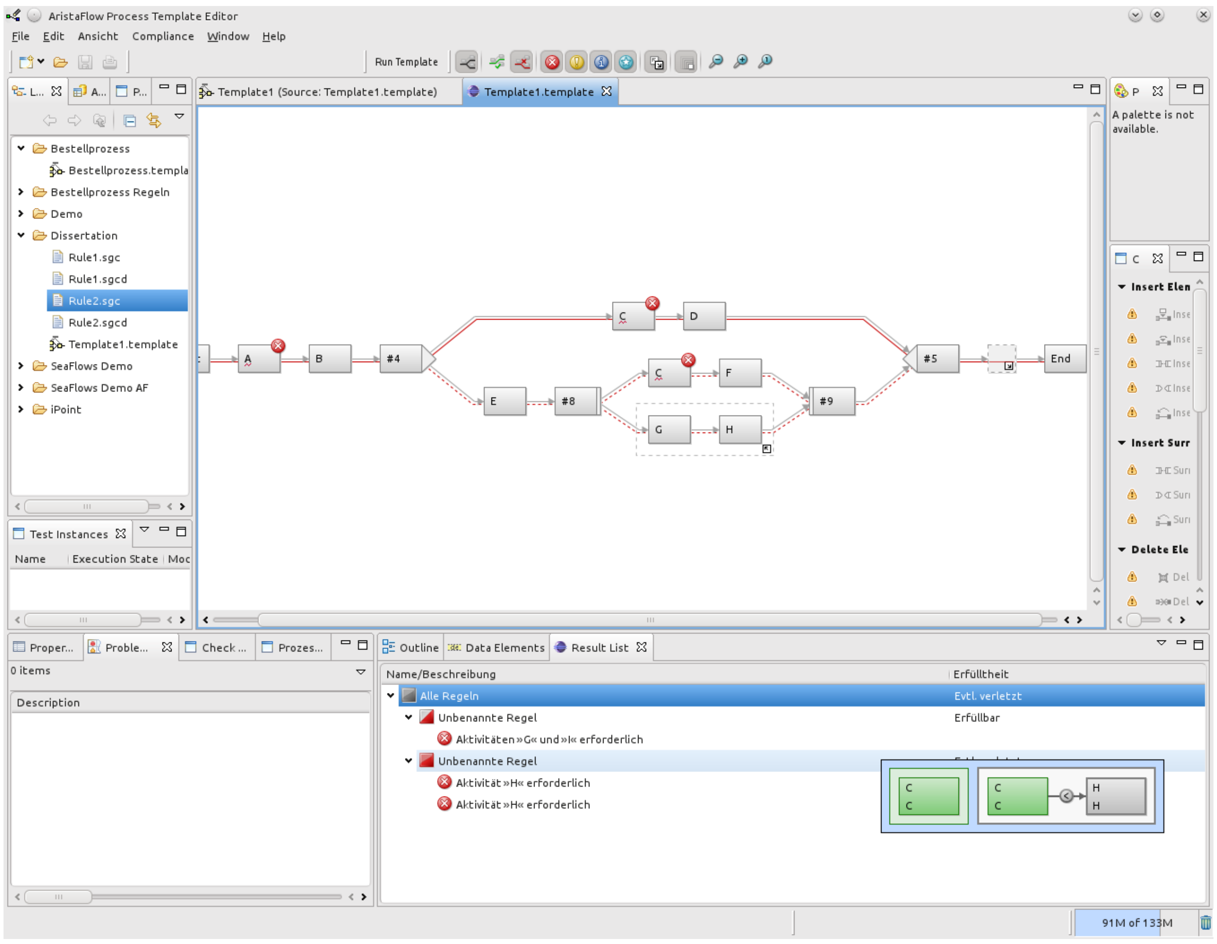
So far, we focused on techniques to detect compliance violations within a process model or occurring during process execution. Clearly, each detected compliance violation has to be conveyed to the respective stakeholders (such as process designers working on a process model or process supervisors in charge of a running process instance). The challenge is to pinpoint the source of the violation and to convey it to the stakeholders in an intelligible way. In the following, we describe ideas on conveying compliance violations. Section 9.3.1 addresses the visualization of compliance violations within process models. Runtime is addressed in Section 9.3.2. Section 9.3.3 briefly describes further related issues and provides references to literature.

9.3.1. Visualizing and explaining compliance violations in process models

Conceivably, it will often be difficult for a process designer working on a complex process model comprising up to hundreds of artifacts to pinpoint the source of compliance violations without the system's assistance. In order to be able to evaluate a compliance violation and to apply adequate remedies, it is vital for a process designer to know which process instances will be affected by the violation. To visualize the parts of the process that are affected by a compliance violation, the detected counterexamples (cf. Sect. 8.2) can be "replayed" in the process model to highlight respective parts of the model. However, even with highlighting the parts of the process model affected by a compliance violation, identifying the root-cause of the violation can still be a challenge. Therefore, it can be useful to also relate these parts to the particular compliance rule.

In [AW09, AWW09], Awad et al. address the visualization of compliance violations in process models. In order to verify compliance with a rule, the process model is checked for patterns of rule violations (i.e., referred to as *anti-patterns*). If an anti-pattern of a compliance rule can be found in a process model, a compliance violation is detected. The anti-patterns are expressed as BPMN-Q queries [Awa07, ADW08]. The subgraph of the process model matching the anti-pattern queries can be shown to the user. This approach is designed to work with purely structural compliance rules that are composed from a fixed set of patterns. In [AWW09], Awad et al. address the explanation of data-aware compliance rules in [AWW09]. Here, the set of rules introduced in [ADW08] is extended with data conditions that accompany the activity specification, for example the *conditional precedes* rule ("Before opening the bank account, the respondent bank rating must be accepted."). For these rules, visualization and explanation are realized based on *temporal logic querying* to extract data conditions of a violation and processing anti-pattern queries to show the subgraph of the process affected by the violation.

In the SeaFlows project, concepts on conveying compliance violations were developed in the context of a master thesis [Mer10] aiming at providing assistance for root-cause analysis. In contrast to the work done by Awad et al. ([AW09, AWW09]), this approach focuses on only



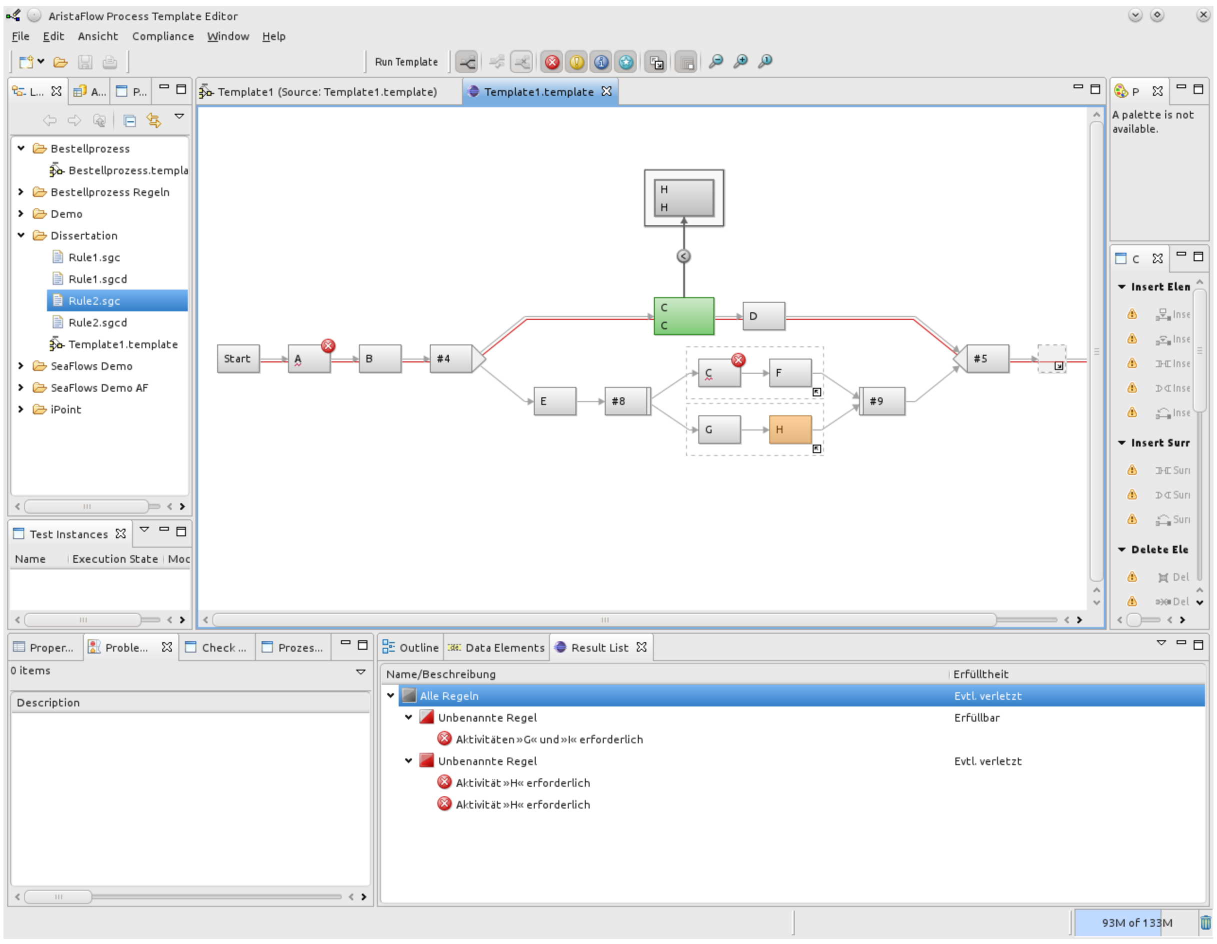
Name/Beschreibung	Erfülltheit
Alle Regeln	Evtl. verletzt
Unbenannte Regel	Erfüllbar
Aktivitäten »G« und »I« erforderlich	
Unbenannte Regel	
Aktivität »H« erforderlich	
Aktivität »H« erforderlich	

Figure 9.16.: Highlighting parts of a process model affected by a compliance violation

the visualization. To provide an overview of the parts of the model affected by a violation, the approach proposed in [Mer10] highlights the paths of the process model in which a compliance rule is violated / satisfied[7]. Fig. 9.16 shows a screenshot of the proof-of-concept implementation. The corresponding compliance rule requires H to be executed after C. The dotted red line indicates that the compliance rule can become violated when executing that part of the process (in this case, the violation depends on the execution order of the concurrently executable activities). So far, the visualization concepts do not consider data conditions of compliance rules. However, it can be extended to deal with more sophisticated compliance rules.

In addition to highlighting the paths of the process model, the approach exploits the graph structure of CRGs to provide details on the compliance violation aiming at facilitating root-cause analysis. As depicted in the screenshot in Fig. 9.17, the nodes involved in the violation are highlighted. When clicking on the violation report, the user is provided with details on the violated compliance rule by relating directly to the activity involved. In Fig. 9.17, it is shown that activity C contained in the process model requires a subsequent H. In addition, the system tries to assist the process designer in compensating the violation by highlighting nodes

[7]The input data for this can be obtained from our compliance checking framework as described in Section 8.2.2.2.



Name/Beschreibung	Erfülltheit
Alle Regeln	Evtl. verletzt
Unbenannte Regel	Erfüllbar
Aktivitäten »G« und »I« erforderlich	
Unbenannte Regel	Evtl. verletzt
Aktivität »H« erforderlich	
Aktivität »H« erforderlich	

Figure 9.17.: Assistance in identifying the root-cause of a compliance violation

associated with activity *H* contained in the process. This will be useful, for example, if the process designer misplaced an activity in the process model. The work done in [Mer10] was implemented within the SeaFlows Toolset, which will be described in Chapter 10

As described in Section 8.2.2.3, the structure and markings of MarkStructures can be exploited to derive explanations for compliance violations. Based on that, advanced concepts for conveying the root-cause of compliance violations can be developed. This together with the concepts for visualizing affected parts of the process model and the advanced concepts developed in [Mer10] can provide the basis for a sophisticated visualization framework.

9.3.2. Runtime support

At runtime, compliance information on running process instances can be gathered in a process cockpit as implemented in existing tools. In fact, various visualization concepts exist. For example, the traffic lights metaphor is often used. A major benefit of using CRGs for compliance monitoring is that the effective compliance state is always reflected in the MarkStructures.

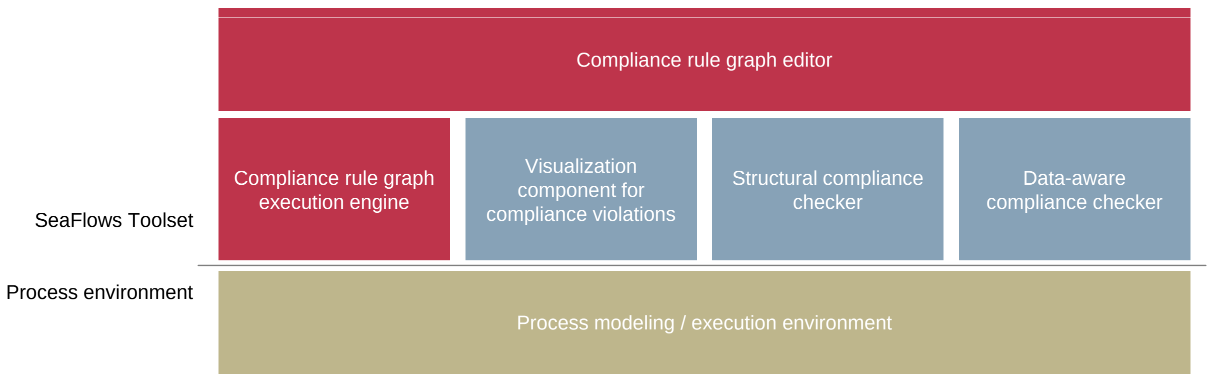
Thus, useful information can be derived from the MarkStructures even before a violation becomes manifest [LRMKD11]. In Section 7.4.2 and 8.4.1, we described how the MarkStructures can be exploited to explain the root-cause of a detected compliance violation. In addition, we showed that information on how to render a currently violable MarkStructure satisfied can be derived. This can be used for preventing violations. Further details are provided in our paper [LRMKD11]. When monitoring process instances with a defined process model, concepts for visualizing violations in process models as described in Section 9.3.1 can be applied to show the parts of the process affected by the violation.

9.3.3. Further issues

Clearly, assistance beyond the pure visualization and explanation of detected compliance violations is desirable. This particularly includes the (semi-)automatic resolution of compliance violations in process models. So far, this has been addressed by only a few approaches [ASW09, GK07, GK07] and is beyond the scope of this thesis. Finally, we see the potential of applying verbalization techniques to derive intelligible explanations for compliance violations from CRGs. This should be addressed in future research.

10 Prototype implementation

In this chapter, we present SeaFlows Toolset, the proof-of-concept implementation developed in the context of this thesis. As a result of the research methodology employed (cf. Section 2.2.3), different techniques and approaches explored in our research resulted in respective prototype implementations. The rationale behind that was to test and explore the feasibility of ideas and approaches. Thus, SeaFlows Toolset comprises different tools. A major part of SeaFlows Toolset has been presented at demo sessions [LKRM$^+$10] or in publications [KLRM$^+$10, LRMKD11, LIMRM12, KR11a].



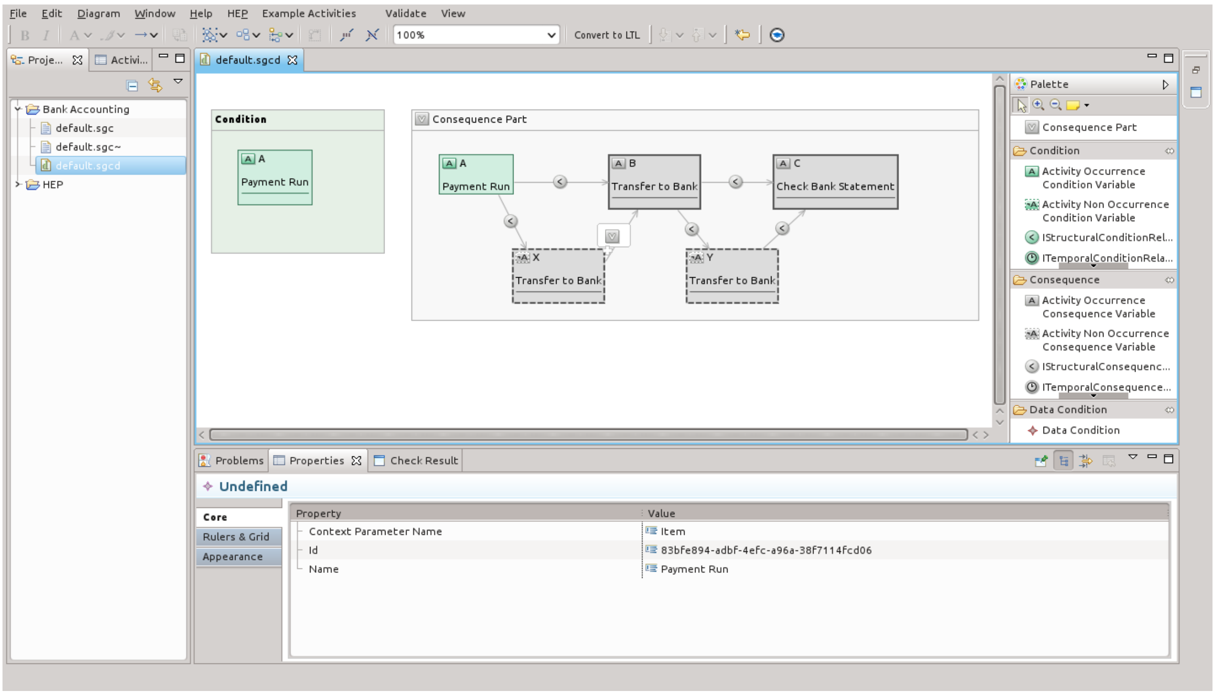
Compliance rule graph editor
Compliance rule graph execution engine
Visualization component for compliance violations
Structural compliance checker
Data-aware compliance checker
SeaFlows Toolset
Process environment
Process modeling / execution environment

Figure 10.1.: Components of SeaFlows Toolset

Fig. 10.1 illustrates the components of SeaFlows Toolset and their interactions with the process modeling and execution environment as provided by PrMSs. The parts that implement the concepts on CRGs presented in Chapter 6 and their operational semantics described in Chapter 7 are shown in red. The *compliance rule graph editor* is a visual modeling environment for CRGs and CRG composites. We describe this tool in more detail in Section 10.1. The *compliance rule* *graph execution engine* implements the operational semantics of CRGs and can be applied for verification of process models and running process instances. Details are provided in Section 10.2. The *visualization component for compliance violations* focuses on the visualization of compliance violations within process models. The underlying concepts are described in Section 9.3.1 and further details are provided in Section 10.3. The *structural compliance checker* was developed to test a compliance verification approach based on applying certain checks to the process structure [LKRM$^+$10]. The *data-aware compliance checker* implements abstraction strategies for taming state explosion (cf. Section 9.2) and uses a model checker for conducting compliance checks [KLRM$^+$10, LKRM$^+$10]. The *structural* and the *data-aware compliance checker* are presented in more detail in Section 10.3.

10.1. Compliance rule graph editor

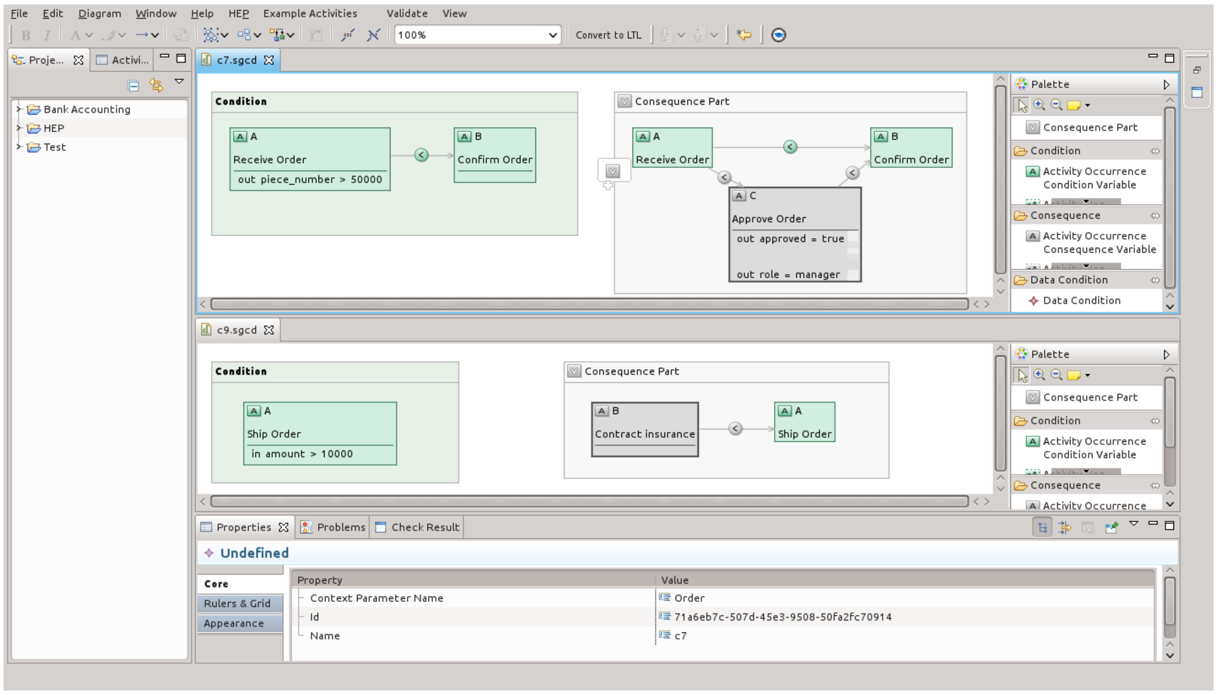
The *compliance rule graph editor* (CRG editor in brief) is the modeling environment for CRGs. The main graphical user interface (GUI) is shown in Fig. 10.2. On the left hand side, one may browse existing compliance rule projects. The actual CRG editor constitutes the main part of the GUI. A CRG is modeled by defining its antecedent and its consequence patterns. This is done in separate areas shown as boxes in the GUI. The rationale behind this is to create awareness for the different semantics of the CRG antecedent and the CRG consequences. This is supposed to help the compliance rule modeler to reflect on the designated rule structure. Multiple consequence CRGs can be defined for a CRG composite. For that, multiple boxes for defining consequences are provided in the GUI.



Property	Value
Context Parameter Name	Item
Id	83bfe894-adbf-4efc-a96a-38f7114fcd06
Name	Payment Run

Figure 10.2.: The compliance rule graph editor

Each CRG node is assigned a unique label, an activity type, and multiple data conditions as indicated in Fig. 10.3. As mentioned in Section 7.7.2, a modeled CRG can be assigned a data object that sets the context for the compliance rule activation. When executing the CRG over an execution trace, the data object works as selection criteria for events that are related and, hence, can contribute to the same rule activation. This constitutes a data relation between the involved events as they all necessarily refer to the same data object context.



Property	Value
Context Parameter Name	Order
Id	71a6eb7c-507d-45e3-9508-50fa2fc70914
Name	c7

Figure 10.3.: Modeling CRGs in the compliance rule graph editor

Activity types can be defined within the editor for testing or can be imported from an event model source (e.g., from an event log) or from a PrMS. The CRG editor is specifically integrated with the commercial PrMS AristaFlow BPM Suite[1], which, in turn, is based on the research done in the ADEPT project [Rei00]. Thus, activity types defined in the activity repository of AristaFlow BPM Suite managing process artifacts relevant within the business domain can be assigned to CRG nodes. This ensures that processes to be verified and modeled compliance rules both refer to the same set of artifacts.

The implementation was done using the Eclipse Modeling Framework (CMF) and the Eclipse Graphical Modeling Framework (GMF). Further checks to enforce certain conditions on the modeled CRGs can be defined. In the implementation, it is, for example, checked whether labels assigned to CRG nodes are unique. A simple versioning tool based on CVS is also integrated in the CRG editor, thus, enabling the establishment of a rule repository.

[1]http://www.aristaflow.com

10.2. Compliance rule graph execution engine

The *compliance rule graph execution engine* (CRG engine in brief) implements the CRG operational semantics. Specifically, the execution and marking rules for processing EX events and pruning rules (cf. Section 7.5.1) are implemented in the CRG engine. Similar to the CRG editor, the CRG engine was implemented in Java based on the Eclipse framework. Together with the CRG editor, the CRG engine was used in the case studies that will be described in Chapter 11.

10.2.1. Process design time

As described in Section 8.1.1, a process model can be verified against imposed CRGs by exploring the model. Compliance verification can be conducted on-the-fly when exploring the process model. However, we opted for implementing the exploration of the process model and the application of the verification algorithms separately in order to enable the direct application of the algorithms described in Section 8.2. For that purpose, we implemented a component for transforming a process model into a process event graph (PEG), an automaton-like structure whose nodes are associated with execution events of the process (cf. Section 5.2.6). Specifically, we utilized Algorithm 4 for ADEPT [Rei00] process models (i.e., WSM nets) described in Section 9.1. The implementation further incorporates node state, event type, and data state abstraction[2]. These abstraction strategies are described in Section 9.2. The PEG obtained from an ADEPT process model can be visualized in the tool as shown in Fig. 10.5.

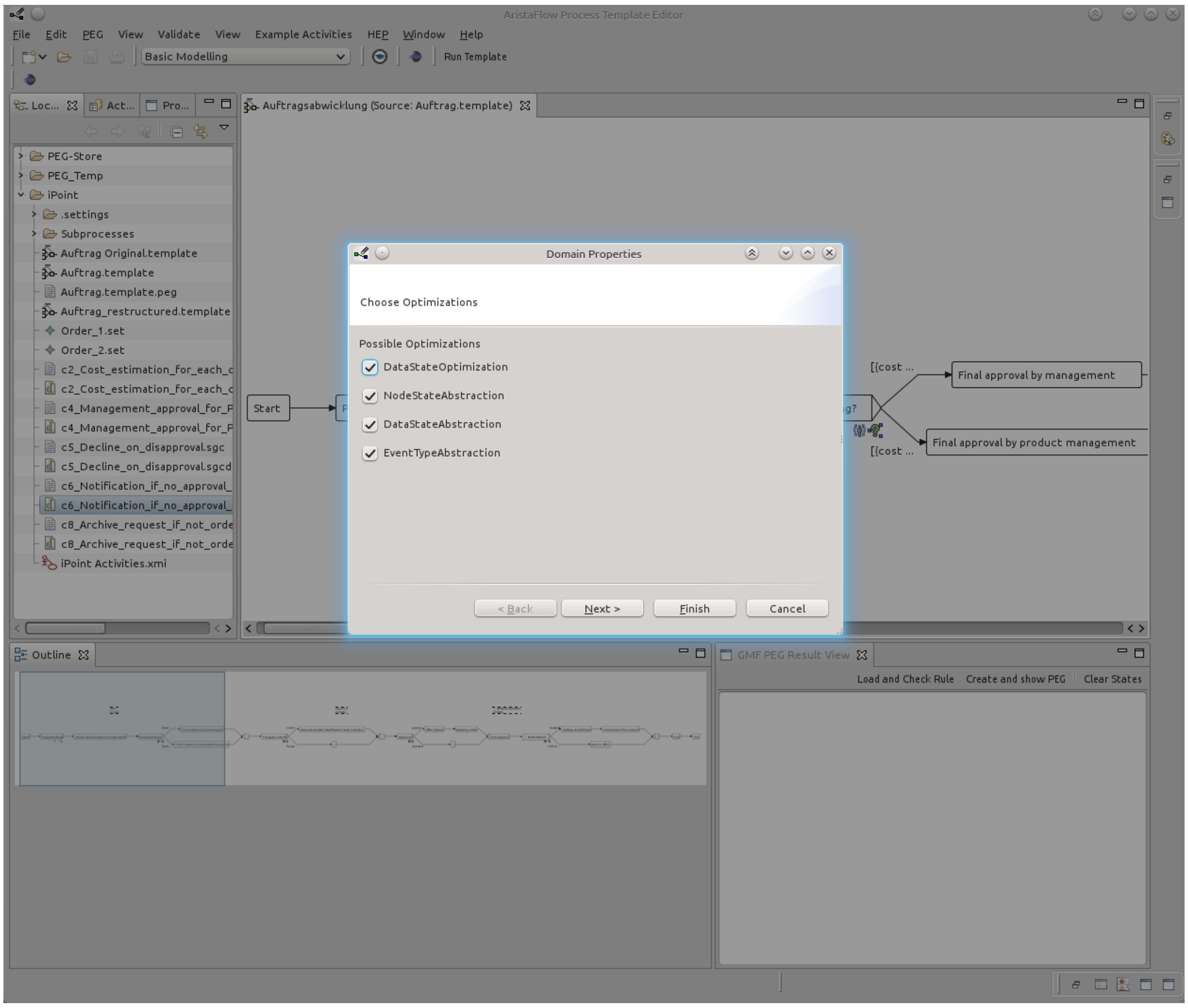
Algorithm 6 and Algorithm 7 introduced in Section 8.2.1 can be applied for annotating a PEG capturing the process' behavior with reachable compliance states. As described in Section 8.2.1, the compliance states are represented by MarkStructures that can be interpreted as shown in Section 7.4.2 and Section 8.2.2. MarkStructures can be visualized in the tool as shown in Fig. 10.6. Specifically, nodes that constitute the source of a violation in VIOLATED MarkStructures are highlighted in order to facilitate root-cause analysis. While the tracking of compliance states throughout a PEG as proposed in Section 8.2.2.2 has not been implemented yet in the proof-of-concept tool, this can be done in future extensions. This would leverage the practical application of the tool as it would enable advanced support for root-cause analysis. As this implementation is integrated into the PrMS AristaFlow BPM Suite, the process designer is relieved of switching between different tools for process modeling and compliance verification.

10.2.2. Process runtime

In order to test runtime monitoring, we implemented a component that applies the CRG engine to an execution trace. The component enables the creation of an execution trace for testing. Specifically, a graphical user interface is provided in order to create, edit, and browse the events of the trace as shown in Fig. 10.7.

For an execution trace, one can simulate the execution of a process by stepping through the trace event by event. The runtime monitoring component invokes the CRG engine for each event processed. The resulting compliance state in each step is represented by a set of MarkStructures

[2]Note that data domain abstraction is implemented in another tool that is described in Section 10.3.3.


AristaFlow Process Template Editor
File Edit PEG View Validate View Example Activities HEP Window Help
Basic Modelling
Run Template
Auftragsabwicklung (Source: Auftrag.template)
PEG-Store
PEG_Temp
iPoint
.settings
Subprocesses
Auftrag Original.template
Auftrag.template
Auftrag.template.peg
Auftrag_restructured.template
Order_1.set
Order_2.set
iPoint Activities.xmi
Start
Final approval by management
Final approval by product management
Outline
GMF PEG Result View
Load and Check Rule
Create and show PEG
Clear States
< Back
Next >
Finish
Cancel

Domain Properties

Choose Optimizations

Possible Optimizations

- ✓ DataStateOptimization
- ✓ NodeStateAbstraction
- ✓ DataStateAbstraction
- ✓ EventTypeAbstraction

Figure 10.4.: Configuration of abstraction strategies

as described in Section 8.4.1. These MarkStructures are illustrated in the GUI. In violated MarkStructures, nodes that constitute the source of the violation are highlighted in order to facilitate compliance diagnosis as shown in Fig. 10.6. For a commercial implementation, a more sophisticated GUI exploiting the information provided by the CRG execution engine for root-cause analysis can be designed. As this component is integrated directly in the CRG editor, it may also be used to test CRGs in order to ensure that they exhibit the semantics intended by the rule designer.

Altogether, the CRG editor and the CRG engine with its additional components for exploring process models and for investigating process executions demonstrate the technical feasibility of the compliance checking framework proposed in this thesis. The integration of these tools with a commercial PrMS shows that our framework is suitable for enhancing PrMSs with compliance checking functionality.

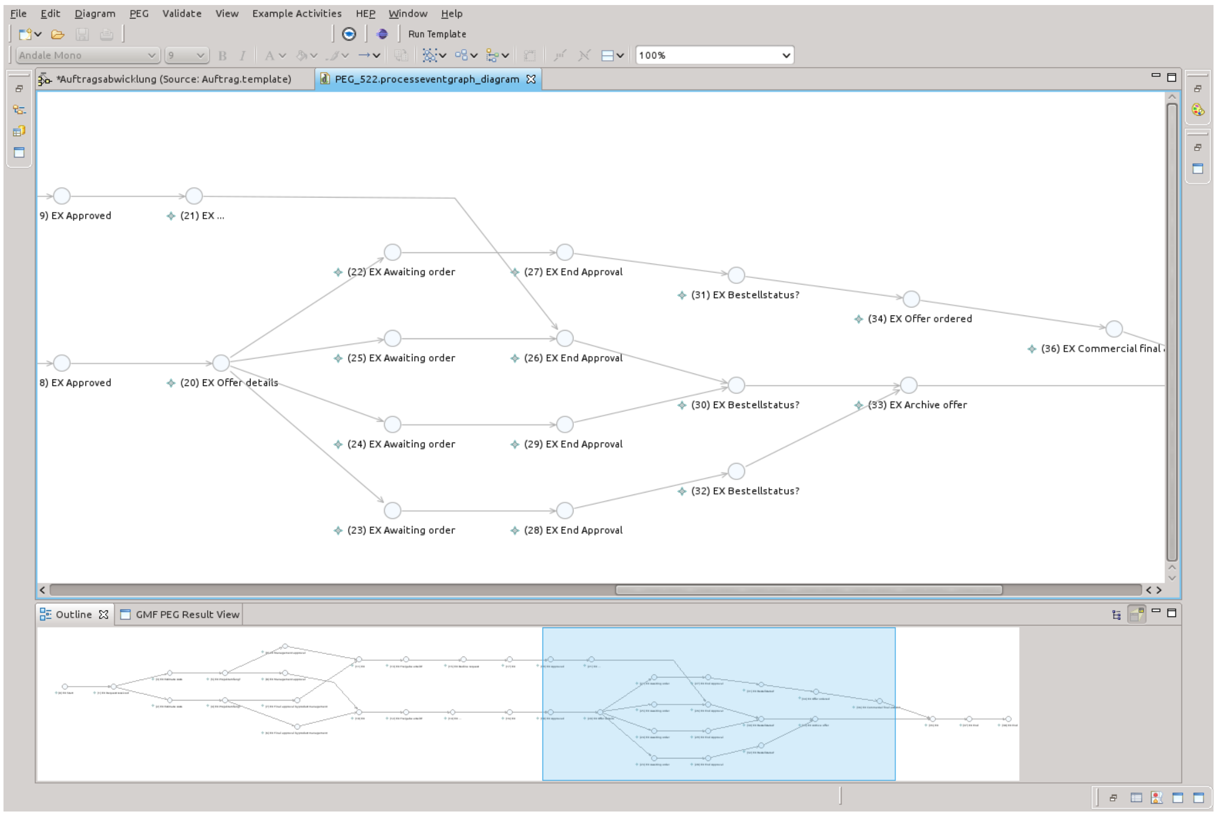


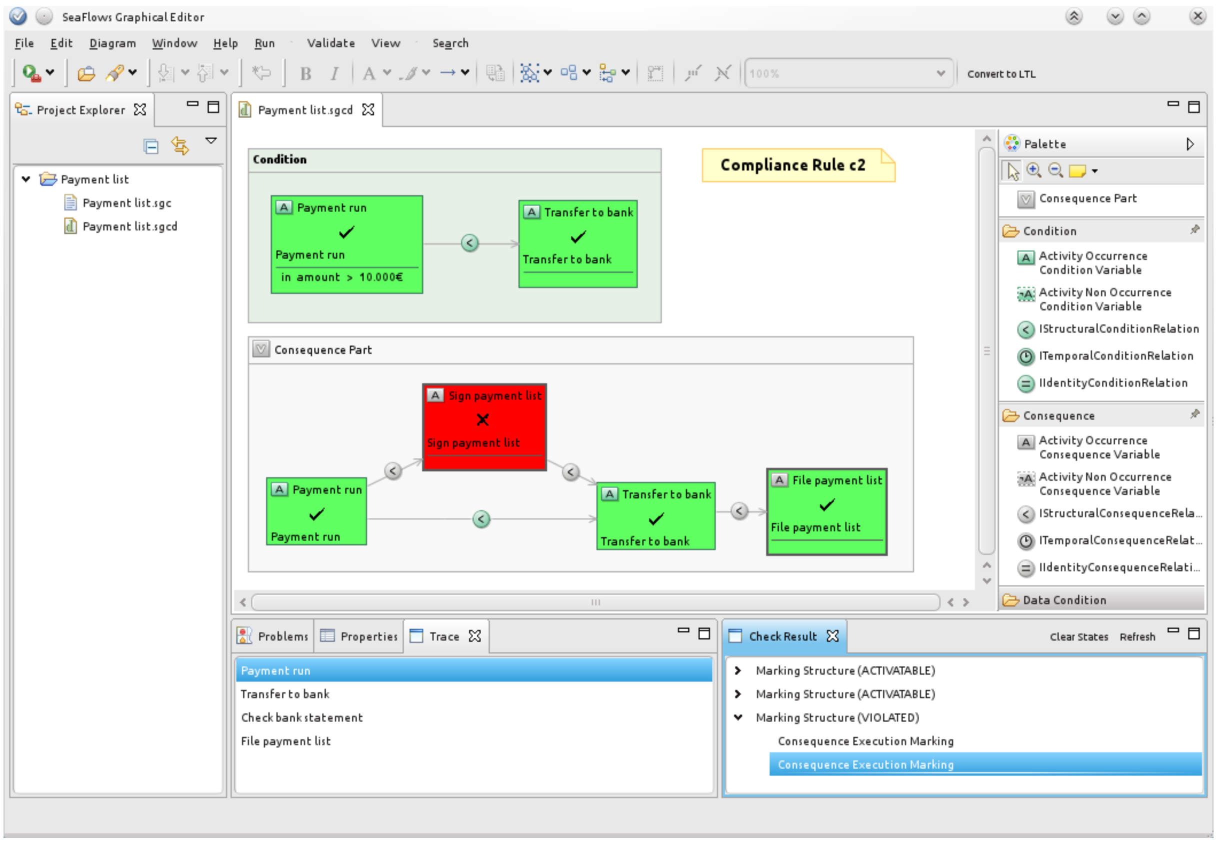
Figure 10.5.: A PEG obtained from exploring a process model

10.3. Further tools

In the course of our research, we experimented with different approaches, which resulted in tools besides the ones implementing the main concepts described in the thesis. In the following, we would like to give a brief overview on these tools and underlying ideas.

10.3.1. Visualization component for compliance violations

The visualization component for compliance violations has already been introduced in Section 9.3.1 when describing ideas on the visualization of violations within process models. This component was developed in the context of a master thesis [Mer10] conducted in the SeaFlows project. To provide an overview on the parts of the model affected by a violation, paths of the process model in which a compliance rule is violated / satisfied are highlighted as indicated in Fig. 10.8. In addition to highlighting the paths of the process model, the approach further tries to support process designer in resolving noncompliance by integrating the imposed CRG into the process model visualization. In Fig. 10.8, the consequence part of the CRG is visualized when clicking on the activity that belongs to the antecedent CRG in the process. This facilitates the deduction of what is necessary to satisfy a compliance rule. In order to assist in resolving the violation, the system highlights the relevant activities contained in the process model when



```
Payment run
Transfer to bank
Check bank statement
File payment list
```

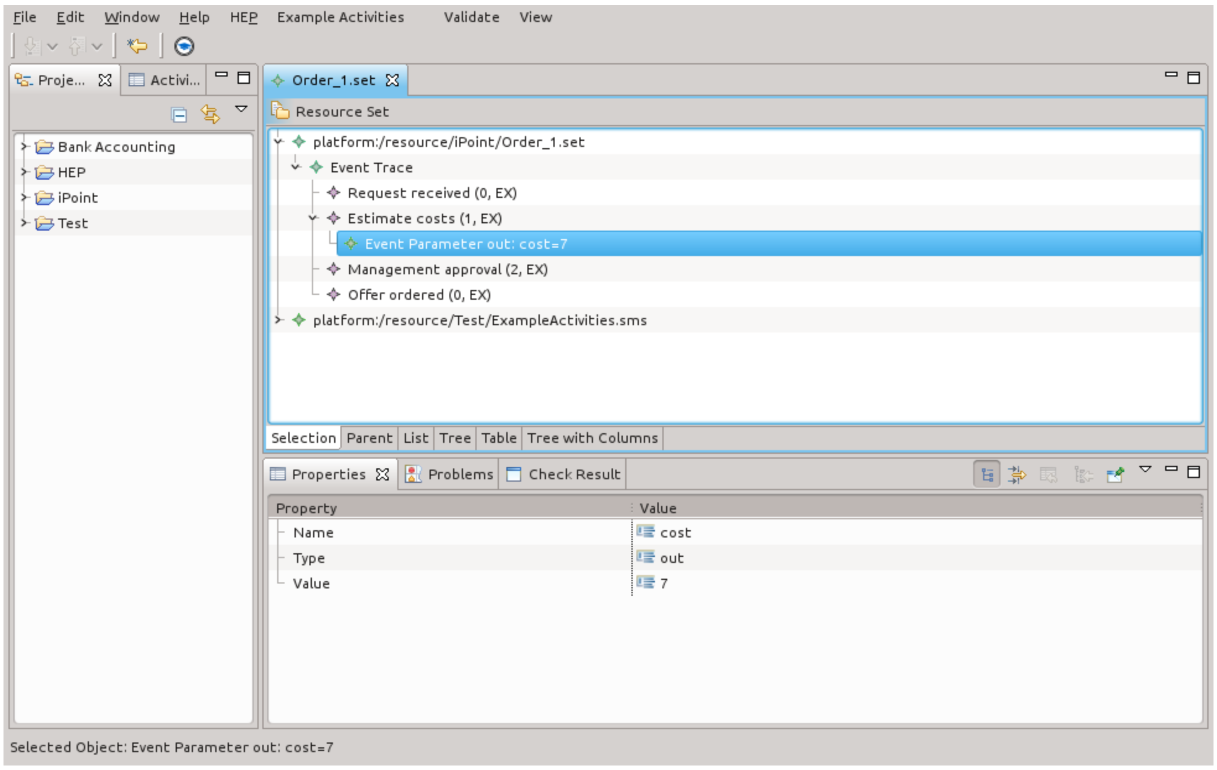
```
> Marking Structure (ACTIVATABLE)
> Marking Structure (ACTIVATABLE)
v Marking Structure (VIOLATED)
    Consequence Execution Marking
    Consequence Execution Marking
```

Figure 10.6.: Visualization of a VIOLATED MARKSTRUCTURE

pointing to the nodes of the consequence part. The visualization component further integrates interesting features specifically for dealing with complex process models. For example, it implements concepts for automatically reducing the process model in order to show all relevant activities at the same time on screen or for folding and unfolding branches of the process model to obtain more compact representations. It is integrated with the commercial PRMS AristaFlow BPM Suite, from which it obtains the process model. For further screenshots and details on the concept, the reader is referred to Section 9.3.1.

10.3.2. Structural compliance checker

The *structural compliance checker* implements an approach that aims at verifying process models against imposed rules by checking the process structure (instead of behavioral exploration). We introduced structural compliance checking in [LRD06], however, so far we only addressed basic exclusion and dependency constraints. We have further extended our approach in order to provide support for a broader range of compliance rules. Based on the assumption of unique labels (i.e., unique occurrences of activities within a process model), we have developed a structural compliance checking approach for a subset of CRGs. This approach is designed to support loop-free process models and abstracts from data conditions.



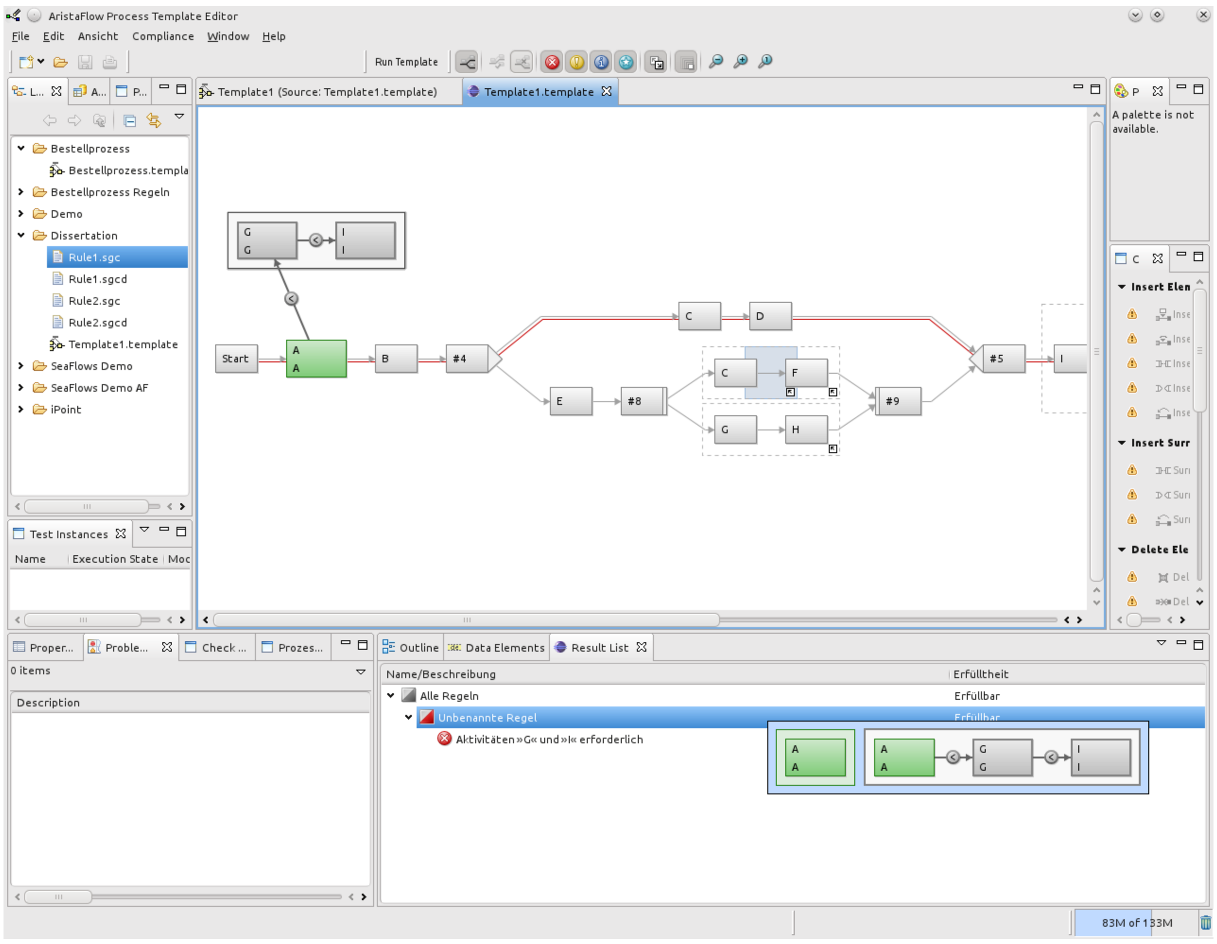
```
platform:/resource/iPoint/Order_1.set
  Event Trace
    Request received (0, EX)
    Estimate costs (1, EX)
      Event Parameter out: cost=7
    Management approval (2, EX)
    Offer ordered (0, EX)
platform:/resource/Test/ExampleActivities.sms
```

Property	Value
Name	cost
Type	out
Value	7

Selected Object: Event Parameter out: cost=7

Figure 10.7.: The execution trace editor for testing compliance monitoring

The structural compliance checking approach is conducted in three steps. In step 1, for each CRG, a set of structural criteria to be checked over the process model is automatically determined. These criteria can be considered queries on the relations of nodes within the process model (i.e., node relations) that are relevant to the compliance rule. We define five structural criteria consisting of containment, occurrence, and precedence relations. In step 2, the process model is checked for compliance with the derived structural criteria. Thus, we can precisely identify the structural criteria causing noncompliance. In case a compliance violation is detected, these structural criteria will be collected in step 3 and will be used for error diagnosis and for the generation of textual feedback as indicated in Fig. 10.9. By showing which structural criteria are not satisfied by the process model, the system can help to resolve noncompliance. By exploiting certain properties of the process description language such as block-structuring, the structural criteria can be efficiently evaluated (cf. [RD98]). Adopting the paradigm of dynamic programming, we cache node relations already queried to enable faster evaluation when the same relations are queried a second time. The verification tool is integrated directly into the process modeling environment of AristaFlow BPM Suite. This relieves the process designer of switching between tools in order to apply compliance checks. For details on the tool implementation and on the underlying concepts, the reader is referred to [LKRM$^+$10].

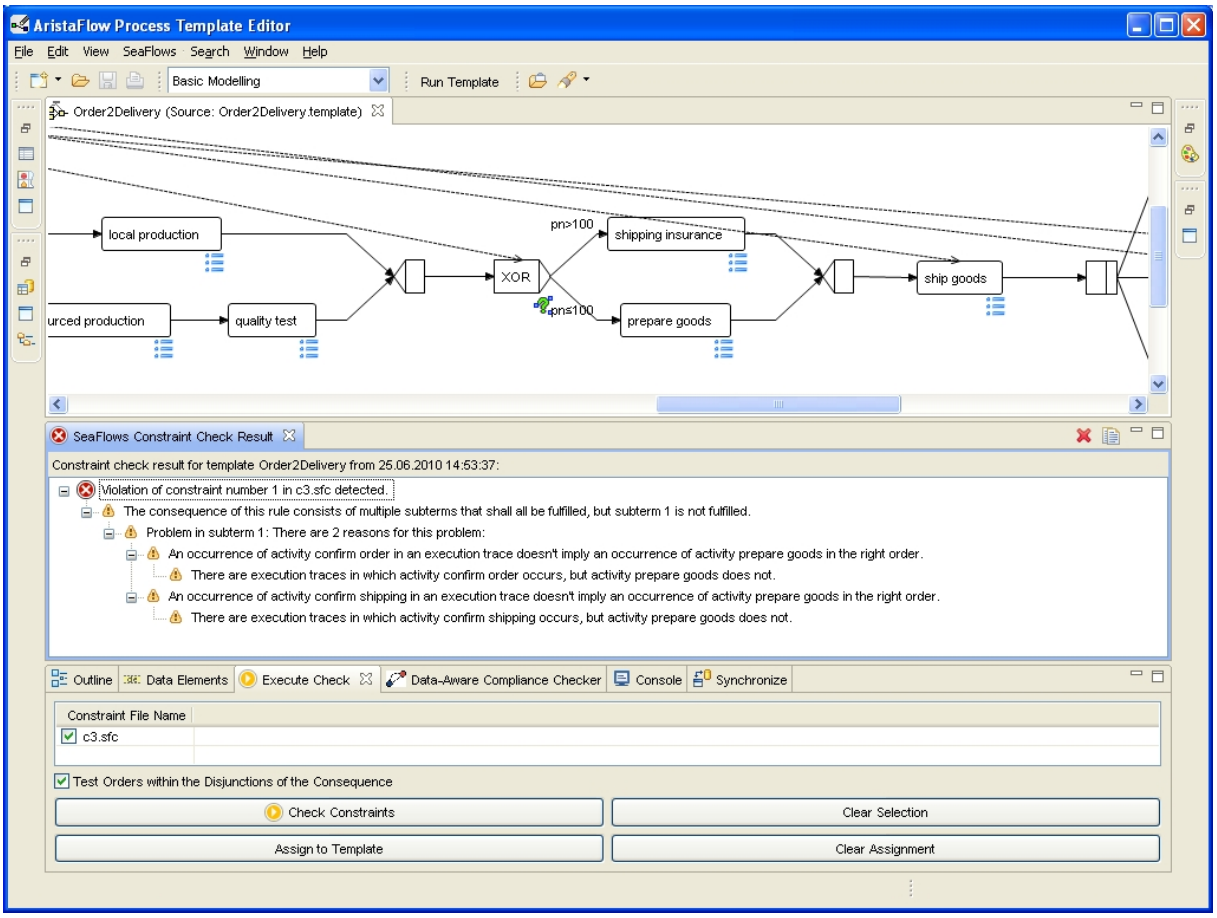


Name/Beschreibung	Erfülltheit
Alle Regeln	Erfüllbar
Unbenannte Regel	Erfüllbar
Aktivitäten »G« und »I« erforderlich	

Figure 10.8.: Visualization of compliance violations in a process model

10.3.3. Data-aware compliance checker

The *data-aware compliance checker* was introduced in [LKRM$^+$10]. It was implemented to showcase the feasibility of the abstraction approach developed in a master thesis [Knu08] conducted in the SeaFlows project. The abstraction approach, described in Section 9.2.6 and in [KLRM$^+$10], aims at reducing the state space to be explored for compliance checking. Given a process model to be verified, the data-aware compliance checker first performs automatic data domain abstraction as described in Section 9.2.6. The basic idea is to reduce the domain of data objects, thus, merging states that differ in data allocations but are nevertheless indistinguishable to the rule to be checked. The abstracted data domain is computed by exploiting the data conditions contained in the process model and in the compliance rules (in this implementation, we used LTL rules). Then, the *data-aware compliance checker* transforms the abstract process model into a state space representation for the actual compliance checking applying node state abstraction as described in Section 9.2.2. The resulting model can be used as input for compliance checking, for example using the approach proposed in this thesis. In the implementation of the data-aware compliance checker, we employed the model checker SAL [BGL$^+$00]. The counterexample



Constraint check result for template Order2Delivery from 25.06.2010 14:53:37:

Violation of constraint number 1 in c3.sfc detected.

The consequence of this rule consists of multiple subterms that shall all be fulfilled, but subterm 1 is not fulfilled.

Problem in subterm 1: There are 2 reasons for this problem:

An occurrence of activity confirm order in an execution trace doesn't imply an occurrence of activity prepare goods in the right order.

There are execution traces in which activity confirm order occurs, but activity prepare goods does not.

An occurrence of activity confirm shipping in an execution trace doesn't imply an occurrence of activity prepare goods in the right order.

There are execution traces in which activity confirm shipping occurs, but activity prepare goods does not.

[✓] c3.sfc

[✓] Test Orders within the Disjunctions of the Consequence

Figure 10.9.: Textual feedback provided by the structural compliance checker

obtained from SAL is backtransformed in order to provide feedback on detected noncompliance. Similar to the other tools, the *data-aware compliance checker* is integrated with the PrMS AristaFlow BPM Suite.

10.4. Summary and outlook

In this chapter, we presented the components of SeaFlows Toolset, our proof-of-concept implementation. Specifically, we introduced the CRG editor and the CRG engine that addresses design and runtime compliance verification. In addition, we described further tools that resulted from the iterative research procedure employed in our project. Most of our tools are integrated with the commercial PrMS AristaFlow BPM Suite. However, the toolset can also be adapted for integration with other PrMSs. Due to the general event format utilized by the implementation, it can be integrated with other tools for process analysis such as the process mining framework ProM [VBDA10].

11 Practical evaluation

With the prototypical implementation of our concepts, we demonstrated its technical feasibility. In order to assess the practicability of our approach, further evaluation is required. In this chapter, we describe our efforts to evaluate the compliance checking framework proposed in this thesis. In Section 11.1, we utilize CRGs to model compliance rule patterns that we often encountered in meta-analyses. We further applied the SeaFlows Toolset to real world data. In particular, we utilized the compliance checking framework developed in this thesis to analyze processes from the higher education domain. The results from this study are described in Section 11.2. One part of the results was also published in [LIMRM12]. Finally, we also analyzed a process of an IT company targeting suppliers in the automotive domain. The results from that case study are described in Section 11.3. Section 11.4 finally summarizes the lessons learned.

11.1. Pattern-based evaluation

In meta-analyses, we observed that many approaches in literature came up with a set of compliance rule patterns. For example, patterns were proposed by Awad et al. [ADW08, AWW09], Namiri et al. [NS08, Nam08], and Giblin [GMP06]. Specifically, these patterns are based on the property specification patterns collected by Dwyer and Corbett [DAC99] (cf. Section 2.1.1.1). In [LRMD10], we showed how some these patterns can be expressed using CRGs. In the following, we extend our considerations to the complete set of property specification patterns. In addition, we further show how scopes, a useful concept of the property specification patterns, can be expressed very naturally using CRGs.

As described in Section 2.1.1.1, the property specification patterns consist of *occurrence* and *order patterns*, which can be configured with *scopes* defining in which region the constraints apply.

11.1.1. Occurrence patterns

Recall the interpretation of the patterns for process activities (as described in Section 2.1.1.1)[1]. *Absence* can be expressed using ConsAbs nodes. *Existence* requires that a region contains a certain activity. This can be expressed using ConsOcc nodes. *Bounded existence* defines that a region contains at most a specified number of a certain activity. For example, the constraint that there are at most two executions of activity A in a process execution can be captured by modeling that if there are two executions of A, there shall be no further execution of A. This can be modeled using two AnteOcc and a ConsAbs node. An interpretation of this pattern for a sequence of A is illustrated in Fig. 11.1. A different interpretation that does not refer to sequential occurrences of A can be expressed using the same nodes but with DIFF edges instead of ORDER edges (i.e., following "if there are already two executions of A in a process execution, no further execution of A is allowed"). Alternatively, one may also define the pattern identifying a violation of a bounded existence constraint using CRGs (i.e., the negated pattern). If this pattern is detected in a process execution, the constraint is violated.

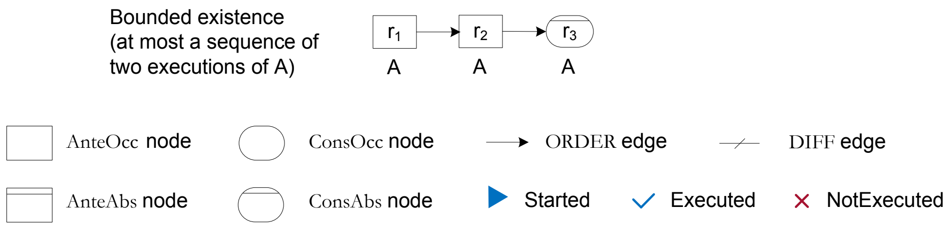


Figure 11.1.: An interpretation of the *bounded existence* pattern modeled as CRG

11.1.2. Order patterns

Fig. 11.2 depicts CRGs expressing the *precedence*, *response*, *chain precedence*, and *chain response* patterns. *Precedence* requires the occurrence of a certain activity prior to the occurrence of another activity (i.e., the first activity is premise to the later activity). *Response* requires the occurrence of a certain activity in response to the occurrence of a prior activity (i.e., stimulus-response). Different variations of precedence chains (i.e., 1 cause-2 effect and 2 cause-1 effect) and response chains (i.e., 1 stimulus-2 response and 2 stimulus-1 response) are depicted in Fig. 11.2.

It is notable that these patterns, for example, the 1 stimulus-2 response chain, can be extended easily. Imagine, for example, we would want to further modify the 1 stimulus-2 response chain such that the more specific response chain S and T without R being executed between S and T is required after P. This can be easily integrated by adding an absence constraint (i.e., a ConsAbs node) between the respective nodes in the CRG.

[1]Note that the *universality* pattern is not applicable to our event-based view of process executions.

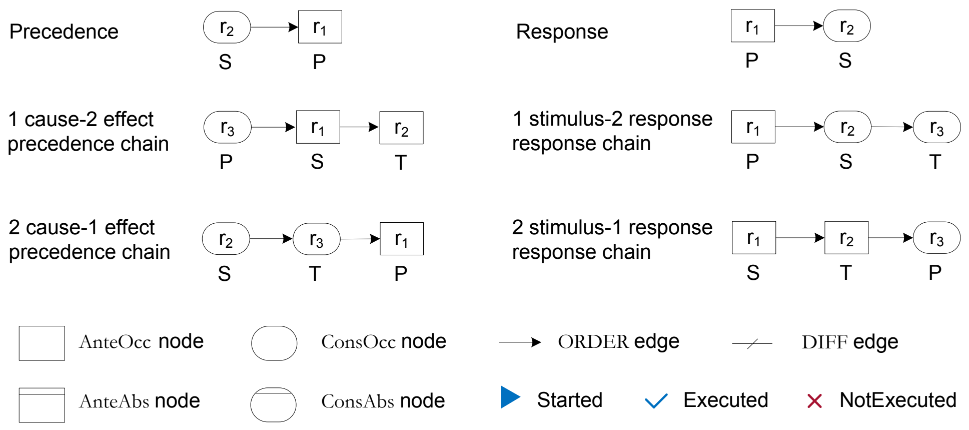


Figure 11.2.: Order patterns modeled as CRGs

11.1.3. Scopes

Five scopes were introduced by Dwyer and Corbett as shown in Fig. 2.1 in Section 2.1.1.1: *global*, *after R*, *before R*, *between Q and R*, and *after Q until R*. As these scopes define the region in which a constraint applies, they are naturally associated with a compliance rule's antecedent part. Hence, it is apparent that AnteOcc nodes can be utilized to model scopes. Clearly, *global* scope does not require any definition. *After R* and *before R* can be captured by using an AnteOcc node for R and relating the compliance rule to this node. In a similar manner, *between Q and R* can be modeled. Fig. 11.3 depicts three interpretations of the *1 stimulus-2 response response chain* in combination with *between Q and R* scope. While the first variant requires both the stimulus and the response to occur between Q and R, the second variant does not require the response to occur before R. The CRG defined in the first and the second variant applies to each sequence with Q followed by R. However, one may want to be more precise about the scope of the compliance rule. In the third variant, for example, the scope is refined such that the response chain has to apply between Q and the first next occurrence of R. In that manner, further refinements can be realized by incrementally adding or removing primitives of the CRG language.

Expressing *after Q until R* scope is not as straightforward as CRGs do not know the weak until operator, which is available in LTL. However, the semantics can be captured by defining two CRGs, one for the case that R occurs after Q and one for the case that R does not occur after Q.

11.1.4. Discussion

Predefining patterns of frequently occurring compliance rules is a feasible approach in general and can facilitate compliance rule modeling. Such patterns can be defined using CRGs. We believe that the definition of compliance rules in a pattern matching manner as with CRGs is more convenient than by navigating through a trace as, for example, with LTL. Existing patterns, such as *precedence chain*, can easily be extended, for example, by adding additional absence constraints between the occurrences of the involved activity executions. This is not possible

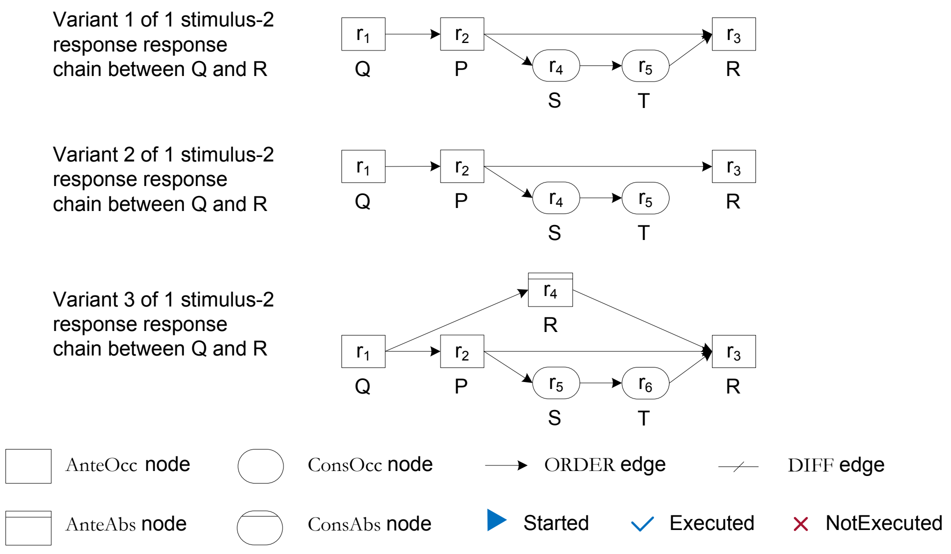

Variant 1 of 1 stimulus-2 response response chain between Q and R
r1
Q
r2
P
r3
R
r4
S
r5
T
Variant 2 of 1 stimulus-2 response response chain between Q and R
r1
Q
r2
P
r3
R
r4
S
r5
T
Variant 3 of 1 stimulus-2 response response chain between Q and R
r4
R
r1
Q
r2
P
r3
R
r5
S
r6
T
AnteOcc node
ConsOcc node
ORDER edge
DIFF edge
AnteAbs node
ConsAbs node
Started
Executed
NotExecuted

Figure 11.3.: Three interpretations of the *1 stimulus-2 response response chain* in combination with *between Q and R* scope

when just combining property specification patterns (and their mappings to, for example, LTL) using the logical conjunction. To leverage the definition of scopes, a semantic level can be introduced to the CRG language enabling the semantic distinction between the scope defining nodes and normal antecedent nodes.

11.2. Higher education processes

In the HEP project[2] conducted at the University of Vienna, higher education processes were investigated. The HEP data set consists of different process types reflecting different courses. For this case study, we selected one course, which took place in three consecutive years. Collecting data for this course yielded log data of 330 instances (one instance per student) with 18511 events. A log of a process instance corresponds to an execution trace reflecting the events occurring in the process instance.

In this case study, we employed the compliance checking framework proposed in this thesis to analyze process logs with respect to compliance with imposed rules. This provided the basis for *semantic log purging*, a preprocessing step for process mining. In the purging process, logs that violate certain rules were removed from the log set. The remaining logs were given as input to process mining to detect reference process models based on the course process instances that comply with imposed rules. The process models obtained from process mining with and without semantic log purging were compared to a reference process model that was created with the help of the instructors. The research methodology is illustrated in Fig. 11.4. The objective of the overall study was to find out whether semantic log purging can help to improve certain quality

[2]www.wst.univie.ac.at/communities/hep/

aspects of mined process models. The methodology and the findings of that experiment are described in our paper [LRMGD12].

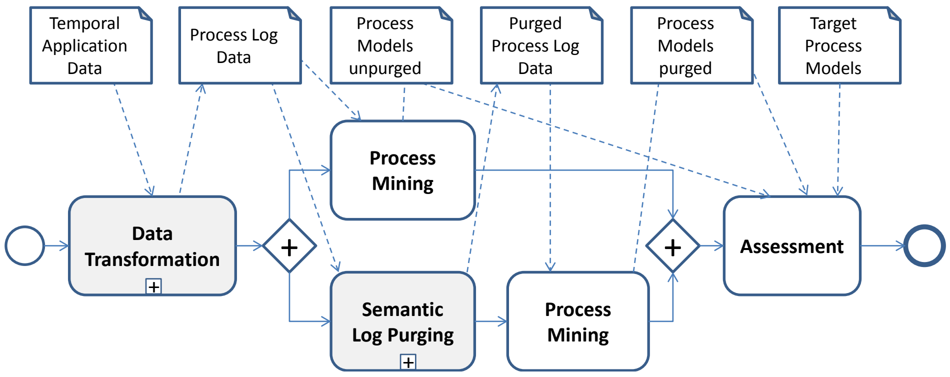


Figure 11.4.: Research methodology in the HEP project (taken from [LRMKD11])

The process of semantic log purging using SeaFlows Toolset is illustrated in Fig. 11.5. First, the log data is imported to our tool, which, in turn, creates an event model of the events occurring in the traces. Based on this event model, the constraints imposed on the course process are defined as CRG composites. Then, the logs are verified against the CRG composites using SeaFlows Toolset. Logs that violate at least one CRG composite are purged from the log set. Thus, a log set remains that contains only logs complying with imposed constraints.

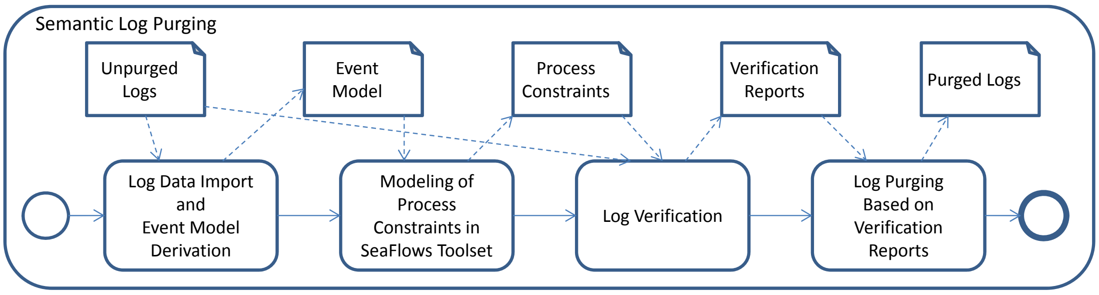

Semantic Log Purging
Unpurged Logs
Event Model
Process Constraints
Verification Reports
Purged Logs
Log Data Import and Event Model Derivation
Modeling of Process Constraints in SeaFlows Toolset
Log Verification
Log Purging Based on Verification Reports

Figure 11.5.: The semantic log purging process (taken from [LRMKD11])

11.2.1. Compliance rules

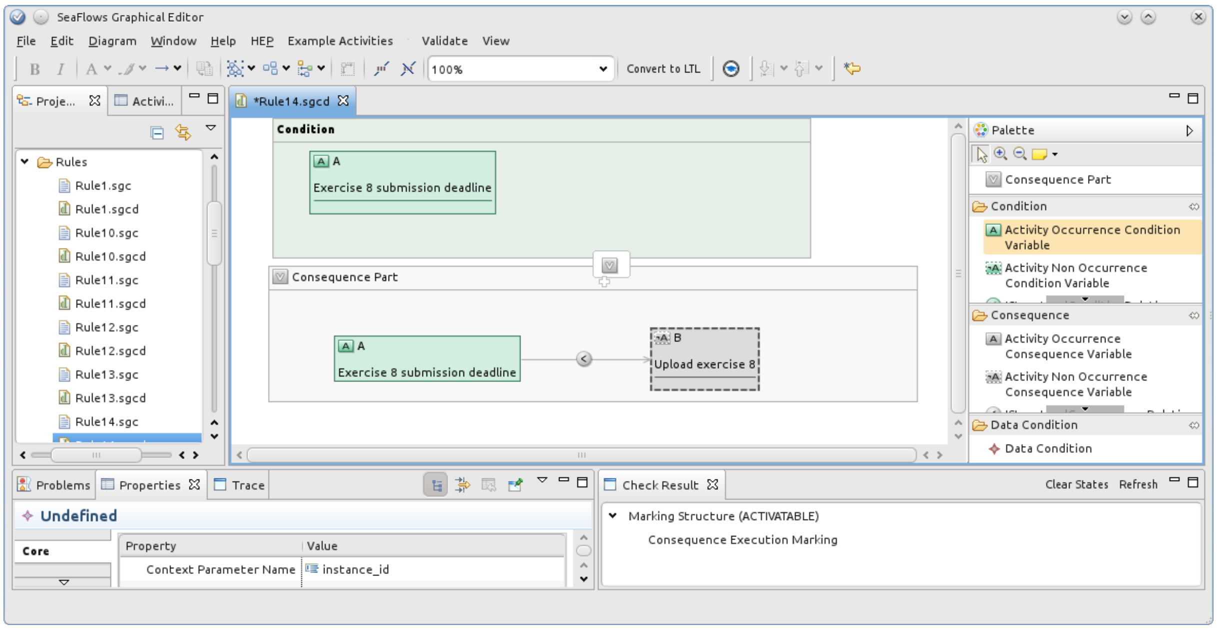
In the scope of this thesis, the log purging part exploiting the compliance checking framework proposed is most interesting. Together with the instructors, we collected a set of process constraints (i.e., compliance rules), that served as the basis for semantic log purging. These constraints define expected behavior in the course process. In the course process, different milestones and exercises have to be passed by students. Certain deadlines for each milestone and exercises were defined by the course instructor. All actions with respect to the exercises and milestones, i.e., the submission rounds of milestones or exercises by students and the evaluation of them by supervisors, were logged by the online learning environment. Thus, this data could be extracted and transformed into event logs. The events of the log data, therefore, do not

	Constraint	Enforcement level
c_1	For each milestone, no upload must take place after the corresponding milestone deadline.	high
c_2	For each exercise, no upload must take place after the corresponding exercise deadline.	high
c_3	For each uploaded milestone, the instructor gives feedback.	low

Table 11.1.: Constraints imposed on the course process

correspond to activity executions but are rather instantaneous. While deadlines mark the due date of milestone and exercise submissions, the blended learning environment does not forbid belated submissions. Therefore, a major requirement was to detect deadline violations. Some constraint examples are summarized in Table 11.1 [LRMGD12].

Clearly, these constraints can be expressed by means of CRGs. As each milestone and exercise deadline constitute a unique event type, CRGs were defined for each such deadline event. Fig. 11.6 shows a constraint defined in the SeaFlows CRG editor.


SeaFlows Graphical Editor
File Edit Diagram Window Help HEP Example Activities Validate View
100%
Convert to LTL
Proje... Activi...
*Rule14.sgcd
Rules
Rule1.sgc
Rule1.sgcd
Rule10.sgc
Rule10.sgcd
Rule11.sgc
Rule11.sgcd
Rule12.sgc
Rule12.sgcd
Rule13.sgc
Rule13.sgcd
Rule14.sgc
Condition
A
Exercise 8 submission deadline
Consequence Part
A
Exercise 8 submission deadline
B
Upload exercise 8
Palette
Consequence Part
Condition
Activity Occurrence Condition Variable
Activity Non Occurrence Condition Variable
Consequence
Activity Occurrence Consequence Variable
Activity Non Occurrence Consequence Variable
Data Condition
Data Condition
Problems Properties Trace
Undefined
Core
Check Result
Clear States Refresh

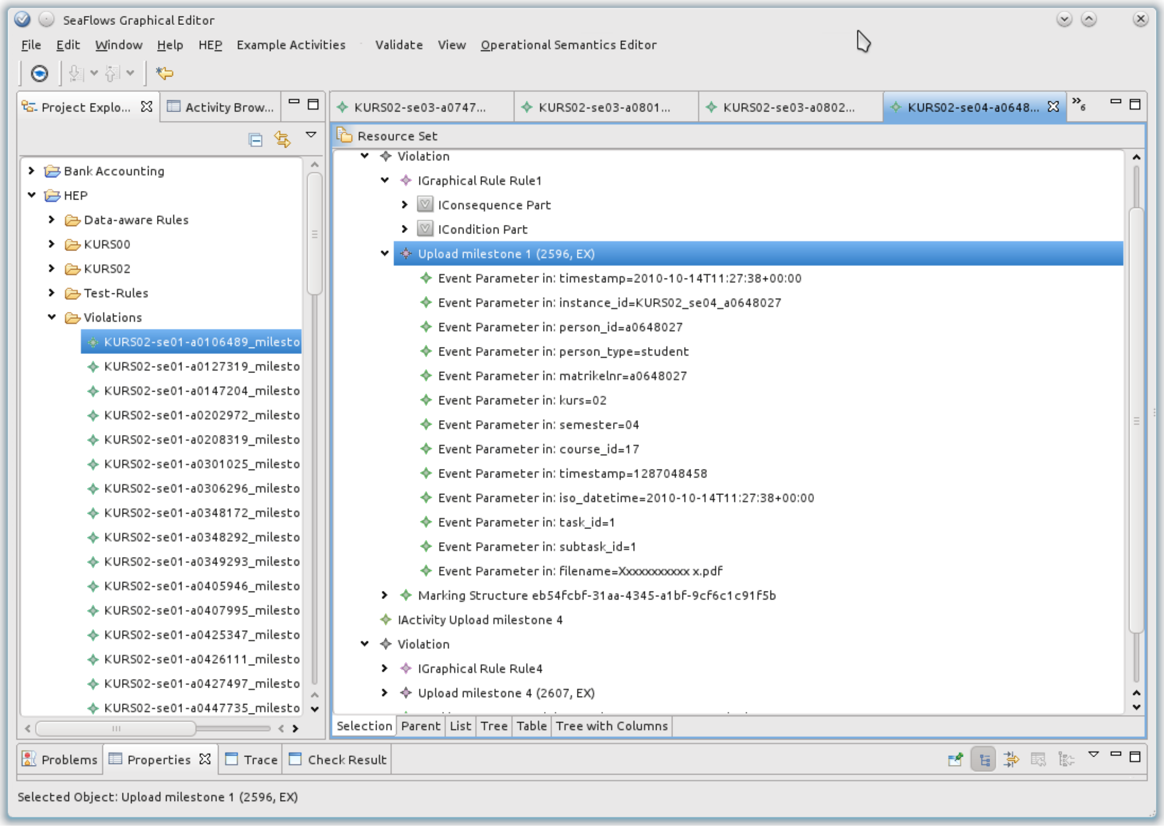
Property	Value
Context Parameter Name	instance_id

Marking Structure (ACTIVATABLE)
Consequence Execution Marking

Figure 11.6.: A constraint of the course process modeled as CRG

11.2.2. Compliance analysis

The log set containing the 330 instances were checked for compliance with the imposed rules using our compliance checking framework. To facilitate the a posteriori analysis, we equipped the SeaFlows Toolset with batch processing features for processing multiple traces in one batch activity. For each log (i.e., each process instance) violating at least one compliance rule, a report file is generated (cf. Fig. 11.7). The file contains information on violated compliance rule activations and the events associated with them.


SeaFlows Graphical Editor
File Edit Window Help HEP Example Activities Validate View Operational Semantics Editor
Project Explo...
Activity Brow...
KURS02-se03-a0747...
KURS02-se03-a0801...
KURS02-se03-a0802...
KURS02-se04-a0648...
Resource Set
Selection Parent List Tree Table Tree with Columns
Problems Properties Trace Check Result

```
Bank Accounting
HEP
  Data-aware Rules
  KURS00
  KURS02
  Test-Rules
  Violations
    KURS02-se01-a0106489_milesto
    KURS02-se01-a0127319_milesto
    KURS02-se01-a0147204_milesto
    KURS02-se01-a0202972_milesto
    KURS02-se01-a0208319_milesto
    KURS02-se01-a0301025_milesto
    KURS02-se01-a0306296_milesto
    KURS02-se01-a0348172_milesto
    KURS02-se01-a0348292_milesto
    KURS02-se01-a0349293_milesto
    KURS02-se01-a0405946_milesto
    KURS02-se01-a0407995_milesto
    KURS02-se01-a0425347_milesto
    KURS02-se01-a0426111_milesto
    KURS02-se01-a0427497_milesto
    KURS02-se01-a0447735_milesto
```

```
Violation
  IGraphical Rule Rule1
    IConsequence Part
    ICondition Part
  Upload milestone 1 (2596, EX)
    Event Parameter in: timestamp=2010-10-14T11:27:38+00:00
    Event Parameter in: instance_id=KURS02_se04_a0648027
    Event Parameter in: person_id=a0648027
    Event Parameter in: person_type=student
    Event Parameter in: matrikelnr=a0648027
    Event Parameter in: kurs=02
    Event Parameter in: semester=04
    Event Parameter in: course_id=17
    Event Parameter in: timestamp=1287048458
    Event Parameter in: iso_datetime=2010-10-14T11:27:38+00:00
    Event Parameter in: task_id=1
    Event Parameter in: subtask_id=1
    Event Parameter in: filename=Xxxxxxxxxxx x.pdf
  Marking Structure eb54fcbf-31aa-4345-a1bf-9cf6c1c91f5b
IActivity Upload milestone 4
Violation
  IGraphical Rule Rule4
  Upload milestone 4 (2607, EX)
```

Selected Object: Upload milestone 1 (2596, EX)

Figure 11.7.: Report files for each violated process instance

The MarkStructures of violated rule instances can be shown to the user and details of violated MarkStructures can be browsed as shown in Fig. 11.7. Based on these MarkStructures, explanations for violations can be generated as described in Section 7.4.2.

11.2.3. Lessons learned

This case study investigated the application of our compliance checking framework to a posteriori analysis. However, the logs collected can also be used to simulate compliance monitoring using our framework (cf. Section 10.2.2). While the constraints imposed on the course process have a rather simple structure, they nevertheless led to interesting insights. Specifically, this study showed that the particular CRG composites that capture the semantics of certain imposed constraints largely depend on the event model.

Firstly, in this specific case, each exercise and milestone deadline constitute a unique event type. Therefore, the constraint that no submissions are allowed after the corresponding deadline, for example, was defined for each deadline event type. If, however, the deadlines for different milestones and exercises would be represented by the same event type, the process constraint could be expressed through a single CRG using the milestone / exercise as context data object. In this case, each CRG would refer to an artifact (e.g., a milestone or an exercise). Thus, to support such artifact-oriented constraints [RKG06, Mül09, Loh11, KR11b] properly, it must be possible to define constraints for (data) objects. While data relations are not supported in CRGs, our prototype implementation already enables data relations that can be utilized for supporting data-oriented constraints. In particular, a modeled CRG (or a CRG composite) can be defined for a data object. When executing the CRG over an execution trace, rule activations will be detected based on the data object information.

Secondly, in the log data obtained from the online learning environment, deadlines were represented by specific events. Thus, it was possible to use the deadline events for CRG definition. If, however, deadlines would not be represented by specific events, a notion of time would have been necessary in order to capture the process constraints. This is not supported in our approach so far and should be investigated in future research.

11.3. Project process in an IT company

This study was conducted in cooperation with an IT company targeting sub-contractors in the automotive domain. The process under examination is the company's project process and describes how a project is processed from a project request to the completion of the project. It is a complex business process involving different departments of the company. It covers a sales and approval subprocess, the actual project workflow and an administrative cost calculation subprocess. The coarse model is depicted in Fig. 11.8. In the process, an incoming customer request (e.g., for an IT project or a training) is processed. The incoming request initiates different activities, such as the creation of an offer for the customer. If the customer places an order, the actual project will be conducted. After project completion, some administrative cost calculations are conducted. The business level process model contains around 100 artifacts when being modeled using an EPC-like notation.

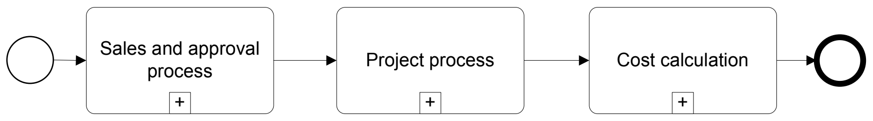


Figure 11.8.: The coarse project process modeled in BPMN

For this case study, we first discussed the project process with a practitioner and collected constraints that the process is supposed to obey. The compliance rules collected were described informally. While the major objective was to apply the SeaFlows compliance checking framework to analyze compliance with imposed rules, we nevertheless approached this study in a more general way. Thus, we tried to utilize the compliance checking framework for realizing compliance controls when applicable but also considered other kinds of compliance controls when suitable (following Section 1.1). We basically used this study to get an idea of a compliance project in practice.

In order to enable automated compliance checking using our framework, the informal compliance rules have to be translated into checkable rules on the concrete implementation of the project process. For that purpose, a closer look at the process implementation was necessary. In the following, the compliance rules we collected and their translation into checkable compliance rules are described in Section 11.3.1. Then, the application and results of the compliance analysis are described in Section 11.3.2. Section 11.3.3 summarizes the lessons learned in this case study.

11.3.1. Compliance rules

Together with the project manager, a practitioner best familiar with project processing in the company, we discussed the different stages of the project process and collected compliance rules imposed on it. The rules collected are summarized in Table 11.2. As apparent, the compliance rules are described in natural language and neither imply a specific approach for checking compliance nor indicate a specific process implementation. Most of the compliance rules collected refer to the sales and approval part of the project process involving the handling and approval of customer requests and the creation of offers for the customer.

	Constraint
c_1	If the incoming customer request comes from a new customer, a record for the new customer has to be created in the Alfresco system for managing customer data.
c_2	The cost of the project has to be roughly estimated for each incoming customer request.
c_3	For critical customers, extra effort should be calculated.
c_4	All customer requests requiring more than five working days necessitate management approval.
c_5	If management approval is necessary but approval is not provided, the request has to be declined.
c_6	The management must be notified of customer requests for which management approval is not necessary.
c_7	The request should be replied before the associated due date if a due date is provided
c_8	An offer that is not ordered, should be archived.
c_9	If an offer expires without an order being placed, the sales shall be notified.
c_{10}	If an offer is not ordered before expiry date, it should be archived or adapted.
c_{11}	The installation of the implementation should be started no later than two days after completion of internal tests.

Table 11.2.: Constraints imposed on the project process

In the second step, the rules were modeled as CRGs assuming a suitable event model in the process model reflecting the semantic concepts associated with the rules. The CRGs are depicted in Fig. 11.9. Note that c_{10a} and c_{10b} together form a CRG composite. Apparently, the CRGs from Fig. 11.9 imply the existence of corresponding events / concepts in the process domain for each CRG node. The rationale behind this is to obtain a formal description of compliance rules that reflects the structure of the informal rules but is not yet tied to a particular process implementation.

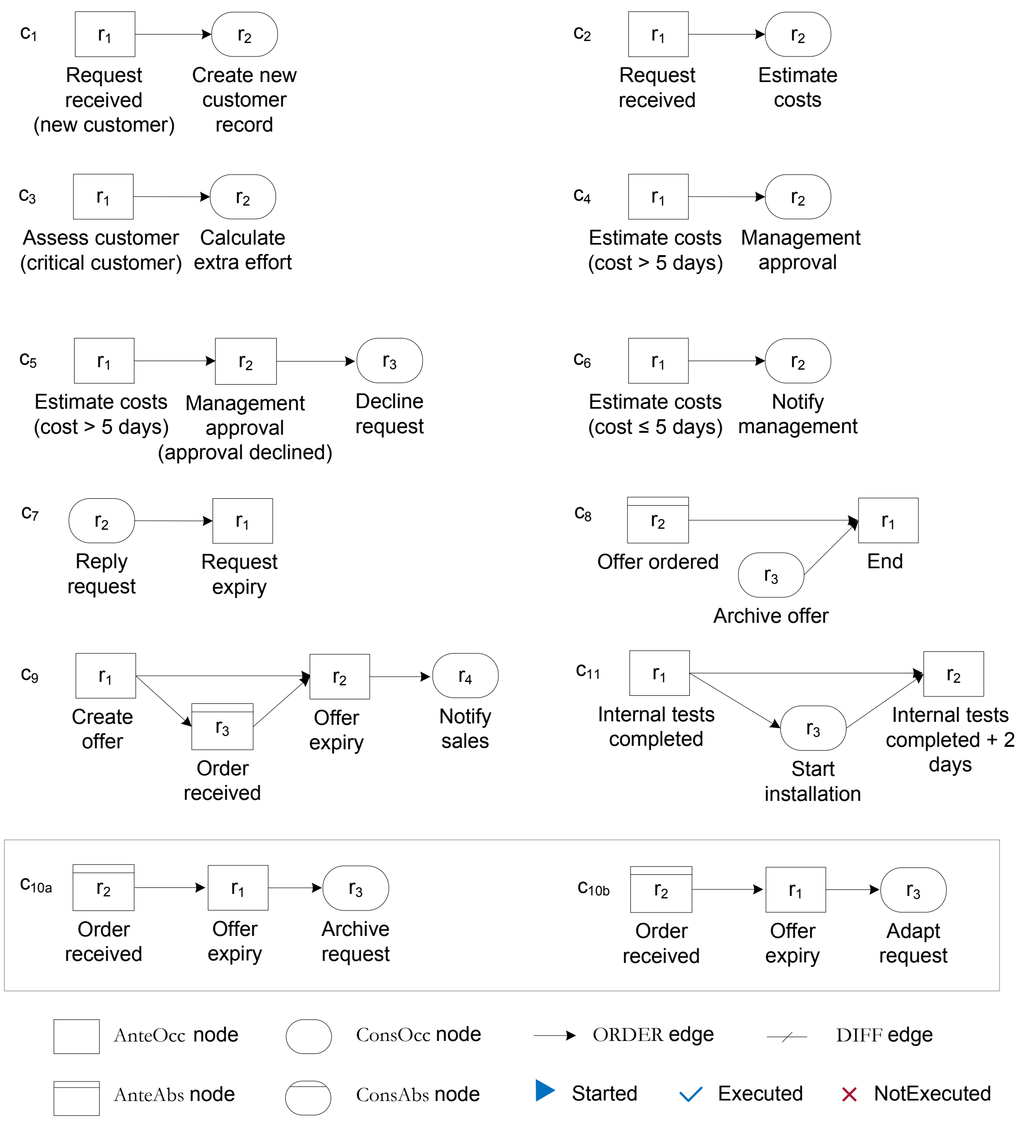


Figure 11.9.: CRGs capturing the constraints from Tab. 11.2

While the CRGs depicted in Fig. 11.9 reflect the basic structure of the underlying constraints, they cannot be utilized directly for compliance checks. As described in Section 1.1, informal compliance rules have to be interpreted with respect to the specific process implementation in order to derive suitable automated compliance controls. Against the background of the SeaFlows compliance checking framework, this necessitates the formalization of compliance rules based on implementation artifacts (e.g., process activities or events) in order to enable automated compliance checking and compliance monitoring. This constituted a major challenge in this case study. Yet this procedure is rewarding as it provides insights into the mechanisms of deriving compliance controls from informal compliance rules.

Based on the implementation of the project process, the informal rules were translated to CRGs that convey the intended semantics with respect to the implemented project process. This specifically means that the concepts associated with the nodes of the CRGs from Fig. 11.9 have to be translated to process artifacts. Note that the structure of the preliminary CRGs may have to be altered if concepts are reflected by a combination of process artifacts. In the translation process, we made interesting observations. Similar as in the HEP project (cf. Section 11.2), we observed that the particular structure of a CRG derived from an informal rule largely depends on the process implementation. Many parts of the process were not fully automated. For example, the internal and the integration tests. In the implementation of the process, the internal and the integration tests are indicated by assigning costs to these tests via an input mask. Thus, these activities / events attest that these tests were conducted.

Some of the compliance rules could not be automated due to the particular implementation of the project process. Consider, for example, compliance rule c_3. Provided that the process implementation provides artifacts indicating that a customer is critical and that potential extra effort is calculated, clearly c_3 could be modeled as a CRG as shown in Fig. 11.9. In the implementation of the project process, however, no such artifacts exist. The evaluation of a customer as critical and the calculation of extra effort happen within a graphical user interface of a custom application within a single activity. Thus, it would become necessary to integrate CRGs into this application in order to support c_3. We abstained from doing that. The situation is similar with c_1. Both c_1 and c_3 could be modeled as CRGs, suitable events in the process domain provided.

As CRGs do not support quantitative time constraints, c_{11} cannot be supported unless time events are provided (as, for example, illustrated in Fig. 11.9). In contrast to the HEP project (cf. Section 11.2), where such time events were provided, the implementation of the project process does not provide time events. For the same reason, c_7, c_9, and c_{10} cannot be checked using our compliance checking framework unless a timer is provided that triggers time events such as the expiry of a deadline.

The remaining CRGs, namely c_2, c_4, c_5, c_6, and c_8, were modeled based on the event model of the implemented project process. For that, it has to be investigated how the concepts in the process-independent CRGs are reflected in the process implementation. In this case, the structure of the CRGs could be preserved and only the activities associated with CRG nodes had to be adapted. Fig. 11.10 shows some of the CRGs modeled based on the process event model.

11.3.2. Compliance analysis

Using the modeled CRGs, the implemented project process was analyzed with respect to compliance. As the implementation level process contains many technical activities (such as database retrieval tasks) that do not affect compliance, these parts of the process model can be abstracted from when verifying the process. Thus, we manually reduced the process model leaving only those activities relevant to the imposed rules. Using SeaFlows Toolset, the reduced process model was explored and a `PEG` was created from this model (cf. Fig 11.11). We then applied the CRG execution engine (cf. Section 10.2) to verify compliance of the process model.

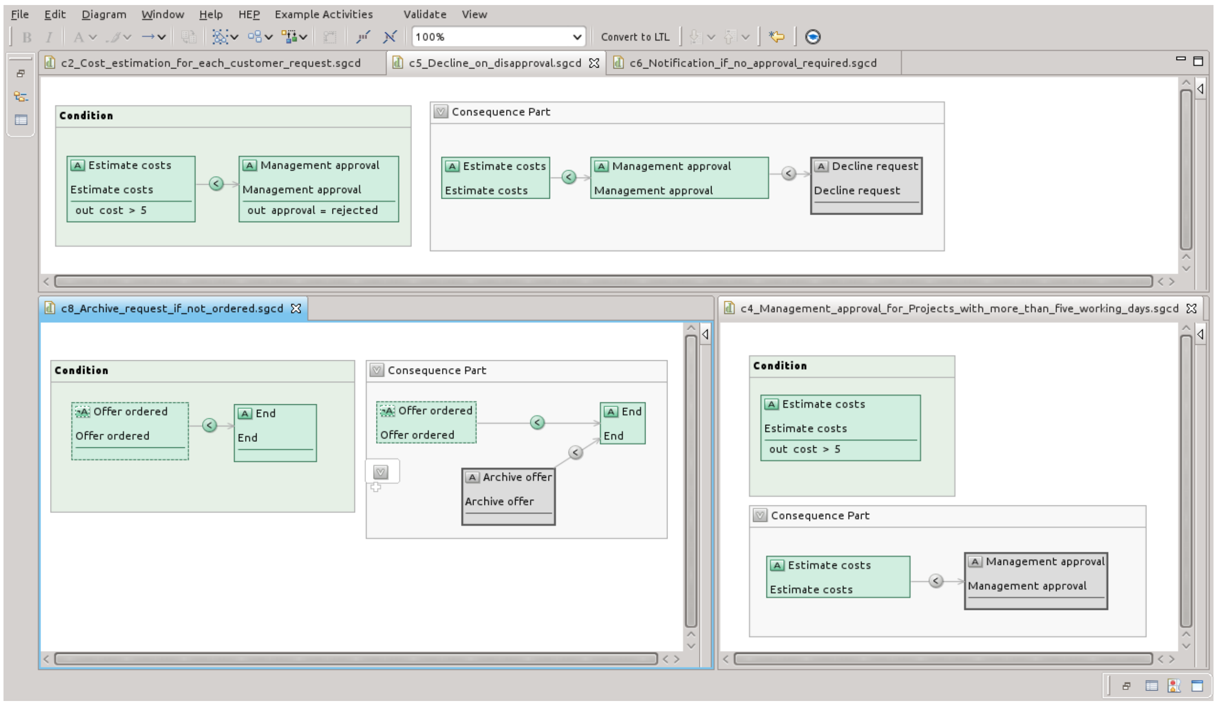

File Edit Diagram Window Help HEP Example Activities Validate View
100%
Convert to LTL
c2_Cost_estimation_for_each_customer_request.sgcd
c5_Decline_on_disapproval.sgcd
c6_Notification_if_no_approval_required.sgcd
Condition
Estimate costs
out cost > 5
Management approval
out approval = rejected
Consequence Part
Decline request
c8_Archive_request_if_not_ordered.sgcd
Offer ordered
End
Archive offer
c4_Management_approval_for_Projects_with_more_than_five_working_days.sgcd

Figure 11.10.: Rules imposed on the project process modeled as CRGs

It turned out that most but not all of the compliance rules are enforced in the project process model. Specifically, compliance verification using the compliance checking framework revealed that c_6 is not enforced in the project process. Among the constraints not checked using our framework, c_9 and c_{10} are not yet enforced in the process model. As mentioned previously, c_1 and c_3 require intra-activity constraints. This is, for example, addressed in the work of Künzle et al. [KR11b]. Compliance rule c_9 can be enforced by defining escalation strategies for activities (i.e., define actions due when, for example, an activity expires). The same is true for c_8. The compliance rules c_{10} and c_{11} require time events or time-aware constraints as, for example, addressed by Lanz et al. [LWR10].

We further simulated the execution of the project process and monitored compliance with imposed CRGs using the CRG engine (cf. Section 10.2). As the underlying process model bears compliance violations, clearly compliance violations can occur at the process instance level. The rationale behind simulating process instance execution was to apply compliance checks from the runtime perspective. Fig. 11.12 shows the visualization of a VIOLABLE MARKSTRUCTURE yielded during process instance execution. Specifically, it shows that the activity *Decline request* is pending and needs to be executed in order to comply.

11.3.3. Lessons learned

Even though the process under investigation probably does not yet belong to the huge processes that we may encounter in practice [BRB07], it was yet difficult to check compliance of the project process manually. This is particular due to the multitude of branches and associated conditions.

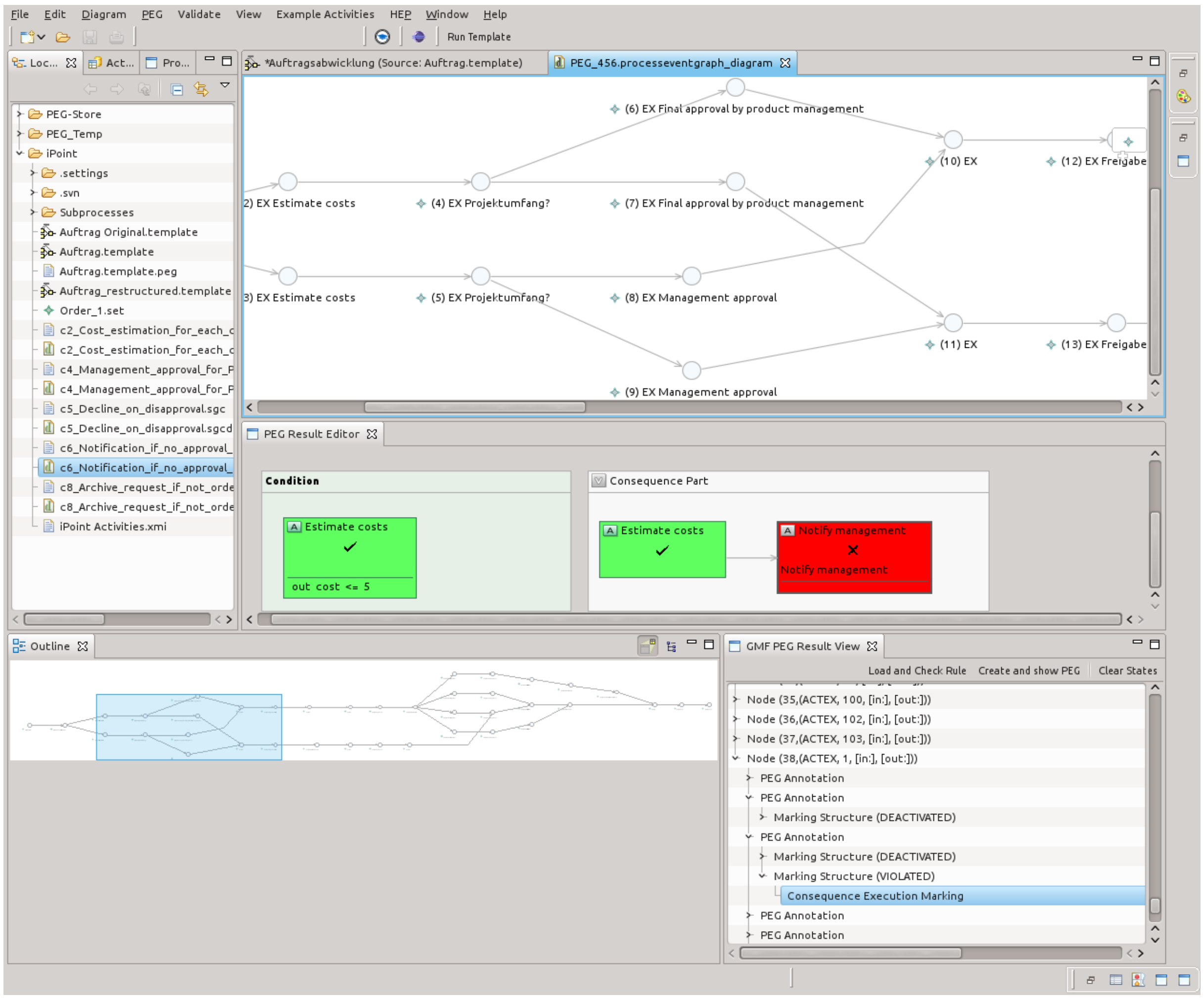
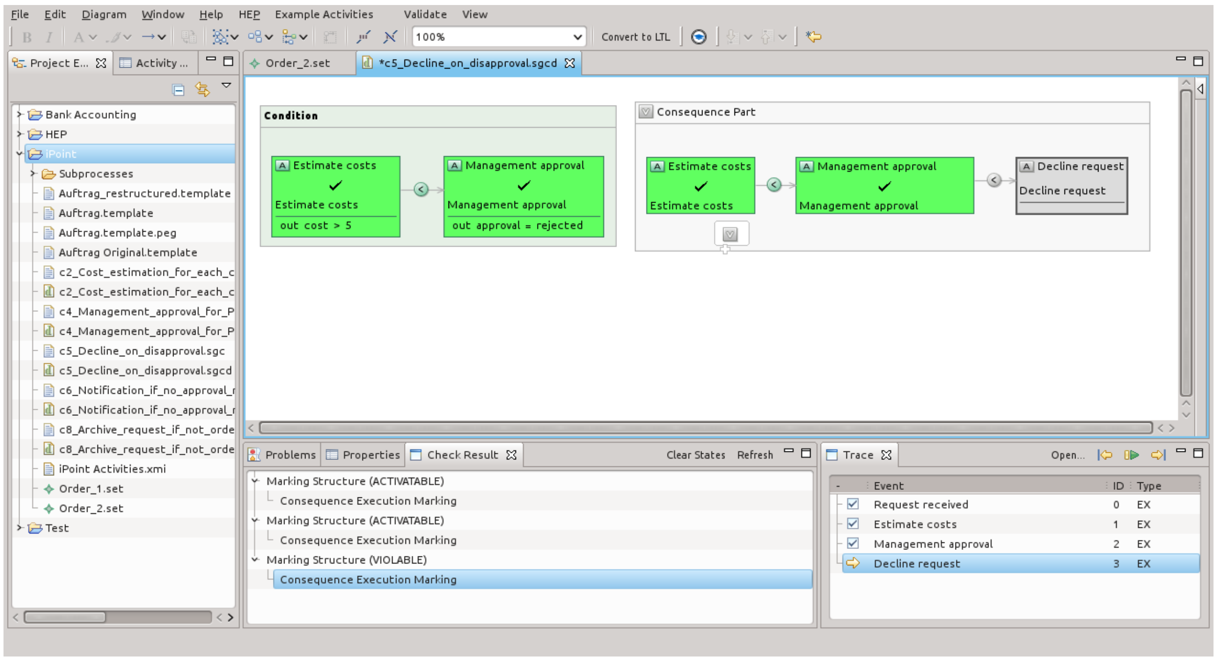

File Edit Diagram PEG Validate View Example Activities HEP Window Help
Run Template
*Auftragsabwicklung (Source: Auftrag.template)
PEG_456.processeventgraph_diagram
PEG-Store
PEG_Temp
iPoint
.settings
.svn
Subprocesses
Auftrag Original.template
Auftrag.template
Auftrag.template.peg
Auftrag_restructured.template
Order_1.set
c2_Cost_estimation_for_each_c
c4_Management_approval_for_F
c5_Decline_on_disapproval.sgc
c5_Decline_on_disapproval.sgcd
c6_Notification_if_no_approval_
c8_Archive_request_if_not_orde
iPoint Activities.xmi
(6) EX Final approval by product management
(10) EX
(12) EX Freigabe
2) EX Estimate costs
(4) EX Projektumfang?
(7) EX Final approval by product management
3) EX Estimate costs
(5) EX Projektumfang?
(8) EX Management approval
(11) EX
(13) EX Freigabe
(9) EX Management approval
PEG Result Editor
Condition
Estimate costs
out cost <= 5
Consequence Part
Estimate costs
Notify management
Notify management
Outline
GMF PEG Result View
Load and Check Rule
Create and show PEG
Clear States
Node (35,(ACTEX, 100, [in:], [out:]))
Node (36,(ACTEX, 102, [in:], [out:]))
Node (37,(ACTEX, 103, [in:], [out:]))
Node (38,(ACTEX, 1, [in:], [out:]))
PEG Annotation
PEG Annotation
Marking Structure (DEACTIVATED)
PEG Annotation
Marking Structure (DEACTIVATED)
Marking Structure (VIOLATED)
Consequence Execution Marking
PEG Annotation
PEG Annotation

Figure 11.11.: Design time compliance verification of the abstracted project process

Hence, the possibility to check compliance in an automated manner seems to be useful already for processes of that size.

The translation of informal constraints to checkable compliance rules at the level of the implemented project process constituted a major challenge in this study. It confirmed our observation in the HEP project (cf. Section 11.2) that the structure of the derived compliance rules heavily depends on the event model employed. The approach to first model compliance rules at the informal level as CRGs and then map the artifacts used in these CRGs to implementation artifacts seems viable. We can think of tool support for facilitating this approach. Buchwald et al. introduce an approach for bridging the gap between business process models and service composition specifications in [BBR11]. This approach may be adopted for mapping between CRGs of different specification levels. We further observed that the structure of compliance rules modeled as CRGs further depends on the "frame" that can be assumed. For example, since we can assume that the process follows a certain workflow specification, we can assume a certain ordering of the activities in the process. This affects the degree of details that need



Marking Structure (ACTIVATABLE)
  Consequence Execution Marking
Marking Structure (ACTIVATABLE)
  Consequence Execution Marking
Marking Structure (VIOLABLE)
  Consequence Execution Marking

Event	ID	Type
Request received	0	EX
Estimate costs	1	EX
Management approval	2	EX
Decline request	3	EX

Figure 11.12.: Simulated compliance monitoring for the abstracted project process

to be compiled into a CRG specification. Finally, we identified that time-aware constraints as already envisioned in Section 6.4.2 constitute a desirable extension of the CRG language.

11.4. Summary and outlook

In this chapter, we presented our efforts to evaluate the proposed compliance checking framework in practice. We first showed how CRGs can be utilized to model the property specification patterns described in Section 11.1. As mentioned earlier, we believe that the predefinition of frequently occurring rule patterns can leverage the practical application of a compliance checking framework. An interesting observation when modeling the property specification patterns was that already modeled patterns, such as *precedence chain*, can be easily extended using the primitives of the CRG language. We can further think of introducing a semantically higher layer based on the CRG modeling primitives in order to provide a semantically high level interface for compliance rule modeling. This can be utilized, for example, to semantically discriminate between scopes and the actual rule antecedent.

In Section 11.2 and 11.3, we described the application of the SeaFlows compliance checking framework to realistic data in two different settings. In both these case studies, we observed that the checkable compliance rule derived from abstract / informal constraints largely depends on the particular process implementation. Generally, the process of translating informal compliance requirements to checkable compliance rules is highly interesting and constituted a major challenge in the studies. As described in Section 11.3.3, an approach that facilitates mapping between compliance rules of different specification levels could be beneficial. We further identified interesting extensions of the CRG language and the compliance checking framework. Specifically, the study confirmed the intuition that data relations and quantitative time constraints constitute important extensions of the CRG language (cf. Section 6.4.2). In future work, it would be interesting to apply the proposed compliance checking framework in an extended case study involving the complete process lifecycle to gain a round-trip experience. This, however, requires an extensive project that can be accompanied from early on.

12 Summary and outlook

Nowadays, compliance-awareness is undoubtedly of utmost importance for companies. Still the costs for compliance-awareness are a major issue. Even though an automated approach to compliance checking and enforcement has been advocated in recent literature as a means to tame the high costs for compliance-awareness, the potential of automated mechanisms for supporting business process compliance is not yet depleted. Business process compliance deals with the question whether processes are designed and executed in harmony with imposed regulations. In this thesis, we proposed a compliance checking framework for automating business process compliance verification within PrMSs. Such process-aware information systems constitute an ideal environment for the systematic integration of automated business process compliance checking since they bring together different perspectives on a business process and provide access to process data.

In Chapter 2, we described requirements on a compliance checking framework. Specifically, it has to support the modeling of compliance requirements in a manner suitable for automated checks and has to accommodate mechanisms to verify compliance at both process design and process runtime. Key to the practical application of a compliance checking framework will further be its ability to provide meaningful and intelligible reports based on which concrete process adaptations or other measures for resolving noncompliance can be derived. As discussed in the state-of-the-art discussion in Chapter 3, existing compliance checking approaches predominantly focus on specific scenarios rather than on the integrated support of process design and process runtime. The requirement for intelligible compliance reports explaining the root-cause of compliance violations and detected compliance states is still not yet addressed in an adequate manner or necessitates supporting mechanisms. Existing compliance frameworks that employ a holistic view on business process compliance issues, in contrast, focus on compliance management in the large rather than on the actual compliance checks. While these frameworks are valuable for establishing an overall approach to compliance management in general, they particularly rely on existing techniques, such as model checking, for conducting compliance checks. These techniques, again, are applicable only to specific scenarios (e.g., process design time) and suffer from known limitations as discussed in Chapter 3. Thus, there is still need for a compliance checking framework that supports both process design and process runtime and is capable of providing insights into detected compliance states. The SeaFlows compliance checking framework proposed in this thesis can become part of an overall integrated compliance management framework. In Section 12.1, we first summarize the contributions of this work with respect to the research objectives. Section 12.2 then provides an outlook on future research challenges that we identified in the course of our research.

12.1. Contributions

Our compliance checking framework resulted from two major working packages addressed in this work. One working package deals with the question how to model compliance rules in an adequate manner. The other working package addresses the question how to check compliance rules at process design and process runtime such that detailed compliance reports can be provided.

Compliance rule specification In this work, we proposed the compliance rule graph (CRG) language, an approach for modeling compliance rules based on a graph notation. The rationale behind this is to hide the complexity of a formal language from the modeler, an objective that is motivated by our experience with LTL. Our major goal was to provide a simple yet powerful and extensible compliance rule language. Thus, we adopt the assumption underlying graph-based process description languages that a graph notation is suitable to represent constraints on the occurrence and ordering of activities. In contrast to many related approaches in literature, the CRG language is not a collection of patterns but a truly compositional language that consists of modeling primitives such as nodes and edges expressing the occurrence, absence, or ordering of activity executions. A complex compliance rule is composed by assembling these primitives. Instead of employing a navigational mental model when modeling rules as with LTL, we aimed at a more straight-forward pattern specification model. This specifically enables to easily refer to specific past or future events. Thus, an existing CRG can be refined by inserting further modeling primitives without having to restructure the complete rule structure. The proposed language can be parsed in order to generate natural language descriptions. Due to its extensibility, further features can be integrated easily.

Following the requirement for formalization (cf. Section 2.1.1.2), the language is further associated with defined formal semantics, which enables the unambiguous interpretation of CRGs. In particular, each compliance rule modeled using the CRG language can be mapped to a rule formula specified in first-order predicate logic. Thus, existing algorithms can be applied for their analysis. Rule formulas, in turn, can be formally interpreted over execution traces. Being footed on an event-based execution trace model as described in Chapter 5, the formal semantics of CRGs is not restricted to a specific process description language. In fact, execution traces serve as language-independent representation of process executions in our framework. Hence, CRGs can be utilized to constrain business processes specified using a multitude of description languages.

Compliance checking at process design and process runtime Clearly, the ability to verify process models and process instances against imposed compliance rules is key to any compliance checking framework. Our objective was to enable checking process models and process instances against imposed CRGs using the same mechanisms. This way, compliance rules do not have to be translated into other representations for conducting design or runtime checks. In order to achieve that, we equipped CRGs with operational semantics that can be applied to conduct compliance checks as described in Chapter 7.

The basic idea is to exploit the graph structure of CRGs for compliance checking. In order to encode compliance states such that they can be interpreted for generating compliance reports, we introduce state markings for CRG nodes. This way, each compliance state with respect to a CRG that a process execution may yield can be represented using the very CRG and suitable state markings (so-called MarkStructures). In contrast to states of automatons generated from temporal logic formulas as used for explicit model checking, the reachable compliance states can be interpreted easily as they refer directly to the CRG structure. The transitions between compliance states are defined by execution and marking rules that alter markings in an adequate manner.

A major objective of our work is to enable meaningful compliance reports as this requirement still constitutes a major limitation of most related approaches in literature. One benefit of the approach proposed in this thesis is specifically that explanations for compliance violations can be derived from the compliance states encoded using CRGs and state markings. Even if no compliance violation has occurred (yet), the meaningful compliance states can provide valuable insights into the compliance situation. In particular, it is easily possible to derive measures to avert potential violations from the information encoded in a reached compliance state. This can be seized for providing support to comply at runtime.

The operationalization of CRGs is inspired by pattern matching mechanisms and is conducted by applying rules that alter compliance states according to observed events in a process to be verified. Thus, the operational semantics can be applied to both design and runtime compliance checks as it supports incremental application (as required for runtime compliance monitoring). It further enables to "instantiate" a CRG for each new activation of the compliance rule observed in the process execution. Thus, the framework is able to provide compliance reports not only on the general enforcement of the CRG but also on the particular rule activations in a process and their individual compliance (cf. Section 2.1.2).

Similar to the formal semantics of CRGs, the operational semantics is defined over event-based execution traces. Thus, the proposed compliance checking framework can be applied to business processes specified using a multitude of description languages. This will leverage the practical application of the framework.

The CRG operational semantics described in Chapter 7 together with the CRG language described in Chapter 6 provide the fundament for a compliance checking framework. The application of the proposed concepts to realize compliance checks at process design and process runtime is described in Chapter 8. At process design time, a process model is verified against an imposed CRG by applying the operational semantics to explore the process model and to detect whether the model enables executions bearing compliance violations. We described different strategies for doing this, which can be applied depending on the desired level of granularity. Specifically, the compliance verification detects compliance states reflecting the behavior with respect to the imposed CRG enabled by the process model. These compliance states not only reveal compliance violations but also enable the derivation of meaningful compliance diagnoses. For example, it can be detected in which traces a violation occurs and whether a CRG becomes activated or is violated in all or only some specific executions of the process. This information and the MarkStructures bearing compliance violations can be utilized for explaining and visualizing detected noncompliance within the process model to be checked. At process runtime, events observed during process execution can be processed in order to provide detailed feedback on the effective compliance state of a running process instance. Moreover, the possible future behavior of a process instance predefined in the underlying process model can be exploited for updating compliance predictions. Together with monitoring the effective compliance state of a process instance, this can help to identify potential compliance threats in a timely manner. Altogether, the proposed compliance checking framework can be considered a model checking approach that aims at comprehensive diagnoses and is tailored towards the CRG language.

Pre- and postprocessing activities in the compliance checking process Beyond the mechanisms for modeling compliance requirements in a checkable manner and for conducting the actual compliance checks, further aspects are vital for a comprehensive compliance checking framework. Specifically, questions on necessary activities before and after conducting the actual compliance checks arise. In Chapter 9, we outlined how processes can be transformed into state space representations for compliance checking for the example of the process description language ADEPT [Rei00] (i.e., WSM nets). We further addressed the state explosion problem that may arise for complex processes to be verified (cf. Section 2.1.2.1) by pointing out a variety of abstraction strategies collected from literature and developed in the SeaFlows project. After conducting the actual compliance checks, detected compliance violations need to be conveyed to users in an intelligible manner. Considerations on how to accomplish this are also described in Chapter 9.

Implementation and practical application The technical feasibility of the proposed compliance checking framework is demonstrated by means of a proof-of-concept implementation described in Chapter 10. Due to the iterative research procedure employed in our work (cf. Section 2.2.3), our prototype implementation, the SeaFlows Toolset, comprises different tools. Specifically, SeaFlows Toolset comprises tools for modeling compliance rules using the CRG language and for conducting compliance checks using the operational semantics of CRGs. Moreover, SeaFlows Toolset also features prototype implementations that showcase the application of abstraction strategies for taming the state explosion problem. Furthermore, a prototype demonstrating ideas on the presentation of compliance violations within process models is included in the toolset.

In meta-analyses, we often encountered certain property specification patterns initially collected by Dwyer et al. [DAC99] and applied by a plethora of compliance checking approaches (e.g., [ASW09, YMHJ06, MMWA11]). Clearly, these patterns have to be supported. As shown in Chapter 11, the CRG language enables the specification of these patterns in a straight-forward manner. We further showed that the patterns can be interpreted in different ways and that the

CRG language enables to be precise about the intended interpretation. We specifically consider the possibility of easily extending and refining such patterns by adding or removing CRG modeling primitives, for example, to add a further absence constraint within the context of an existing rule pattern, a major asset of the CRG language.

In an effort to evaluate the practical feasibility of our compliance checking framework, we applied it to realistic compliance rules and processes from two different domains. In particular, we utilized the SeaFlows compliance checking framework to analyze higher education processes in a project called HEP together with colleagues from University of Vienna. Moreover, we analyzed a process describing the processing of projects in an IT company spanning all activities from a project request to its completion. The case studies confirmed that the proposed compliance checking framework is suitable for analyzing business process compliance. We are positive that intelligible and meaningful compliance reports can be derived based on the compliance state information that our framework is able to provide. As summarized in Chapter 11, we further observed that the translation of informal compliance requirements to checkable compliance rules over the process implementation constitutes a major challenge. In the case of the project process, in particular, we followed the procedure of first modeling compliance requirements using CRGs but without paying attention to the process implementation and then mapping the associated events to process implementation artifacts. These two case studies were specifically rewarding as they provided insights into the overall process of deriving compliance controls from compliance requirements described in Section 1.1.

12.2. Outlook

In each chapter introducing the concepts of the SeaFlows compliance checking framework, we elaborated on possible extensions of the proposed approach in the discussion part. The ideas on future extensions and research are in three respects:

- Language-wise: Extensions of the CRG language and its operationalization,
- Method-wise: Extensions with respect to methodic support for compliance checking,
- Application-wise: Further applications for the CRG approach

Extensions of the CRG language and operationalization As pointed out in Chapter 6, the CRG language can be extended in different respects in order to accommodate more sophisticated compliance requirements. In the case studies described in Chapter 11, we observed that quantitative time constraints constitute a desirable extension of the CRG language. For that extensions at the notation and at the formal level become necessary. In this context, the integration of concepts for time-aware process-aware information systems developed in the ATAPIS project [Lan08, LWR10] will be particularly interesting. As described in Section 6.4.2, data relations constitute a further desirable extension of the CRG language. Notation-wise, data relations may be represented through designated nodes and edges. As CRGs are formalized as rule formulas in first-order predicate logic, data relations can be integrated easily. At the operational level, this requires extending the operational semantics taking into account the evaluation of data relations. So far, SeaFlows Toolset already supports data relations that enable to associate different events with the same object. This, in turn, can be utilized to define artifact-oriented compliance rules as illustrated in [LRMKD11]. More advanced support is left to future research.

Extensions with respect to methodic support For ease of use, the introduction of a higher semantic layer for compliance rule modeling based on the CRG modeling primitives should be considered. For example, one can think of composite constructs for expressing relations that can be defined using a combination of modeling primitives such as *the next occurrence of B after A*. The provision of such higher level modeling constructs in a CRG modeling tool can further facilitate CRG modeling in practice.

As observed in the case studies described in Chapter 11, methodic support for the process of deriving checkable compliance rules from informal compliance requirements is desirable. In our case study analyzing the project process of an IT company (cf. Section 11.3), we first modeled compliance rules using the CRG language based on an assumed event model directly derived from the informal requirements. Then, the artifacts referred to in the CRGs were translated to corresponding artifacts in the implemented process. In this context, tool support for the systematic derivation of checkable rules from informal requirements suggests itself. Existing approaches for mapping between business and implementation level process models such as [BBR11] can be adopted for this purpose.

Application-related issues As pointed out earlier, the proposed compliance checking framework can be integrated into an overall compliance management framework as, for example, devised in the COMPAS project [The11]. On the one hand, the compliance management framework would benefit from a compliance checking approach tackling both process design and process runtime. On the other hand, functionalities of utmost importance to an integrated approach to compliance management, such as managing relevant passages of compliance requirements, will be provided by the compliance management framework.

As described in Section 8.5.2, declarative process models can be defined using CRGs and enacted using the CRG operational semantics. This could be an interesting application of the CRG approach as the easy derivation of measures to comply with an imposed rule can be of particular advantage in this context.

Bibliography

[AAD+04] Acker, H., Atkinson, C., Dadam, P., Rinderle, S., et al. Aspekte der komponentenorientierten Entwicklung adaptiver prozessorientierter Unternehmenssoftware. In *Architekturen, Komponenten, Anwendungen – Proc. 1. Verbundtagung AKA'04*, number P-57 in LNI, pages 7–24. Koellen-Verlag, 2004. In German.

[Aal99] van der Aalst, W. Formalization and verification of event-driven process chains. *Information and Software Technology*, 41(10):639 – 650, 1999.

[ABD05] van der Aalst, W., de Beer, H., and van Dongen, B. Process mining and verification of properties: An approach based on temporal logic. In *On the Move to Meaningful Internet Systems 2005: OTM Confederated International Conferences CoopIS, DOA, and ODBASE 2005, Proceedings, Part I*, volume 3761 of *LNCS*, pages 130–147. 2005.

[ACG+06] Alberti, M., Chesani, F., Gavanelli, M., Lamma, E., et al. Computational logic for run-time verification of web services choreographies: Exploiting the SOCS-SI tool. In *Proceedings of the Third International Conference on Web Services and Formal Methods (WS-FM)*, volume 4184 of *LNCS*, pages 58–72. Springer, 2006.

[ACG+07] Alberti, M., Chesani, F., Gavanelli, M., Lamma, E., et al. Verifiable agent interaction in abductive logic programming: the SCIFF framework. *ACM Transactions on Computational Logic*, 5:1–41, 2007.

[ACG+08] Alberti, M., Chesani, F., Gavanelli, M., Lamma, E., et al. Expressing and verifying business contracts with abductive logic programming. *Int. J. Electron. Commerce*, 12(4):9–38, 2008.

[ADW08] Awad, A., Decker, G., and Weske, M. Efficient compliance checking using BPMN-Q and temporal logic. In *Proceedings of the 6th International Conference on Business Process Management (BPM 2008)*, volume 5240 of *LNCS*, pages 326–341. Springer, 2008.

[AHW+11] van der Aalst, W., van Hee, K., van der Werf, J.M., Kumar, A., et al. Conceptual model for online auditing – on quantitative methods for detection of financial fraud. *Decision Support Systems*, 50(3):636 – 647, 2011.

[AJKL06] Agrawal, R., Johnson, C., Kiernan, J., and Leymann, F. Taming compliance with Sarbanes-Oxley internal controls using database technology. In *Proceedings of the 22nd International Conference on Data Engineering (ICDE'06)*, page 92. IEEE Computer Society, 2006.

[All83] Allen, J.F. Maintaining knowledge about temporal intervals. *Communications of the ACM*, 26(11):832–843, 1983.

[ALS11] Accorsi, R., Lowis, L., and Sato, Y. Automated certification for compliant cloud-based business processes. *Business & Information Systems Engineering*, 3(3):145–154, 2011.

[AM05] van der Aalst, W.M.P. and de Medeiros, A.K.A. Process mining and security: Detecting anomalous process executions and checking process conformance. *Electr. Notes Theor. Comput. Sci.*, 121:3–21, 2005.

[And07] Andriessen, D. Combining design-based research and action research to test management solutions. In *7th World Congress Action Learning, Action Research and Process Management, Groningen, 22-24 August, 2007*. 2007.

[AP06] van der Aalst, W. and Pesic, M. DecSerFlow: Towards a truly declarative service flow language. In *Proceedings of the Third International Workshop on Web Services and Formal Methods (WS-FM 2006)*, volume 4184 of *LNCS*, pages 1–23. Springer, 2006.

[ASI10] Abdullah, N.S., Sadiq, S.W., and Indulska, M. Emerging challenges in information systems research for regulatory compliance management. In *Proceedings of the 22nd International Conference on Advanced Information Systems Engineering (CAiSE 2010)*, volume 6051 of *LNCS*, pages 251–265. Springer, 2010.

[ASSR93] Attie, P., Singh, M., Sheth, A., and Rusinkiewicz, M. Specifying and enforcing intertask dependencies. In *Proceedings of the 19th International Conference on Very Large Data Bases (VLDB'93)*, pages 134–145. Morgan Kaufmann Publishers Inc., 1993.

[ASW09] Awad, A., Smirnov, S., and Weske, M. Towards resolving compliance violations in business process models. In *2nd International Workshop on Governance, Risks and Compliance in Information Systems*. 2009.

[AW09] Awad, A. and Weske, M. Visualization of compliance violation in business process models. In *Business Process Management Workshops, BPM 2009 International Workshops. Revised Papers*, volume 43 of *LNBIP*, pages 182–193. Springer, 2009.

[Awa07] Awad, A. BPMN-Q: A language to query business processes. In *Proceedings of the 2nd International Workshop on Enterprise Modelling and Information Systems Architectures (EMISA 2007)*, volume P-119 of *LNI*, pages 115–128. GI, 2007.

[AWW09] Awad, A., Weidlich, M., and Weske, M. Specification, verification and explanation of violation for data aware compliance rules. In *Proceedings of 7th International Conference on Service Oriented Computing (ICSOC-Service Wave'09)*, volume 5900 of *LNCS*, pages 500–515. Springer, 2009.

[AWW10] Awad, A., Weidlich, M., and Weske, M. Consistency checking of compliance rules. In *Business Information Systems*, volume 47 of *LNBIP*, pages 106–118. Springer, 2010.

[BAC$^+$11] Becker, J., Ahrendt, C., Coners, A., Weiß, B., et al. Modeling and analysis of business process compliance. In *Governance and Sustainability in Information Systems*, volume 366 of *IFIP Advances in Information and Communication Technology*, pages 259–269. Springer, 2011.

[BBB$^+$11] Becker, J., Bergener, P., Breuker, D., Delfmann, P., et al. An efficient business process compliance checking approach. In *Governance and Sustainability in Information Systems*, volume 366 of *IFIP Advances in Information and Communication Technology*, pages 282–287. Springer, 2011.

[BBD$^+$11] Becker, J., Bergener, P., Delfmann, P., Eggert, M., et al. Supporting business process compliance in financial institutions – a model-driven approach. In *10. Internationale Tagung Wirtschaftsinformatik (WI 2011)*, page 75. 2011.

[BBF$^+$01] Bérard, B., Bidoit, M., Finkel, A., Laroussinie, F., et al. *Systems and Software Verification – Model-Checking Techniques and Tools.* Springer, 2001.

[BBMNS11] Beheshti, S.M.R., Benatallah, B., Motahari-Nezhad, H., and Sakr, S. A query language for analyzing business processes execution. In *Proceedings of the 9th International Conference on Business Process Management (BPM 2011)*, volume 6896 of *LNCS*, pages 281–297. Springer, 2011.

[BBR11] Buchwald, S., Bauer, T., and Reichert, M. Bridging the gap between business process models and service composition specifications. In *Service Life Cycle Tools and Technologies: Methods, Trends and Advances*, pages 124–153. Idea Group Reference, November 2011.

[BDL$^+$10] Birukou, A., D'Andrea, V., Leymann, F., Serafinski, J., et al. An integrated solution for runtime compliance governance in SOA. In *Proceedings of the 8th International Conference on Service-Oriented Computing (ICSOC 2010)*, volume 6470 of *LNCS*, pages 122–136. Springer, 2010.

[BDSV05] Brambilla, M., Deutsch, A., Sui, L., and Vianu, V. The role of visual tools in a web application design and verification framework: A visual notation for LTL formulae. In *Proceedings of the 5th International Conference on Web Engineering (ICWE 2005)*, volume 3579 of *LNCS*, pages 557–568. Springer, 2005.

[BGG97] Börger, E., Grädel, E., and Gurevich, Y. *The Classical Decision Problem.* Perspectives in Mathematical Logic. Springer, 1997.

[BGL$^+$00] Bensalem, S., Ganesh, V., Lakhnech, Y., noz, C.M., et al. An overview of SAL. In *Proc. of the Fifth NASA Langley Formal Methods Workshop*, pages 187–196. NASA Langley Research Center, 2000.

[Bit06] Bitsch, F. *Verfahren zur Spezifikation funktionaler Sicherheitsanforderungen für Automatisierungssysteme in Temporallogik.* Ph.D. thesis, Universität Stuttgart, 2006. In German.

[Bra05] Brambilla, M. LTL formalization of BPML semantics and visual notation for linear temporal logic. Technical report, Dipartimento Elettronica e Informazione, Politecnico di Milano, 2005.

[BRB07] Bobrik, R., Reichert, M., and Bauer, T. View-based process visualization. In *Proceedings of the 5th International Conference on Business Process Management (BPM 2007)*, volume 4714 of *LNCS*, pages 88–95. Springer, 2007.

[BSB$^+$07] Blaser, R., Schnabel, M., Biber, C., Bäumlein, M., et al. Improving pathway compliance and clinician performance by using information technology. *I. J. Medical Informatics*, 76(2-3):151–156, 2007.

[CC77] Cousot, P. and Cousot, R. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages (POPL)*, pages 238–252. ACM, 1977.

[CGK11] Courcelle, B., Gavoille, C., and Kanté, M.M. Compact labelings for efficient first-order model-checking. *J. Comb. Optim.*, 21(1):19–46, 2011.

[CGMP99] Clarke, E.M., Grumberg, O., Minea, M., and Peled, D.A. State space reduction using partial order techniques. *International Journal on Software Tools for Technology Transfer*, 2(3):279–287, 1999.

[CGP99] Clarke, E.M., Grumberg, O., and Peled, D.A. *Model Checking.* The MIT Press, 1999.

[Cha00] Chan, W. Temporal-logic queries. In *Computer Aided Verification*, volume 1855 of *LNCS*, pages 450–463. Springer, 2000.

[Chu36] Church, A. An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58:345–363, 1936.

[CMMS07a] Chesani, F., Mello, P., Montali, M., and Storari, S. Testing careflow process execution conformance by translating a graphical language to computational logic. In *Proceedings of the 11th Conference on Artificial Intelligence in Medicine (AIME 2007)*, volume 4594 of *LNAI*, pages 479–488. Springer, 2007.

[CMMS07b] Chesani, F., Mello, P., Montali, M., and Storari, S. Towards a DecSerFlow declarative semantics based on computational logic. Technical Report DEIS-LIA-07-002, DEIS Bologna, Italy, 2007.

[CPRS05] Cole, R., Purao, S., Rossi, M., and Sein, M.K. Being proactive: where action research meets design research. In *Proceedings of the 26th International Conference on Information Systems (ICIS 2005)*, pages 325–336. Association for Information Systems, 2005.

[DAC98] Dwyer, M.B., Avrunin, G.S., and Corbett, J.C. Property specification patterns for finite-state verification. In *Proceedings of the Second Workshop on Formal Methods in Software Practice*, pages 7–15. ACM, 1998.

[DAC99] Dwyer, M.B., Avrunin, G.S., and Corbett, J.C. Patterns in property specifications for finite-state verification. In *Proceedings of the 21st International Conference on Software Engineering (ICSE 1999)*, pages 411 – 420. ACM, 1999.

[Das03] Das, S. *Predicate Abstraction.* Ph.D. thesis, Stanford University, 2003.

[DAV05] van Dongen, B.F., van der Aalst, W.M.P., and Verbeek, H.M.W. Verification of EPCs: Using reduction rules and Petri nets. In *Proceedings of the 17th International Conference on Advanced Information Systems Engineering (CAiSE 2005)*, volume 3520 of *LNCS*, pages 372–386. Springer, 2005.

[Daw07] Dawar, A. Model-checking first-order logic: Automata and locality. In *Proceedings of the 21st International Workshop on Computer Science Logic (CSL 2007)*, volume 4646 of *LNCS*, page 6. Springer, 2007.

[Det97] Dettmer, H.W. *Goldratt's Theory of Constraints: A Systems Approach to Continuous Improvement.* ASQ Quality Press, 1997.

[DGG$^+$10] D'Aprile, D., Giordano, L., Gliozzi, V., Martelli, A., et al. Verifying business process compliance by reasoning about actions. In *Proceedings of the 11th International Workshop on Computational Logic in Multi-Agent Systems (CLIMA 2011)*, volume 6245 of *LNCS*, pages 99–116. Springer, 2010.

[DKR04] Davulcu, H., Kifer, M., and Ramakrishnan, I.V. CTR-S: a logic for specifying contracts in semantic web services. In *Proceedings of the 13th International Conference on World Wide Web – Alternate Track Papers & Posters (WWW 2004)*, pages 144–153. ACM, 2004.

[DKRR98] Davulcu, H., Kifer, M., Ramakrishnan, C.R., and Ramakrishnan, I.V. Logic based modeling and analysis of workflows. In *Proceedings of the 17th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS 1998)*, pages 25–33. ACM Press, 1998.

[DRR$^+$08] Dadam, P., Reichert, M., Rinderle, S., Jurisch, M., et al. Towards truly flexible and adaptive process-aware information systems. In *Proceedings of the 2nd International United Information Systems Conference (UNISCON 2008)*, pages 72–83. 2008.

[EKR95] Ellis, C., Keddara, K., and Rozenberg, G. Dynamic change within workflow systems. In *Proceedings of the Conference on Organizational Computing Systems (COOCS 1995)*, pages 10–21. ACM, 1995.

[ETHP10a] Elgammal, A., Turetken, O., van den Heuvel, W., and Papazoglou, M. On the formal specification of regulatory compliance: A comparative analysis. In *Service-Oriented Computing – ICSOC 2010 International Workshops, PAASC, WESOA, SEE, and SOC-LOG, Revised Selected Papers*, volume 6568 of *LNCS*, pages 27–38. Springer, 2010.

[ETHP10b] Elgammal, A., Turetken, O., van den Heuvel, W.J., and Papazoglou, M. Root-cause analysis of design-time compliance violations on the basis of property patterns. In *Proceedings of the 8th International Conference on Service-Oriented Computing (ICSOC 2010)*, volume 6470 of *LNCS*, pages 17–31. Springer, 2010.

[FES05] Förster, A., Engels, G., and Schattkowsky, T. Activity diagram patterns for modeling quality constraints in business processes. In *Proceedings of the 8th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2005)*, volume 3713 of *LNCS*, pages 2–16. Springer, 2005.

[FESS06] Förster, A., Engels, G., Schattkowsky, T., and Van der Straeten, R. A pattern-driven development process for quality standard-conforming business process models. In *2006 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC 2006)*, pages 135–142. IEEE Computer Society, 2006.

[FESS07] Förster, A., Engels, G., Schattkowsky, T., and Van der Straeten, R. Verficiation of business process quality constraints based on visual process patterns. In *First Joint IEEE/IFIP Symposium on Theoretical Aspects of Software Engineering (TASE 2007)*, pages 197–208. IEEE Computer Society, 2007.

[FFS08] Feja, S., Fötsch, D., and Sebastian, S. Grafische Validierungsregeln am Beispiel von EPKs. In *Software Engineering 2008 – Workshopband, Fachtagung des GI-Fachbereichs Softwaretechnik, 18.-22.2.2008 in München*, volume 122 of *LNI*, pages 198–204. GI, 2008. In German.

[FLM$^+$09] Fahland, D., Lübke, D., Mendling, J., Reijers, H.A., et al. Declarative versus imperative process modeling languages: The issue of understandability. In *Enterprise, Business-Process and Information Systems Modeling, 10th International Workshop, BPMDS 2009, and 14th International Conference, EMMSAD 2009, held at CAiSE 2009*, volume 29 of *LNBIP*, pages 353–366. Springer, 2009.

[FPR06] Fötsch, D., Pulvermüller, E., and Rossak, W. Modeling and verifying workflow-based regulations. In *Proceedings of the CAISE 2006 Workshop on Regulations Modelling and their Validation and Verification (ReMo2V'06)*, volume 241 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2006.

[FS10] Feja, S. and Speck, A. Checkable graphical business process representation. In *Proceedings of the 14th East European conference on Advances in Databases and Information Systems (ADBIS 2010)*, volume 6295 of *LNCS*, pages 176–189. Springer, 2010.

[FUMK06] Foster, H., Uchitel, S., Magee, J., and Kramer, J. Model-based analysis of obligations in web service choreography. In *Advanced International Conference on Telecommunications and International Conference on Internet and Web Applications and Services (AICT/ICIW 2006)*, page 149. IEEE Computer Society, 2006.

[FW99] Fan, W. and Weinstein, S. Specifying and reasoning about workflows with path constraints. In *Internet Applications, 5th International Computer Science Conference (ICSC 1999)*, volume 1749 of *LNCS*, pages 226–235. Springer, 1999.

[FWB$^+$10] Feja, S., Witt, S., Brosche, A., Speck, A., et al. Modellierung und Validierung von Datenschutzanforderungen in Prozessmodellen. In *Vernetzte IT für einen effektiven Staat – Gemeinsame Fachtagung Verwaltungsinformatik (FTVI) und Fachtagung Rechtsinformatik (FTRI) 2010*, LNI, pages 155–166. GI, 2010. In German.

[FWS11] Feja, S., Witt, S., and Speck, A. Bam: A requirements validation and verification framework for business process models. In *Proceedings of the 11th International Conference on Quality Software (QSIC 2011)*, pages 186–191. IEEE Computer Society, 2011.

[GHSW08] Governatori, G., Hoffmann, J., Sadiq, S., and Weber, I. Detecting regulatory compliance for business process models through semantic annotations. In *Business Process Management 2008 Workshops, Revised Papers*, number 17 in LNBIP, pages 5–17. Springer, 2008.

[GK07] Ghose, A. and Koliadis, G. Auditing business process compliance. In *Proceedings of the Fifth International Conference on Service-Oriented Computing (ICSOC 2007)*, volume 4749 of *LNCS*, pages 169–180. Springer, 2007.

[GLM$^+$05] Giblin, C., Liu, A., Müller, S., Pfitzmann, B., et al. Regulations expressed as logical models (REALM). In *Proceedings of the 18th Annual Conference on Legal Knowledge and Information Systems (JURIX 2005)*, volume 134 of *Frontiers in Artificial Intelligence and Applications*, pages 37–48. IOS Press, 2005.

[GM05] Governatori, G. and Milosevic, Z. Dealing with contract violations: formalism and domain specific language. In *Proceedings of the Ninth IEEE International EDOC Enterprise Computing Conference (EDOC 2005)*, pages 46–57. IEEE Computer Society, 2005.

[GM06] Governatori, G. and Milosevic, Z. A formal analysis of a business contract language. *International Journal of Cooperative Information Systems*, 15(4):659–685, 2006.

[GMP06] Giblin, C., Müller, S., and Pfitzmann, B. From regulatory policies to event monitoring rules: Towards model-driven compliance automation. Technical Report Research Report RZ-3662, IBM Research GmbH, 2006.

[GMS06] Governatori, G., Milosevic, Z., and Sadiq, S. Compliance checking between business processes and business contracts. In *Proceedings of the 10th IEEE International Enterprise Distributed Object Computing Conference (EDOC 2006)*, pages 221–232. IEEE Computer Society, 2006.

[God94] Godefroid, P. *Partial Order Methods for the Verification of Concurrent Systems – An Approach to Tackle the State-Explosion Problem.* Ph.D. thesis, Université de Liège, 1994.

[GOR11] Grambow, G., Oberhauser, R., and Reichert, M. Semantically-driven workflow generation using declarative modeling for processes in software engineering. In *Workshops Proceedings of the 15th IEEE International Enterprise Distributed Object Computing Conference (EDOCW 2011)*, pages 164–173. IEEE Computer Society, 2011.

[Gro01] Grohe, M. Generalized model-checking problems for first-order logic. In *18th Annual Symposium on Theoretical Aspects of Computer Science (STACS 2001)*, volume 2010 of *LNCS*, pages 12–26. Springer, 2001.

[Gro08] Grohe, M. Logic, graphs, and algorithms. In *Logic and Automata: History and Perspectives*, volume 2 of *Texts in Logic and Games*, pages 357–422. Amsterdam University Press, 2008.

[HMPR04] Hevner, A.R., March, S.T., Park, J., and Ram, S. Design science in information systems research. *MIS Q.*, 28(1):75–105, March 2004.

[HMZD11] Holmes, T., Mulo, E., Zdun, U., and Dustdar, S. Model-aware monitoring of SOAs for compliance. In S. Dustdar and F. Li, editors, *Service Engineering*, pages 117–136. Springer, 2011.

[HSG04] Hustadt, U., Schmidt, R.A., and Georgieva, L. A survey of decidable first-order fragments and description logics. *Journal on Relational Methods in Computer Science*, 1:251–276, 2004.

[HTZD10] Holmes, T., Tran, H., Zdun, U., and Dustdar, S. Model-driven and domain-specific architectural knowledge view for compliance meta-data in process-driven SOAs. In *Proceedings of the 2010 ICSE Workshop on Sharing and Reusing Architectural Knowledge*, pages 64–71. ACM, 2010.

[ID96] Ip, C.N. and Dill, D.L. Better verification through symmetry. *Form. Methods Syst. Des.*, 9:41–75, August 1996.

[JML09] Jacobsen, H.A., Muthusamy, V., and Li, G. The PADRES event processing network: Uniform querying of past and future events. *it - Information Technology*, pages 250–261, 2009.

[KGE11] Khaluf, L., Gerth, C., and Engels, G. Pattern-based modeling and formalizing of business process quality constraints. In *Proceedings of the 23rd International Conference on Advanced Information Systems Engineering (CAiSE 2011)*, volume 6741 of *LNCS*, pages 521–535. Springer, 2011.

[Kle91] Klein, J. Advanced rule driven transaction management. In *Compcon Spring '91. Digest of Papers*, pages 562–567. IEEE, 1991.

[KLRM$^+$10] Knuplesch, D., Ly, L.T., Rinderle-Ma, S., Pfeifer, H., et al. On enabling data-aware compliance checking of business process models. In *Proceedings of the 29th International Conference on Conceptual Modeling (ER 2010)*, number 6412 in LNCS, pages 332–346. Springer, 2010.

[KMSA08] Kharbili, M.E., de Medeiros, A.K.A., Stein, S., and van der Aalst, W.M.P. Business process compliance checking: Current state and future challenges. In *Modellierung betrieblicher Informationssysteme – Modellierung zwischen SOA und Compliance Management*, volume 141 of *LNI*, pages 107–113. GI, 2008.

[Knu08] Knuplesch, D. *Verifikation von ADEPT-Prozessen mittels Model Checking.* Master's thesis, University of Ulm, 2008. In German.

[KR11a] Knuplesch, D. and Reichert, M. Ensuring business process compliance along the process life cycle. Technical Report UIB-2011-06, University of Ulm, Ulm, July 2011.

[KR11b] Künzle, V. and Reichert, M. PHILharmonicFlows: towards a framework for object-aware process management. *Journal of Software Maintenance and Evolution: Research and Practice*, 23(4):205–244, June 2011.

[KRG07] Küster, J., Ryndina, K., and Gall, H. Generation of business process models for object life cycle compliance. In *Proceedings of the 5th International Conference on Business Process Management (BPM 2007)*, volume 4714 of *LNCS*, pages 165–181. Springer, 2007.

[Kri63] Kripke, S.A. Semantical considerations on modal logic. *Acta Philosophica Fennica*, 16(1963):83–94, 1963.

[KS08] Kharbili, M.E. and Stein, S. Compliance Management auf Basis von semantischen Richtlinien. volume XI of *Leipziger Beiträge zur Informatik*, pages 253–262. Eigenverlag Leipziger Informatik-Verbund (LIV), Leipzig, September 2008.

[KSMP08a] Kharbili, M.E., Stein, S., Markovic, I., and Pulvermüller, E. Towards a framework for semantic business process compliance management. In *Proceedings of the 1st International Workshop on Governance, Risk and Compliance – Applications in Information Systems (GRCIS'08) held in conjunction with the CAiSE 2008 Conference*. 2008.

[KSMP08b] Kharbili, M.E., Stein, S., Markovic, I., and Pulvermüller, E. Towards policy-powered semantic enterprise compliance management. In *In 3rd International Workshop on Semantic Business Process Management*, pages 16–21. 2008.

[KSP08] Kharbili, M.E., Stein, S., and Pulvermüller, E. Policy-based semantic compliance checking for business process management. In *Proceedings of the Workshops colocated with the MobIS2008 Conference: Including EPK2008, KobAS2008 and ModKollGP2008*, number 420 in CEUR Workshop Proceedings, pages 178–192. CEUR-WS.org, 2008.

[Lan08] Lanz, A. *Realisierung einer Zeitmanagementkomponente eines adaptiven Prozess-Management-Systems*. Master's thesis, University of Ulm, 2008. In German.

[LGRMD08] Ly, L.T., Göser, K., Rinderle-Ma, S., and Dadam, P. Compliance of semantic constraints - a requirements analysis for process management systems. In *Proceedings of the 1st International Workshop on Governance, Risk and Compliance – Applications in Information Systems (GRCIS'08) held in conjunction with the CAiSE 2008 Conference*. 2008.

[LIMRM12] Ly, L.T., Indiono, C., Mangler, J., and Rinderle-Ma, S. Data transformation and semantic log purging for process mining. In *Proceedings of the 24th International Conference on Advanced Information Systems Engineering (CAiSE 2012)*, volume 7328 of *LNCS*, pages 238–253. Springer, 2012.

[LKRM$^+$10] Ly, L.T., Knuplesch, D., Rinderle-Ma, S., Göser, K., et al. SeaFlows Toolset – compliance verification made easy for process-aware information systems. In *Information Systems Evolution – CAiSE Forum 2010, Hammamet, Tunisia, June 7-9, 2010, Selected Extended Papers*, volume 72 of *LNBIP*, pages 76–91. Springer, 2010.

[LMX07] Liu, Y., Müller, S., and Xu, K. A static compliance-checking framework for business process models. *IBM Systems Journal*, 46(2):335–361, 2007.

[Loh11] Lohmann, N. Compliance by design for artifact-centric business processes. In *Proceedings of the 9th International Conference on Business Process Management (BPM 2011)*, volume 6896 of *LNCS*, pages 99–115. Springer, 2011.

[LR00] Leymann, F. and Roller, D. *Production Workflow: Concepts and Techniques*. Prentice Hall, 2000.

[LR07] Lenz, R. and Reichert, M. IT support for healthcare processes – premises, challenges, perspectives. *Data Knowl. Eng.*, 61(1):39–58, 2007.

[LRD06] Ly, L.T., Rinderle, S., and Dadam, P. Semantic correctness in adaptive process management systems. In *Proceedings of the 4th International Conference on Business Process Management (BPM 2006)*, volume 4102 of *LNCS*, pages 193–208. Springer, 2006.

[LRD08] Ly, L.T., Rinderle, S., and Dadam, P. Integration and verification of semantic constraints in adaptive process management systems. *Data Knowl. Eng.*, 64(1):3–23, 2008.

[LRDR06] Ly, L.T., Rinderle, S., Dadam, P., and Reichert, M. Mining staff assignment rules from event-based data. In *Business Process Management Workshops, BPM 2005 International Workshops, BPI, BPD, ENEI, BPRM, WSCOBPM, BPS, Nancy, France, September 5, 2005, Revised Selected Papers*, volume 3812 of *LNCS*, pages 177–190. Springer, 2006.

[LRMD10] Ly, L.T., Rinderle-Ma, S., and Dadam, P. Design and verification of instantiable compliance rule graphs in process-aware information systems. In *Proceedings of the 22nd International Conference on Advanced Information Systems Engineering (CAiSE 2010)*, volume 6051 of *LNCS*, pages 9–23. Springer, 2010.

[LRMGD12] Ly, L.T., Rinderle-Ma, S., Göser, K., and Dadam, P. On enabling integrated process compliance with semantic constraints in process management systems – requirements, challenges, solutions. *Information Systems Frontiers*, 14(2):195–219, 2012.

[LRMKD11] Ly, L.T., Rinderle-Ma, S., Knuplesch, D., and Dadam, P. Monitoring business process compliance using compliance rule graphs. In *On the Move to Meaningful Internet Systems: OTM Confederated International Conferences CoopIS, DOA-SVI, and ODBASE 2011, Proceedings, Part I*, volume 7044 of *LNCS*, pages 82–99. Springer, 2011.

[LRY$^+$04] Liu, Y.A., Rothamel, T., Yu, F., Stoller, S.D., et al. Parametric regular path queries. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, pages 219–230. ACM, 2004.

[LSG07] Lu, R., Sadiq, S., and Governatori, G. Compliance aware process design. In *Business Process Management 2007 Workshops, BPI, BPD, CBP, ProHealth, RefMod, semantics4ws, Revised Selected Papers*, volume 4928 of *LNCS*, pages 120–131. Springer, 2007.

[LWR09] Lanz, A., Weber, B., and Reichert, M. Time patterns for process-aware information systems: A pattern-based analysis – revised version. Technical report, University of Ulm, 2009.

[LWR10] Lanz, A., Weber, B., and Reichert, M. Workflow time patterns for process-aware information systems. In *Proceedings Enterprise, Business Process, and Information Systems Modelling: 11th International Workshop BPMDS and 15th International Conference EMMSAD at CAiSE 2010*, LNBIP, pages 94–107. Springer, 2010.

[May81] Mayr, E.W. An algorithm for the general Petri net reachability problem. In *Proceedings of the 13th Annual ACM Symposium on Theory of Computing (STOC 1981)*, pages 238–246. ACM, 1981.

[MDK+03] Mukherjee, S., Davulcu, H., Kifer, M., Senkul, P., et al. Logic based approaches to workflow modeling and verification. In *Logics for Emerging Applications of Databases*, pages 167–202. 2003.

[Men07] Mendling, J. *Detection and Prediction of Errors in EPC Business Process Models.* Ph.D. thesis, Institute of Information Systems and New Media, Vienna University of Economics and Business Administration (WU Wien), Austria, 2007.

[Mer10] Merkel, P. *Anwender- und aufgabenzentrierte Auswertung der Erfüllung semantischer Constraints in Prozess-Management-Systemen.* Master's thesis, University of Ulm, 2010. In German.

[MMC+11] Montali, M., Maggi, F., Chesani, F., Mello, P., et al. Monitoring business constraints with the event calculus. Technical report, Universita degli Studi di Bologna, 2011.

[MMWA11] Maggi, F., Montali, M., Westergaard, M., and van der Aalst, W. Monitoring business constraints with linear temporal logic: An approach based on colored automata. In *Proceedings of the 9th International Conference on Business Process Management (BPM 2011)*, volume 6896 of *LNCS*, pages 132–147. Springer, 2011.

[MPA+10] Montali, M., Pesic, M., Aalst, W.M.P.v.d., Chesani, F., et al. Declarative specification and verification of service choreographiess. *ACM Trans. Web*, 4(1):3:1–3:62, January 2010.

[MRM11] Mangler, J. and Rinderle-Ma, S. IUPC: Identification and unification of process constraints. *CoRR*, abs/1104.3609, 2011.

[MS03] Mangan, P. and Sadiq, S. A constraint specification approach to building flexible workflows. *Journal of Research and Practice in Information Technology*, 35(1):21–39, 2003.

[MW06] Medeiros, A.K.A.D. and Weijters, A.J.M.M. Process equivalence: Comparing two process models based on observed behavior. In *Proceedings of the 4th International Conference on Business Process Management (BPM 2006)*, volume 4102 of *LNCS*, pages 129–144. Springer, 2006.

[MWMA11] Maggi, F.M., Westergaard, M., Montali, M., and van der Aalst, W.M.P. Runtime verification of LTL-based declarative process models. In *Second International Conference on Runtime Verification (RV 2011), Revised Selected Papers*, volume 7186 of *LNCS*, pages 131–146. Springer, 2011.

[Mül09] Müller, D. *Management datengetriebener Prozessstrukturen.* Ph.D. thesis, University of Ulm, 2009. In German.

[Nam08] Namiri, K. *Model-Driven Management of Internal Controls for Business Process Compliance.* Ph.D. thesis, Universität Fridericiana zu Karlsruhe, 2008.

[NS07a] Namiri, K. and Stojanovic, N. A formal approach for internal controls compliance in business processes. In *8th Workshop on Business Process Modeling, Development, and Support (BPMDS 2007).* 2007.

[NS07b] Namiri, K. and Stojanovic, N. A model-driven approach for internal controls compliance in business processes. In *Proceedings of the Workshop on Semantic Business Process and Product Lifecycle Management (SBPM 2007)*, volume 251 of *CEUR Workshop Proceedings*, pages 40–44. CEUR-WS.org, 2007.

[NS07c] Namiri, K. and Stojanovic, N. Pattern-based design and validation of business process compliance. In *On the Move to Meaningful Internet Systems 2007: OTM Confederated International Conferences CoopIS, DOA, ODBASE, GADA, and IS 2007, Proceedings, Part I*, volume 4803 of *LNCS*, pages 59–76. Springer, 2007.

[NS08] Namiri, K. and Stojanovic, N. Towards a formal framework for business process compliance. In *Multikonferenz Wirtschaftsinformatik (MKWI 2008).* GITO-Verlag, Berlin, 2008.

[OMG08] OMG. Semantics of business vocabulary and business rules (SBVR), v1.0, 2008.

[OMG11] OMG. Business process model and notation (BPMN) – version 2.0, January 2011.

[PA06] Pesic, M. and van der Aalst, W.M.P. A declarative approach for flexible business processes management. In J. Eder and S. Dustdar, editors, *Business Process Management 2006 Workshops, BPD, BPI, ENEI, GPWW, DPM, semantics4ws*, volume 4103 of *LNCS*, pages 169–180. Springer, 2006.

[Pel96] Peled, D. Partial order reduction: Model-checking using representatives. In *Proceedings of the 21st International Symposium on Mathematical Foundations of Computer Science*, volume 1113 of *LNCS*, pages 93–112. Springer, 1996.

[Pel98] Peled, D. Ten years of partial order reduction. In *Computer Aided Verification*, volume 1427 of *LNCS*, pages 17–28. Springer, 1998.

[Pes08] Pesic, M. *Constraint-Based Workflow Management Systems: Shifting Control to Users.* Ph.D. thesis, Eindhoven University of Technology, 2008.

[PSSA07] Pesic, M., Schonenberg, M., Sidorova, N., and van der Aalst, W. Constraint-based workflow models: Change made easy. In *On the Move to Meaningful Internet Systems 2007: OTM Confederated International Conferences CoopIS, DOA,* *ODBASE, GADA, and IS 2007, Proceedings, Part I*, volume 4803 of *LNCS*, pages 77–94. Springer, 2007.

[RA08] Rozinat, A. and van der Aalst, W.M.P. Conformance checking of processes based on monitoring real behavior. *Inf. Syst.*, 33(1):64–95, 2008.

[Ram30] Ramsey, F.P. On a problem of formal logic. *Proceedings of the London Mathematical Society*, s2-30(1):264–286, 1930.

[RD98] Reichert, M. and Dadam, P. ADEPTflex – supporting dynamic changes of workflows without losing control. *Journal of Intelligent Information Systems, Special Issue on Workflow Management Systems*, 10(2):93–129, 1998.

[Rei00] Reichert, M. *Dynamische Ablaufänderungen in Workflow-Management-Systemen.* Ph.D. thesis, University of Ulm, 2000. In German.

[Rin04] Rinderle, S. *Schema Evolution in Process Management Systems.* Ph.D. thesis, University of Ulm, 2004.

[RKBB12] Reichert, M., Kolb, J., Bobrik, R., and Bauer, T. Enabling personalized visualization of large business processes through parameterizable views. In *27th ACM Symposium On Applied Computing (SAC'12), 9th Enterprise Engineering Track*, pages 1653–1660. ACM, 2012.

[RKG06] Ryndina, K., Küster, J.M., and Gall, H. Consistency of business process models and object life cycles. In *Models in Software Engineering, Workshops and Symposia at MoDELS 2006, Reports and Revised Selected Papers*, volume 4364 of *LNCS*, pages 80–90. Springer, 2006.

[RMLD08] Rinderle-Ma, S., Ly, L.T., and Dadam, P. Business Process Compliance (Aktuelles Schlagwort). *EMISA Forum*, pages 24–29, 2008. In German.

[RMM11] Rinderle-Ma, S. and Mangler, J. Integration of process constraints from heterogeneous sources in process-aware information systems. In *Proceedings of the 4th International Workshop on Enterprise Modelling and Information Systems Architectures (EMISA 2011)*, pages 51–64. 2011.

[RMR09] Rinderle-Ma, S. and Reichert, M. Comprehensive life cycle support for access rules in information systems: The CEOSIS project. *Enterprise Information Systems*, 3(3):219–251, 2009.

[Ros11] Ross, R. How strongly should a business rule be enforced? 02 2011. Http://www.modernanalyst.com/Resources/Articles/tabid/115/articleType/ArticleView/articleId/1637/How-Strongly-Should-a-Business-Rule-Be-Enforced.aspx. Last visited: 2012-11-20.

[RSDC10] Rodríguez, C., Silveira, P., Daniel, F., and Casati, F. Analyzing compliance of service-based business processes for root-cause analysis and prediction. In *Current Trends in Web Engineering – 10th International Conference on Web Engineering, ICWE 2010 Workshops, Revised Selected Papers*, volume 6385 of *LNCS*, pages 277–288. Springer, 2010.

[RWMR13] Rabbi, F., Wang, H., Maccaull, W., and Rutle, A. A model slicing method for workflow verification. *Electron. Notes Theor. Comput. Sci.*, 295:79–93, May 2013.

[RWRW05] Rinderle, S., Weber, B., Reichert, M., and Wild, W. Integrating process learning and process evolution – a semantics based approach. In *Proceedings of the 3rd International Conference on Business Process Management (BPM 2005)*, volume 3649 of *LNCS*, pages 252–267. Springer, 2005.

[SAL+10] Schumm, D., Anstett, T., Leymann, F., Schleicher, D., et al. Essential aspects of compliance management with focus on business process automation. In *Proceedings of the 3rd International Conference on Business Processes and Services Computing (BPSC) – INFORMATIK 2010*, volume P-177 of *LNI*, pages 127–138. GI, 2010.

[Sch03] Schneider, K. *Verification of Reactive Systems.* Springer, 2003.

[SFV+12] Santos, E.A.P., Francisco, R., Vieira, A.D., F.R. Loures, E., et al. Modeling business rules for supervisory control of process-aware information systems. In *Business Process Management 2011 Workshops, Revised Selected Papers, Part II*, volume 100 of *LNBIP*, pages 447–458. Springer, 2012.

[SGN07] Sadiq, S., Governatori, G., and Naimiri, K. Modeling control objectives for business process compliance. In *Proceedings of the 5th International Conference on Business Process Management (BPM 2007)*, volume 4714 of *LNCS*, pages 149–164. Springer, 2007.

[Sin96] Singh, M.P. Semantical considerations on workflows: An algebra for intertask dependencies. In *Proceedings of the Fifth International Workshop on Database Programming Languages*, Electronic Workshops in Computing, page 5. Springer, 1996.

[Sin97] Singh, M. Formal aspects of workflow management, part 1: Semantics. Technical report, Raleigh, NC, USA, 1997.

[SKGL08] Sackmann, S., Kähmer, M., Gilliot, M., and Lowis, L. A classification model for automating compliance. In *Proceedings of the 2008 10th IEEE Conference on E-Commerce Technology and the Fifth IEEE Conference on Enterprise Computing, E-Commerce and E-Services*, pages 79–86. IEEE Computer Society, 2008.

[SLM+10] Schumm, D., Leymann, F., Ma, Z., Scheibler, T., et al. Integrating compliance into business processes: Process fragments as reusable compliance controls. In *Proceedings of the Multikonferenz Wirtschaftsinformatik, MKWI 2010, 23-25 February 2010, Göttingen, Germany.* Universitätsverlag Göttingen, 2010.

[SO00] Sadiq, W. and Orlowska, M.E. Analyzing process models using graph reduction techniques. *Inf. Syst.*, 25(2):117–134, 2000.

[SOS05] Sadiq, S., Orlowska, M., and Sadiq, W. Specification and validation of process constraints for flexible workflows. *Inf. Syst.*, 30(5):349–378, 2005.

[TBJ+03] Tonti, G., Bradshaw, J., Jeffers, R., Montanari, R., et al. Semantic web languages for policy representation and reasoning: A comparison of KAoS, Rei, and Ponder. In *Proceedings of the Second International Semantic Web Conference (ISWC 2003)*, volume 2870 of *LNCS*, pages 419–437. Springer, 2003.

[TEHP12] Türetken, O., Elgammal, A., van den Heuvel, W.J., and Papazoglou, M.P. Capturing compliance requirements: A pattern-based approach. *IEEE Software*, 29(3):28–36, 2012.

[The08] The COMPAS consortium. State-of-the-art in the field of compliance languages. Technical report, COMPAS EU project, 2008.

[The09a] The COMPAS consortium. Design of compliance language run-time environment and architecture. Technical report, COMPAS EU project, 2009.

[The09b] The COMPAS consortium. Initial specification of compliance language constructs and operators. Technical report, COMPAS EU project, 2009.

[The10a] The COMPAS consortium. Implementation of an integrated prototype handling interactive user specified compliance request in a compliance language. Technical report, COMPAS EU project, 2010.

[The10b] The COMPAS consortium. Implementation of an integrated prototype handling interactive user specified compliance requests in a compliance language. Technical report, COMPAS EU project, 2010.

[The10c] The COMPAS consortium. Initial implementation of a compliance language. Technical report, COMPAS EU project, 2010.

[The11] The COMPAS consortium. Compliance-driven models, languages, and architectures for services – achievements and lessons learned. Technical report, COMPAS EU project, 2011.

[THO+10] Tran, H., Holmes, T., Oberortner, E., Mulo, E., et al. An end-to-end framework for business compliance in process-driven SOAs. In *Proceedings of the 12th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC 2010)*, pages 407–414. IEEE Computer Society, 2010.

[Tip95] Tip, F. A survey of program slicing techniques. *Journal of programming languages*, 3(3):121–189, 1995.

[Tur37] Turing, A. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, s2-42(1):230–265, 1937.

[Val96] Valmari, A. The state explosion problem. In *Lectures on Petri Nets I: Basic Models, Advances in Petri Nets*, volume 1491 of *LNCS*, pages 429–528. Springer, 1996.

[Var01] Vardi, M. Branching vs. linear time: Final showdown. In *Proceedings of the 2001 Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2001)*, volume 2031 of *LNCS*, pages 1–22. Springer, 2001.

[VBDA10] Verbeek, H.M.W., Buijs, J.C.A.M., van Dongen, B.F., and van der Aalst, W.M.P. XES, XESame, and ProM 6. In *Information Systems Evolution – CAiSE Forum 2010, Selected Extended Papers*, volume 72 of *LNBIP*, pages 60–75. Springer, 2010.

[Wes98] Weske, M. Flexible modeling and execution of workflow activities. In *Proceedings of the Thirty-First Annual Hawaii International Conference on System Sciences – Volume 7*, pages 713–722. IEEE Computer Society, Washington, DC, USA, 1998.

[Wes07] Weske, M. *Business Process Management – Concepts, Languages, Architectures.* Springer, 2007.

[WGH08] Weber, I., Governatori, G., and Hoffmann, J. Approximate compliance checking for annotated process models. In *Proceedings of the 1st International Workshop on Governance, Risk and Compliance – Applications in Information Systems (GRCIS'08)*, pages 46–60. 2008.

[WKH08a] Wörzberger, R., Kurpick, T., and Heer, T. Checking correctness and compliance of integrated process models. In *Proceedings of the 10th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC 2008)*, pages 576–583. IEEE Computer Society, 2008.

[WKH08b] Wörzberger, R., Kurpick, T., and Heer, T. On correctness, compliance and consistency of process models. In *Proceedings of the 17th IEEE International Workshops on Enabling Technologies: Infrastructures for Collaborative Enterprises (WETICE 2008)*, pages 251–252. IEEE Computer Society, 2008.

[WLB03] Wainer, J. and de Lima Bezerra, F. Constraint-based flexible workflows. In *Proceedings of the 9th International Workshop Groupware: Design, Implementation, and Use*, volume 2806 of *LNCS*, pages 151–158. Springer, 2003.

[WLBB04] Wainer, J., de Lima Bezerra, F., and Barthelmess, P. Tucupi: a flexible workflow system based on overridable constraints. In *Proceedings of the 2004 ACM Symposium on Applied Computing (SAC'04)*, pages 498–502. 2004.

[WPD$^+$11] Weidlich, M., Polyvyanyy, A., Desai, N., Mendling, J., et al. Process compliance analysis based on behavioural profiles. *Inf. Syst.*, 36(7):1009–1025, 2011.

[WPDM10] Weidlich, M., Polyvyanyy, A., Desai, N., and Mendling, J. Process compliance measurement based on behavioural profiles. In *Proceedings of the 22nd International Conference on Advanced Information Systems Engineering (CAiSE 2010)*, volume 6051 of *LNCS*, pages 499–514. Springer, 2010.

[WSB07] Web Services Business Process Execution Language Version 2.0 – OASIS standard, April 2007.

[WSR09] Weber, B., Sadiq, S., and Reichert, M. Beyond rigidity – dynamic process lifecycle support. *Computer Science – Research and Development*, 23:47–65, 2009.

[WZM$^+$11] Weidlich, M., Ziekow, H., Mendling, J., Günther, O., et al. Event-based monitoring of process execution violations. In *Proceedings of the 9th International Conference on Business Process Management (BPM 2011)*, volume 6896 of *LNCS*, pages 182–198. Springer, 2011.

[XLW08] Xu, K., Liu, Y., and Wu, C. BPSL modeler – visual notation language for intuitive business property reasoning. *Electr. Notes Theor. Comput. Sci.*, 211:211–220, 2008.

[YMHJ06] Yu, J., Manh, T.P., Hand, J., and Jin, Y. Pattern-based property specification and verification for service composition. CeCSES Report SUT.CeCSES-TR010, Swinburne University of Technology, 2006.

List of Figures

List of Tables

List of Definitions

List of Execution Rules

List of Marking Rules

A Appendix

A.1. Criteria for preventing structural redundancy

Definition A.1 (Structural redundancy)
Let $R = (A, C)$ be a syntactically correct CRG. Let A^+ be A without **AnteAbs** nodes and associated edges with $OrderE_{A^+}$ being the set of ORDER edges of A^+ and let C^+ be C without **ConsAbs** nodes and associated edges with $OrderE_{C^+}$ being the set of ORDER edges of C^+. Let further R^+ be R after removal of **AnteAbs** and **ConsAbs** nodes and associated edges. Then, R is considered free of structural redundancy iff the following conditions hold:

i) $\forall s \forall t \in N_R$ with

- $(s, t) \in OrderE_R$ or
- $\exists n_1, \ldots, \exists n_k \in N_R$ such that
  $\exists e_1, \ldots, \exists e_{k+1} \in OrderE_R$ with $e_1 = (s, n_1), \ldots, e_{k+1} = (n_k, t)$

holds:

- $\neg(\exists e' \in DiffE_R$ with $e' = (s, t) \vee e' = (t, s))$.

ii) $\forall s \forall t \in N_{A^+}$ with

- $\exists n_1, \ldots, \exists n_k \in N_{A^+}$ such that
  $\exists e_1, \ldots, \exists e_{k+1} \in OrderE_{A^+}$ with $e_1 = (s, n_1), \ldots, e_{k+1} = (n_k, t)$

holds:

- $(s, t) \notin OrderE_R$.

iii) $\forall s \forall t \in N_{C+}$ with

- $\exists n_1, \ldots, \exists n_k \in N_{C+}$ such that
  $\exists e_1, \ldots, \exists e_{k+1} \in OrderE_{C+}$ with $e_1 = (s, n_1), \ldots, e_{k+1} = (n_k, t)$

holds:

- $(s, t) \notin OrderE_R$.

iv) $\forall s \forall t \in N_{R+}$ with

- $(nt_R(s) = \texttt{AnteOcc} \wedge nt_R(t) = \texttt{ConsOcc}) \vee (nt_R(s) = \texttt{ConsOcc} \wedge nt_R(t) = \texttt{AnteOcc})$ and
- $\exists n_1, \ldots, \exists n_k \in N_{R+}$ such that
  $\exists e_1, \ldots, \exists e_{k+1} \in OrderE_{R+}$ with $e_1 = (s, n_1), \ldots, e_{k+1} = (n_k, t)$

holds:

- $(s, t) \notin OrderE_R$.

vi) $\forall s \forall t \in N_R$ with

- $nt_R(s) \in \{\texttt{AnteAbs}, \texttt{ConsAbs}\} \vee nt_R(t) \in \{\texttt{AnteAbs}, \texttt{ConsAbs}\}$ and
- $\exists n_1, \ldots, \exists n_k \in N_R$ such that
  $\exists e_1, \ldots, \exists e_{k+1} \in OrderE_R$ with $e_1 = (s, n_1), \ldots, e_{k+1} = (n_k, t)$

holds:

- $(s, t) \notin OrderE_R$.

A.2. Execution and marking rules for EX events

A.2.1. Necessary conditions

The necessary execution conditions for EX events largely resemble those for START events. Similar to the necessary execution conditions for START events, absence nodes also become executable over a matching EX event even if they are already `Started`. This enables dealing with execution traces containing both EX event as well as START and END events.

Definition A.2 (Necessary conditions for executing CRG nodes over EX events)
Let $R = (A, C)$ be a CRG and $e \in E^{Ex}$ be an EX event. Let further $match_R : N_R \times E^{Ex} \to \mathbb{B}$ be a function returning `true` if the given EX event satisfies the all conditions associated with the given node and `false` otherwise.

Then,

- $executableAnteAE_R : NS_A^* \times N_A \times E^{Ex} \to \mathbb{B}$ is a function determining whether an antecedent node n is executable under an `AnteStateMark` ns_A over e with:

  $\forall n \in N_A$ with $nt_R(n) = \texttt{AnteOcc}$:

$$executableAnteAE_R(ns_A, n, e) := \begin{cases} true,\ if \\ \ (i)\ ns_A(n) = \texttt{Null} \wedge \\ \ (ii)\ match_R(n, e) = true \wedge \\ \ (iii)\ \forall l \in predAnteOcc(n) : ns_A(l) = \texttt{Completed} \\ false,\ otherwise. \end{cases}$$

  $\forall n \in N_A$ with $nt_R(n) = \texttt{AnteAbs}$:

$$executableAnteAE_R(ns_A, n, e) := \begin{cases} true,\ if \\ \ (i)\ ns_A(n) \in \{\texttt{Null}, \texttt{Started}\} \wedge \\ \ (ii)\ match_R(n, e) = true \wedge \\ \ (iii)\ \forall l \in predAnteOcc(n) : ns_A(l) = \texttt{Completed} \\ false,\ otherwise. \end{cases}$$

- $executableConsAE_R : NS_R^* \times N_C \times E^{Ex} \to \mathbb{B}$ is a function determining whether a consequence node n is executable under a `StateMark` (ns_A, ns_C) over e with:

  $\forall n \in N_C$ with $nt_R(n) = \texttt{ConsOcc}$:

$$executableConsAE_R((ns_A, ns_C), n, e) := \begin{cases} \texttt{true},\ if \\ \ (i)\ ns_C(n) = \texttt{Null} \wedge \\ \ (ii)\ match_R(n, e) = true \wedge \\ \ (iii)\ \forall l \in predAnteOcc(n) : \\ \quad ns_A(l) = \texttt{Completed} \\ \ (iv)\ \forall l \in predConsOcc(n) : \\ \quad ns_C(l) = \texttt{Completed} \\ \texttt{false},\ otherwise. \end{cases}$$

$\forall n \in N_C$ with $nt_R(n) = \texttt{ConsAbs}$:

$$executableConsAE_R((ns_A, ns_C), n, e) := \begin{cases} \texttt{true}, \; if \\ \;(i)\; ns_C(n) \in \{\texttt{Null}, \texttt{Started}\} \wedge \\ \;(ii)\; match_R(n, e) = true \wedge \\ \;(iii)\; \forall l \in predAnteOcc(n): \\ \quad ns_A(l) = \texttt{Completed} \\ \;(iv)\; \forall l \in predConsOcc(n): \\ \quad ns_C(l) = \texttt{Completed} \\ \texttt{false}, \; otherwise. \end{cases}$$

- $exAnteNodesAE_R : NS_A^* \times E^{Ex} \times \{\texttt{AnteOcc}, \texttt{AnteAbs}\} \rightarrow \mathcal{P}(N_R)$
  is a function determining the set of antecedent nodes of node type t that are executable under the given `AnteStateMark` ns_A over e with:

  $exAnteNodesAE_R(ns_A, e, t) :=$
  $\{n \in N_R \mid nt_R(n) = t \;\wedge\; executableAnteAE_R(ns_A, n, e) = \texttt{true}\}.$
- $exConsNodesAE_R : NS_R^* \times E^{Ex} \times \{\texttt{ConsOcc}, \texttt{ConsAbs}\} \rightarrow \mathcal{P}(N_R)$

  is a function determining the set of consequence nodes of node type t that are executable under the given `StateMark` (ns_A, ns_C) over e with:

  $exConsNodesAE_R((ns_A, ns_C), e, t) :=$
  $\{n \in N_R \mid nt_R(n) = t \;\wedge\; executableConsAE_R((ns_A, ns_C), n, e) = \texttt{true}\}.$

A.2.2. Execution and marking rules for atomic execution events

In the style of Algorithm 2 for processing START events, Algorithm 5 executes a MarkStructure over an EX event. The functions *executeAnteOccAE*, *executeAnteAbsAE*, *executeConsOccAE*, and *executeConsAbsAE* are defined below.

Algorithm 5 Executing a MarkStructure over an EX event ($executeAE_R(ms, e)$)

```
1: $R = (A, C)$ is a CRG;
2: $e \in E^{Ex}$ is an EX event;
3: $ms = ((ns_A, nl_A), \{(ns_C^1, nl_C^1), \ldots, (ns_C^k, nl_C^k)\}) \in MS_R^*$ is a MarkStructure of $R$;
4: $M_{ms} = \{m_1 = ((ns_A, nl_A), (ns_C^1, nl_C^1)), \ldots, m_k = ((ns_A, nl_A), (ns_C^k, nl_C^k))\}$ is the set of
   ExMarks of $ms$;

   {INITIALIZATION}
5: $M_{AnteOcc}, M_{ConsOcc}, M_{Res} = \emptyset$; $MS_{Res} = \emptyset$ {Set global variables $QAO, QAA, QCO$, and
   $QCA$ representing the sets of executable nodes in this iteration}
6: $QAO := exAnteNodesAE_R(ns_A, e, \texttt{AnteOcc})$;
7: $QAA := exAnteNodesAE_R(ns_A, e, \texttt{AnteAbs})$;
8: $QCO_{m_i} := exConsNodesAE_R((ns_A, ns_C^i), e, \texttt{ConsOcc}), i = 1, \ldots, k$;
9: $QCA_{m_i} := exConsNodesAE_R((ns_A, ns_C^i), e, \texttt{ConsAbs}), i = 1, \ldots, k$;

   {ITERATION}
10: for all $m \in M_{ms}$ do
11:    $M_{AnteOcc} = executeAnteOccAE_R(m, e)$;
12:    for all $m \in M_{AnteOcc}$ do
13:       $m_{AnteAbs} = executeAnteAbsAE_R(m, e)$;
14:       $M_{ConsOcc} = executeConsOccAE_R(m_{AnteAbs}, e)$;
15:       for all $m \in M_{ConsOcc}$ do
16:          $m_{ConsAbs} = executeConsAbsAE_R(m, e)$;
17:          $M_{Res} = M_{Res} \cup \{m_{ConsAbs}\}$;
18:       end for
19:    end for
20: end for

    {Aggregation of obtained ExMarks to MarkStructures }
21: $MS_{res} = aggregate_R(M_{Res})$;

    {The resulting set of MarkStructures is returned}
22: return $MS_{res}$;
```

A.2.2.1. Execution of AnteOcc nodes

AnteOcc nodes are executed over EX event in a similar manner as they are executed over START events. To correctly implement the semantics of DIFF edges, it has to be ensured that the executed AnteOcc nodes are not connected to each other through any DIFF edges.

Execution Rule ER9 (Execution of AnteOcc nodes over EX events):
Let $R = (A, C)$ be a CRG. Let further
$Q^* := \{Q \subseteq QAO \mid \forall n_1 \forall n_2 \in Q : (n_1, n_2) \notin DiffE_R \wedge (n_2, n_1) \notin DiffE_R\}$.
It is ensured that each $Q \in Q^*$ is free of nodes that are directly connected through a DIFF edge. Then,

- $executeAnteOccAE_R : M_R^* \times E^{Ex} \rightarrow 2^{M_R^*}$
  is a function assigning child ExMarks to an ExMark m of R and an EX event e with:

  $executeAnteOccAE_R(m, e) := \bigcup_{Q \in Q^*} markAnteOccAE_R(m, Q, e)$.

Marking Rule MR11 (Marking of AnteOcc nodes over EX events):
Let $R = (A, C)$ be a CRG and $m = ((ns_A, nl_A), (ns_C, nl_C)) \in M_R^*$ be an ExMark of R. Let further $Q \subseteq N_R$ be a set of AnteOcc nodes in R. Then,

$markAnteOccAE_R : M_R^* \times \mathcal{P}(N_R) \times E^{Ex} \rightarrow M_R^*$ is a function assigning an ExMark m' of R to an original ExMark m of R and a set of AnteOcc nodes Q to be executed, with

$markAnteOccAE_R(m, Q, e) := m' = ((ns'_A, nl'_A), (ns'_C, nl'_C))$ with:

- $\forall n \in N_R$ with $nt_R(n) = $ AnteOcc:

$$(ns'_A(n), nl'_A(n)) := \begin{cases} (\text{Completed}, \emptyset) \; if \; n \in Q \\ (ns_A(n), nl_A(n)), \; otherwise. \end{cases}$$

- $\forall n \in N_R$ with $nt_R(n) = $ AnteAbs:

$$(ns'_A(n), nl'_A(n)) := \begin{cases} (\text{NotExecuted}, \emptyset) \; if \; n \in deadAnteAbs_R(ns_A, Q) \\ (ns_A(n), nl_A(n)), \; otherwise. \end{cases}$$

- $\forall n \in N_R$ with $nt_R(n) = $ ConsOcc:

$$(ns'_C(n), nl'_C(n)) := \begin{cases} (\text{NotExecuted}, \emptyset) \; if \; n \in deadConsOcc_R(ns_C, Q) \\ (ns_C(n), nl_C(n)), \; otherwise. \end{cases}$$

- $\forall n \in N_R$ with $nt_R(n) = $ ConsAbs:

$$(ns'_C(n), nl'_C(n)) := \begin{cases} (\text{NotExecuted}, \emptyset) \; if \; n \in deadConsAbs_R(ns_C, Q) \\ (ns_C(n), nl_C(n)), \; otherwise. \end{cases}$$

- For m' we define attributes as follows:
  - $ExAnteOcc_{m'} := Q$,

– $QCO_{m'} := QCO_m$,

– $QCA_{m'} := QCA_m$.

A.2.2.2. Execution of AnteAbs nodes

AnteAbs nodes that become executable in an iteration (and that are not yet discarded and not in a firing conflict with AnteOcc nodes executed in the very iteration) are marked as Completed. As a result, the complete ExMark and, thus, the corresponding MarkStructure becomes DEACTIVATED and can be discarded from further execution (cf. Section 7.5.1).

Execution Rule ER10 (Execution of AnteAbs nodes over EX events):
Let $R = (A, C)$ be a CRG and $m = ((ns_A, nl_A), (ns_C, nl_C)) \in M_R^*$ be an ExMark of R. Further,

- $D := \{n \in QAA_m \mid \exists l \in ExAnteOcc_m : \ (n, l) \in DiffE_R \lor (l, n) \in DiffE_R\}$ denotes the subset of QAA_m containing those nodes that are connected through a DIFF edge to AnteOcc nodes executed in this iteration.
- $NotExAnteAbs_m := \{n \in QAA_m \mid ns_A(n) = \text{NotExecuted}\}$ denotes the subset of QAA containing nodes already marked as NotExecuted.
- $Q := QAA_m \backslash (NotExAnteAbs_m \cup D)$ then denotes the set of AnteAbs nodes satisfying the necessary execution conditions without nodes that have already been discarded and nodes connected to AnteOcc nodes in $ExAnteOcc_m$.

Then,

- $executeAnteAbsAE_R : M_R^* \times E^{Ex} \rightarrow M_R^*$
  is a function assigning a child ExMark m' to a given ExMark m and an EX event e with:

  $executeAnteAbsAE_R(m, e) := markAnteAbsAE(m, Q, e)$.

Marking Rule MR12 (Marking of AnteAbs nodes over EX events):
Let $R = (A, C)$ be a CRG and $m = ((ns_A, nl_A), (ns_C, nl_C)) \in M_R^*$ be an ExMark of CRG R. Then,

$markAnteAbsAE_R : M_R^* \times \mathcal{P}(N_R) \times E^{Ex} \rightarrow M_R^*$

is a function assigning an ExMark m' of R to an original ExMark m of R and a set of AnteAbs nodes to be executed Q, with

$markAnteAbsAE_R(m, Q, e) := m' = ((ns'_A, nl'_A), (ns'_C, nl'_C))$ with:

- $\forall n \in N_R$ with $nt_R(n) = \text{AnteAbs}$:

$$(ns'_A(n), nl'_A(n)) := \begin{cases} (\textsc{Completed}, \emptyset) \ if \ n \in Q \\ (ns_A(n), nl_A(n)), \ otherwise. \end{cases}$$

- $\forall n \in N_R$ with $nt_R(n) = \textsc{AnteOcc}$:

  $(ns'_A(n), nl'_A(n)) := (ns_A(n), nl_A(n))$.
- $\forall n \in N_R$ with $nt_R(n) \in \{\textsc{ConsOcc}, \textsc{ConsAbs}\}$:

  $(ns'_C(n), nl'_C(n)) := (ns_C(n), nl_C(n))$.
- For m' we define attributes as follows:
  - $ExAnteOcc_{m'} := ExAnteOcc_m$,
  - $QCO_{m'} := QCO_m$,
  - $QCA_{m'} := QCA_m$.

A.2.2.3. Execution of ConsOcc nodes

Generally, ConsOcc nodes can be executed over EX events in the same manner as over START events. Nondeterministic execution of ConsOcc nodes by default, however, can cause unnecessary exploration of the search space. This is avoidable when applying a more greedy approach. The latter is based on the consideration that only ConsOcc nodes having direct ConsAbs successors are critical since we do not know whether an activity execution matching one of these ConsAbs successors will be executed in a future execution iteration. Thus, it is "safe" to deterministically execute ConsOcc nodes *without* direct ConsAbs successors. This avoids creating ExMarks that can be discarded anyway based on the considerations on domination among ConsExMarks as introduced in Section 7.5.1.2. Based on this consideration, execution rule ER11 deterministically executes ConsOcc nodes without direct ConsAbs successors (i.e., must fire if executable). Note that if a CRG does not contain any ConsAbs nodes and all its ConsOcc nodes are not connected through DIFF edges, all ConsOcc nodes will be executed deterministically avoiding unnecessary exploration of the search space.

Execution Rule ER11 (Execution of ConsOcc nodes over EX events):
Let $R = (A, C)$ be a CRG, $m = ((ns_A, nl_A), (ns_C, nl_C)) \in M_R^*$ be an ExMark of R, and $e \in E^{Ex}$ be an EX event. Further,

- $D := \{n \in QCO_m \mid \exists l \in ExAnteOcc_m : \ (n, l) \in DiffE_R \vee (l, n) \in DiffE_R\}$ denotes the subset of QCO_m containing nodes that are directly connected to AnteOcc nodes executed in this iteration through a DIFF edge.
- $NotExConsOcc_m := \{n \in QCO_m \mid ns_C(n) = \textsc{NotExecuted}\}$ denotes the subset of QCO_m containing those nodes already marked as NotExecuted.

- $N_m^{ex} := QCO_m \backslash (NotExConsOcc_m \cup D)$. Then, N_m^{ex} is partitioned into three disjunct subsets: $N_m^{ex} = N_m^{det} \cup N_m^{ndet} \cup N_m^{diff}$ with
  - $N_m^{det} := \{n \in N_m^{ex} \mid succConsAbs_R(n) = \emptyset \wedge (\nexists l \in N_m^{ex} : (n,l) \in DiffE_R \vee (l,n) \in DiffE_R)\}$ being the subset of N_m^{ex} containing only ConsOcc nodes that do not have direct ConsAbs successors and are not directly connected to other nodes in N_m^{ex} through a DIFF edge.
  - $N_m^{ndet} := \{n \in N_m^{ex} \mid succConsAbs_R(n) \neq \emptyset\}$ being the subset of N_m^{ex} containing ConsOcc nodes that have direct ConsAbs successors.
  - $N_m^{diff} := N_m^{ex} \backslash (N_m^{det} \cup N_m^{ndet})$ being the subset of N_m^{ex} containing ConsOcc nodes without direct ConsAbs successors but that are directly connected to other nodes in $N_m^{diff} \cup N_m^{ndet}$ through a DIFF edge.

Then, for ExMark m, the set of sets of ConsOcc nodes executable in an iteration is defined as follows:

- $Q^* := \{Q := (N_m^{det} \cup Q^{ndet} \cup Q^{diff}) \mid Q^{ndet} \subseteq N_m^{ndet}, Q^{diff} \subseteq N_m^{diff}$ and conditions (i) - (ii) are satisfied$\}$ where (i) - (ii) are defined as follows:
  (i) $\forall n_1 \forall n_2 \in Q^{diff} \cup Q^{ndet} : (n_1, n_2), (n_2, n_1) \notin DiffE_R$,
  (ii) $\forall n_1 \in N_m^{diff} : (n_1 \in Q^{diff}) \vee (\exists n_2 \in Q^{diff} \cup Q^{ndet} : (n_1, n_2) \in DiffE_R \vee (n_2, n_1) \in DiffE_R)$.

Then,

- $executeConsOccAE_R : M_R^* \times E^{Ex} \rightarrow 2^{M_R^*}$
  is a function assigning a child ExMark m' to an ExMark m and EX event e with:
  $executeConsOccAE_R(m, e) := \bigcup_{Q \in Q^*} markConsOccAE_R(m, Q, e)$.

Marking Rule MR13 (Marking of ConsOcc nodes over EX events):
Let $R = (A, C)$ be a CRG. Then,

$markConsOccAE_R : M_R^* \times \mathcal{P}(N_R) \times E^{Ex} \rightarrow M_R^*$

is a function assigning an ExMark m' of R to an original ExMark $m = ((ns_A, nl_A), (ns_C, nl_C))$ of R and a set of ConsOcc nodes Q to be executed, with

$markConsOccAE_R(m, Q, e) := m' = ((ns_A, nl_A), (ns'_C, nl'_C))$ with:

- $\forall n \in N_R$ with $nt_R(n) = $ ConsOcc:

$$(ns'_C(n), nl'_C(n)) := \begin{cases} (\text{Completed}, \emptyset) \; if \; n \in Q \\ (ns_C(n), nl_C(n)), \; otherwise. \end{cases}$$

- $\forall n \in N_R$ with $nt_R(n) = \texttt{ConsAbs}$:

$$(ns'_C(n), nl'_C(n)) := \begin{cases} (\texttt{NotExecuted}, \emptyset) \; if \; n \in deadConsAbs_R(ns_C, Q) \\ (ns_C(n), nl_C(n)), \; otherwise. \end{cases}$$

- For m' we define attributes as follows:
  - $ExAnteOcc_{m'} := ExAnteOcc_m$,
  - $ExConsOcc_{m'} := Q$,
  - $QCA_{m'} := QCA_m$.

A.2.2.4. Execution of ConsAbs nodes

All ConsAbs nodes that are executable in an iteration (and that are not yet discarded and not connected to AnteOcc or ConsOcc nodes executed in the very execution iteration) will become Completed. Containing a ConsAbs node marked as Completed, the corresponding ConsExMark becomes violated and, thus, can be discarded based on the considerations introduced in Section 7.5.1.

Execution Rule ER12 (Execution of ConsAbs nodes over EX events):
Let $R = (A, C)$ be a CRG and $m = ((ns_A, nl_A), (ns_C, nl_C)) \in M^*_R$ be an ExMark of R. Further,

- $D := \{n \in QCA_m \mid \exists l \in (ExAnteOcc_m \cup ExConsOcc_m : (n, l) \in Diff_R \vee (l, n) \in Diff_R\}$ denotes the set of ConsAbs nodes in QCA_m that are directly connected through a DIFF edge to AnteOcc and ConsOcc nodes that have been executed in this iteration.
- $NotExConsAbs_m := \{n \in QCA_m \mid ns_C(n) = \texttt{NotExecuted}\}$ denotes the subset QCA_m containing those nodes already marked as NotExecuted.
- $Q := QCA_m \backslash (NotExConsAbs_m \cup D)$ denotes the set of executable ConsAbs nodes of m. It is ensured that Q is free of nodes that are directly connected through a DIFF edge to AnteOcc nodes or ConsOcc nodes executed in same iteration.

Then,

- $executeConsAbsAE_R : M^*_R \times E^{Ex} \rightarrow M^*_R$
  is a function assigning a child ExMark m' to a given ExMark m and an EX event e with:

  $executeConsAbsAE_R(m, e) := markConsAbsAE(m, Q, e)$.

Marking Rule MR14 (Marking of ConsAbs nodes over EX events):
Let $R = (A, C)$ be a CRG. Then,

$markConsAbsAE_R : M_R^* \times \mathcal{P}(N_R) \times E^{Ex} \rightarrow M_R^*$ is a function assigning an $\texttt{ExMark}$ m' of R to an original $\texttt{ExMark}$ $m = ((ns_A, nl_A), (ns_C, nl_C))$ of R and a set Q of $\texttt{ConsAbs}$ nodes to be executed, with

$markConsAbsAE_R(m, Q, e) := m' = ((ns_A, nl_A), (ns'_C, nl'_C)) \in M_R^*$ with:

- $\forall n \in N_R$ with $nt_R(n) = \texttt{ConsAbs}$:

$$(ns'_C(n), nl'_C(n)) := \begin{cases} (\texttt{Completed}, \emptyset) \ if \ n \in Q \\ (ns_C(n), nl_C(n)), \ otherwise. \end{cases}$$

- $\forall n \in N_R$ with $nt_R(n) = \texttt{ConsOcc}$:

  $(ns'_C(n), nl'_C(n)) := (ns_C(n), nl_C(n))$.

A.3. Propagation of compliance states

Algorithm 6 Marking an acyclic process event graph: $propagate(s_{start}, MS_{s_{start}})$

```
1: $X = (S, s_0, S_E, T, el)$ is a PEG
2: $R$ is a CRG
   {INITIALIZATION}
3: $\forall s \in S: \; pr(s) = \emptyset$;
4: $\forall e \in T: \; pr(e) = \emptyset$;
5: $pr(s_0) = \{\{init(R)\}\}$;
6: $Q = \{s_0\}$;
   {ITERATION}
7: while $Q \neq \emptyset$ do
8:   $s = Q[1]$; {Process and remove the head element of $Q$}
9:   $Q = Q \backslash Q[1]$;
     {Propagate MarkStructures of node $s$ to all its outgoing edges}
10:  for all $e = (s, s') \in T$ do
11:    $pr(e) = pr(s)$;
12:  end for
     {For each direct successor of $s$ whose incoming edges have all been signaled, compute the
     reachable compliance states and add the node to $Q$}
13:  for all $s' \in S$ with $(s, s') \in T$ do
14:    if $\forall e = (t, s') \in T: \; pr(e) \neq \emptyset$ then
15:      $In = \bigcup_{e=(t,s') \in T} pr(e)$;
16:      $NewCSs = \emptyset$;
17:      for all $CS \in In$ do
18:        $CS' = \bigcup_{ms \in CS} execute_R(ms, el(s'))$;
19:        $NewCSs = NewCSs \cup \{CS'\}$;
20:      end for
21:      $pr(s') = NewCSs$;
22:      $Q = Q \cup \{s'\}$;
23:    end if
24:  end for
25: end while
26: return $pr$;
```

Algorithm 7 Marking a (cyclic) process event graph: *propagate*

```
 1: $G = (S, s_0, s_E, T, el)$ is a PEG
 2: $R$ is a CRG
    {INITIALIZATION}
 3: $\forall s \in S: \; pr(s) = \emptyset$;
 4: $\forall e \in T: \; pr(e) = \emptyset$;
 5: $\forall s \in S: \; In_s = \emptyset$;
 6: $\forall e \in T: \; sig(e) = \emptyset$;
    {MARKING OF PROPAGATION START NODE AND ITS OUTGOING EDGES}
 7: $pr(s_0) = \{\{init(R)\}\}$;
 8: $Q = \{(s, s') \in T \mid s = s_0\}$;
 9: $\forall e \in Q: \; sig(e) = pr(s_0)$;
10: $\forall e \in Q: \; pr(e) = pr(s_0)$;
    {ITERATION}
11: while $Q \neq \emptyset$ do
12:   $e = (s, s') = Q[1]$; {Process and remove the head element of $Q$}
13:   $Q = Q \backslash Q[1]$;
      {Identify MarkStructures associated with $e$ that have not yet been signaled to $s'$ by its
      other incoming edges}
14:   $newIn = sig(e) \backslash In_{s'}$;
      {If $e$ is associated with new MarkStructures ...}
15:   if $newIn \neq \emptyset$ then
16:     $In_{s'} = In_{s'} \cup newIn$;
17:     $ResCSs = \emptyset$;
18:     for all $CS \in newIn$ do
19:       $CS' = \bigcup_{ms \in CS} execute_R(ms, el(s'))$; {Apply operational semantics for new incoming
          compliance states}
20:       $ResCSs = ResCSs \cup \{CS'\}$;
21:     end for
22:     $newCS = ResCSs \backslash pr(s')$; {... and identify new compliance states}
        {If new compliance states result from the application of CRG operational semantics ...}
23:     if $newCS \neq \emptyset$ then
24:       $pr(s') = pr(s') \cup newCS$; {... enrich annotation of $s'$ with these}
25:       for all $e' = (s', s'') \in T$ do
26:         $sig(e') = sig(e') \cup newCS$;
27:         $pr(e') = pr(e') \cup newCS$;
28:         $Q = Q \cup e'$; {... and propagate new compliance states to all outgoing edges of $s'$}
29:       end for
30:     end if
31:   end if{Reset $sig$ after processing a PEG edge}
32:   $sig(e) = \emptyset$;
33: end while
34: return $pr$;
```

A.4. Rule enforcement levels

Table A.1 lists the levels of enforcement of business rules proposed by the OMG in the specification of *semantics for business vocabularies and rules* (SBVR) [OMG08]. As the compliance rules addressed in this thesis belong to the class of *behavioral rules* of SBVR, the adoption of these enforcement levels for compliance rules suggests itself.

Level of enforcement	Description	Implication for designing procedures
strictly enforced	If an actor violates the behavioral rule, the actor cannot escape sanction(s).	When a violation is detected, the event producing the violation is automatically prevented, if possible, and a designated violation response, if any, is invoked automatically.
deferred enforcement	The behavioral rule is strictly enforced, but such enforcement may be delayed – e.g., until another actor with required skills and proper authorization can become involved.	When a violation is detected, the event producing the violation is allowed, and the relevant work is handed off to another worker (possibly by insertion into a work queue). Additional business rules giving timing criteria may be desirable to ensure that action is taken within an appropriate time frame.
override by pre-authorized actor	The behavioral rule is enforced, but an actor with proper before-the-fact authorization may override it.	When a violation is detected, if the actor involved is pre-authorized, that actor is given an opportunity to override the rule. Overrides by actor and business rule should be tracked for subsequent review.
post-justified override	The behavioral rule may be overridden by an actor who is not explicitly authorized; however, if the override is subsequently deemed inappropriate, the actor may be subject to sanction(s).	When an override of a violation occurs, a review item (with all relevant details) should be inserted into the work queue of an appropriate actor for review and possible action.
override with explanation	The behavioral rule may be overridden simply by providing an explanation.	When a violation is detected, the actor involved is given an opportunity to override the business rule by providing a mandatory explanation. Overrides should be tracked by actor and business rule for subsequent review.
guideline	Suggested, but not enforced.	When a violation is detected, the actor involved (if authorized) is simply informed/reminded of the behavioral rule.

Table A.1.: Levels of enforcement for business rules and their implication for designing procedures taken from [Ros11]