Visually Specifying Compliance Rules and Explaining their Violations for Business Processes

Ahmed Awad, Matthias Weidlich, and Mathias Weske
{ahmed.awad, matthias.weidlich,mathias.weske}@hpi.uni-potsdam.de

Hasso-Plattner-Institute
University of Potsdam, Germany

Abstract

A business process is a set of steps designed to be executed in a certain order to achieve a business value. Such processes are often driven by and documented using process models. Nowadays, process models are also applied to drive process execution. Thus, correctness of business process models is a must. Much of the work has been devoted to check general, domain-independent correctness criteria, such as soundness. However, business processes must also adhere to and show compliance with various regulations and constraints, the so called compliance requirements. These are domain-dependent requirements.

In many situations, verifying compliance on a model level is of great value, since violations can be resolved in an early stage prior to execution. However, this calls for using formal verification techniques, e.g., model checking, that are too complex for business experts to apply. In this paper, we utilize a visual language, BPMN-Q, to express compliance requirements visually in a way similar to that used by business experts to build process models. Still, using a pattern based approach, each BPMN-Q graph has a formal temporal logic expression in computational tree logic (CTL). Moreover, the user is able to express constraints, i.e., compliance rules, regarding control flow and data flow aspects. In order to provide valuable feedback to a user in case of violations, we depend on temporal logic querying approaches as well as BPMN-Q to visually highlight paths in a process model whose execution causes violations.

Keywords: Business Process Modeling, Compliance Checking, Visual Modeling, Anti-Patterns

1. Introduction

Business processes are central to the operation of both, public and private organizations. A business process is a set of coordinated activities to achieve a specific business objective. Business process models, in turn, emerged as a means to document business processes. Also, they are used to estimate and align IT requirements with business requirements. Moreover, business process models are the blueprints for automating the execution of business activities.

Business processes have to comply to certain regulations that might be due to legal requirements or business best practices. That is, the behavior of business processes has to meet compliance requirements that originate from different sources. For instance, the Sarbanes and Oxley act [1] enforces regulations in order to control financial transactions in the public sector. Another example are the best practices and recommendations specified by BASEL [2] for risk management in the banking sector. A specific example is the variety of checks that is required to take place before opening a new bank account.

As business process models capture the way business is conducted, they are the best place to check for and enforce compliance to organization policies and external regulations. Note that we focus on business processes that are driven by process models, such that compliance of the business process can be deduced from compliance of the respective process model. However, in scenarios that do not show a tight coupling between process models and process execution, our approach might still be applied to process models derived via process mining [3], i.e., process models synthesized from process execution logs. Unlike classical correctness (soundness) criteria [4, 5, 6], which aim at domain-independent *verification* of business processes, compliance requirements are domain-dependent *validation*

criteria. Compliance requirements vary between different business domains and might even deviate for different enterprises working in the same domain.

Checking a process model for compliance with respect to some compliance rule is a non-trivial task and, therefore, should be supported by appropriate tools and concepts. In order to provide automated support for compliance checking, compliance rules have to be expressed in a formal language. That, in turn, allows for leveraging mature model checking techniques [7]. However, these techniques cannot be employed directly. They require the rules, to be checked, to be specified in a formalism, i.e., temporal logic, which is inappropriate for most process analysts that conduct process modeling. Thus, there is a notable gap between the formal techniques applied to validate compliance rules for business process models, and the level of process modeling that process analysts are familiar with. Note that this gap can be observed in two directions. On the one hand, process analysts should be able to specify compliance rules similar to the way process modeling is conducted, despite the application of automatic reasoning. On the other hand, feedback on violations of compliance rules has to be given on the process model level as well. Obviously, the pure result of a model checking procedure is of limited use for process analysts. Instead, this information needs to be *visualized* appropriately on the level of the process model.

In this paper, we address the need for means to bridge the gap between formal model checking techniques and the level of process modeling for control flow and data flow aspects of process models. We focus on compliance rules that require activities to exist in a process model or define constraints regarding the order of activity execution. In some cases, such constraints are required to hold solely under certain data conditions. Consequently, the control flow, the data flow, and their dependencies need to be expressed explicitly in compliance rules. To this end, our solution is based on a visual language to express compliance rules, BPMN-Q [8]. BPMN-Q is a language to express queries on the structure of process models in a way very similar to that of expressing process models themselves. In addition, it provides means for abstraction of process model details, a feature that is inevitable for expressing compliance rules. Moreover, some elements of BPMN-Q serve as wrappers for complex temporal logic formulas. Thus, we go beyond structural querying by taking the process behavior into account.

We employ a pattern based approach [9, 10]. Based on a study of common patterns for specifying constraints on behavioral models [10], we have developed a set of patterns in BPMN-Q to express these constraints for process models. Each pattern is visually represented as a BPMN-Q query, has a defined mapping to an equivalent Computational Tree Logic (CTL) formula, and has a set of associated anti-pattern queries. In case of violation, the latter are required to query the part of the process model causing the violation.

This paper is based on a series of papers. BPMN-Q has been introduced as a structural query language in [8], while it has been extended with wrappers for elementary past linear temporal logic (PLTL) formula in [11]. Wrappers for data-aware PLTL formula have been added to BPMN-Q in [12], which also sketched the derivation of anti-patterns to provide on violations. These anti-patterns have been discussed further in [13], without any formalization.

This paper integrates and significantly extends these results in order to provide a holistic framework for the specification of compliance rules that is still grounded on temporal logic to enable automated reasoning. This paper describes an end-to-end compliance checking approach including visualization of compliance violations that, in this entirety, has not been proposed before. In detail, the contributions of this paper are the following.

- An extension of the available set of atomic compliance rules. Based on the classification in [10], we introduce new rules that go beyond the rules defined in our previous work [11, 12]. The set of new rules comprises compliance patterns on precedence to conditional activity execution (Section 4.3.2), conditional before-scope absence (Section 4.3.3), and conditional between-scope absence (Section 4.3.5).
- A formalization of both, compliance rules (Section 4) and anti-patterns (Sections 5.2 to 5.4), in CTL. Compliance rules have been formalized in PLTL in our previous work, while there has been no formalization of anti-patterns. The motivation for the usage of CTL is the availability of several CTL solvers, more efficient reasoning compared to PLTL, and the possibility to define compliance rules and their anti-patterns in the same formalism (anti-patterns cannot be formalized in PLTL).
- An approach to solve temporal logic queries (Section 5.6). It reduces temporal logic queries to a series of model checking problems by leveraging knowledge on the domain of process modeling.
- A description of our implementation of the approach.

We illustrate the applicability of our approach by means of an example scenario. It uses a process model derived from real-world guidelines for financial processing and compliance rules that stem from laws on anti-money laundering [14]. We show how compliance rules are expressed using BPMN-Q, how they are mapped to temporal logic, and

what kind of feedback is given on compliance violations.

Against this background, the remainder of this paper is structured as follows. Section 2 gives preliminaries for our work in terms of background information on process modeling, the usage of model checking techniques for compliance checking, and the query language BPMN-Q. Section 3 introduces our compliance checking approach. Then, we elaborate on its details in the subsequent sections. Section 4 shows how compliance rules are specified visually. Visualization of potential compliance violations is detailed in Section 5. Section 6 presents an example scenario that illustrates the application of our approach. Section 7 discusses the implementation of our approach. Finally, we discuss our contribution in the light of related work in Section 8 and conclude in Section 9.

2. Preliminaries

This section gives preliminaries for our work. First, Section 2.1 discusses common process modeling languages and their relation to formal models. Second, Section 2.2 gives details on Computational Tree Logic (CTL) as the formalism used throughout this paper. Finally, BPMN-Q is summarized in Section 2.3.

2.1. Business Process Modeling

There is a number of graph-based business process modeling languages, e.g., BPMN [15], EPCs [16], and UML activity diagrams (AD) [17]. Despite their variance in expressiveness and the used notation, the aforementioned languages share a common set of concepts. In this paper, we focus on these *core concepts* as a target for compliance checking. For illustration purposes, we use the terminology and notation defined by BPMN. Nevertheless, our findings can easily be transfered to other process modeling languages.

In general, a process model is a graph consisting of nodes and edges. Nodes can be divided into *flow objects*, i.e., nodes that correspond to active elements in a process, and *data objects*, passive data elements that are read or updated during the execution of a process. Flow objects, in turn, can be further classified into *activities*, *events*, and *gateways*. An activity represents the concept of a process step, task, or a function that is atomic and executable. In this paper, we assume these activities to have unique labels. Moreover, events are applied to model the occurrence of real-world events. Also, special types of events are the start and end events which indicate the creation and termination of a process instance, respectively. Finally, gateways are used to implement an execution logic that goes beyond simple sequences of activities or events. Most prominently, there are XOR and AND gateways. Moreover, edges are used to connect the nodes of a process model. We distinguish two types of edges, sequence flow and data association edges. Sequence flow edges connect flow objects to model the control flow order between them. Data association edges, in turn, connect data objects and flow objects. Depending on the direction of the edge, this is interpreted either as a reading access or a writing access of the data object, respectively.

Execution semantics of process models are often defined following on the token flow paradigm known from Petri nets [18]. In particular, gateways activate either one (XOR) or all (AND) of the outgoing sequence flow edges when applied as a split, or wait for the activation of one (XOR) or all (AND) incoming sequence flow edges when applied as a join. For more details on mappings of BPMN to Petri nets, we refer interested readers to [19, 20, 21]. Note that we assume a process model to have a dedicated start event and a dedicated end event. In addition, we assume process models to be well-formed in the sense that they do not show any behavioral anomaly, e.g., a deadlock or a livelock. That is, a process model is sound [4].

As mentioned above, we consider compliance rules that take data aspects into account. To this end, we depend on the notion of a data object to encapsulate data elements accessed by different activities of a process. There is a single copy of each data object that is created upon process instantiation and then handled within the process. Therefore, multiple data object shapes with the same label are considered to refer to the same data object of the process model.

Each data object is assumed to be in a certain state at any time during the execution of the process. This state changes through the execution of activities. An activity can specify which state a data object must be in before an activity is allowed to be executed (precondition) and which state a data object will be in after having executed an activity (effect). Often it is required that a data object is in exactly one state before an activity is allowed to be executed. However, it might also be the case that the data object is allowed to be in one of a set of states. Such a dependency is visualized by multiple data association edges targeting an activity, such that each originates at a data object shape referring to the same data object but in different states. Similarly, alternative effects are denoted by

(a) Anonymous Activity (b) Path Edge (c) Path Edge with Exclude (d) Implicit Data Input Edge

Figure 1: Representation of abstraction concepts introduced by BPMN-Q

multiple data association edges originating at a single activity and targeting multiple data object shapes referring to the same data object in different states. Note that states of data objects are also referenced at XOR gateways that are used to split the control flow. Again, data access semantics can be formally defined through a mapping of data objects and their states, as well as data association edges to Petri nets. Please refer to [22] for further details on this mapping.

2.2. Computational Tree Logic

Computational Tree Logic (CTL) [23] is a language to express properties for model checking. In addition to logical operators $(\neg, \wedge, \vee, \rightarrow)$ and atomic propositions, CTL provides temporal operators to express properties that must hold through a number of states that are temporally related. Moreover, CTL is designed to handle nondeterminism in the behavior of a system. Thus, in addition to introducing temporal operators; it introduces for-all and existential quantifiers. For this paper, we need the following operators for propositions θ and ω.

- $\mathbf{AG}(\theta)$: θ has to hold in *every* future state on *every* execution path.
- $\mathbf{EG}(\theta)$: θ has to hold in *every* future state on *some* execution path.
- $\mathbf{AF}(\theta)$: θ has to hold in *some* future state in *every* execution path.
- $\mathbf{EF}(\theta)$: θ has to hold in *some* future state in *some* execution path.
- $\mathbf{A}[\omega\ \mathbf{U}\ \theta]$: ω has to be true *until* θ is reached in *every* execution path.
- $\mathbf{E}[\omega\ \mathbf{U}\ \theta]$: ω has to keep true *until* θ is reached in *some* execution path.

2.3. Querying Process Models: BPMN-Q

The Business Process Model Notation Query (BPMN-Q) language was designed to help process analysts to access repositories of business process models. While we focus on querying of process models defined in BPMN, BPMN-Q is based on the core concepts of process modeling languages as discussed in Section 2.1. Therefore, it can easily be adapted for other modeling languages such as EPCs or UML ADs by providing an appropriate concrete syntax (visual notation) for the following *abstraction* concepts introduced by BPMN-Q.

- **Path edge.** A path edge connecting two flow objects in a query represents an abstraction over any chain of sequence flow edges and flow objects between both flow objects in a process model. Moreover, path edges may have an *exclude* property. When the *exclude* property is set to some flow object, the path edge represents solely those chains of sequence flow edges and flow objects that do not contain the flow object referenced in the exclude property. In general, path edges consider only the structure of a process. However, execution semantics can be taken into account by typing a path edge between two flow objects as being either a `precedes` or a `leads to` path [11, 12]. That, in turn, allows to reflect precedence and response ordering relations [10].
- **Anonymous activity.** In order to query for the existence of a certain access of data objects in a process model, the concrete activity realizing the data access might be negligible. In this case, a query can contain an anonymous activity (labeled with the '@' symbol) that is matched by any activity in the process model.
- **Implicit data input edge.** This type of edge is introduced to reason about data conditions at the time an activity is about to execute. It is used to state an implicit data dependency for an activity in a compliance query, i.e., the dependency is not required to be stated explicitly in terms of a data association edge in the process model [12].

Figure 1 summarizes the symbols used to represent the new concepts in the BPMN-Q. We illustrate the difference between structural and behavioral queries of BPMN-Q by Figure 2. Figure 2 a) shows an example process model. Figure 2 b) is a structural query that matches a path between activity "A" and "D". The nodes and edges that reside on that path are highlighted in gray in Figure 2 a). However, finding such structural match is not sufficient to conclude that each time "A" executes, "D" will eventually execute thereafter. This requirement would be formalized as "A" `leads to` "D", while Figure 2 c) shows its visualization. Structural matching is also not sufficient to decide whether each

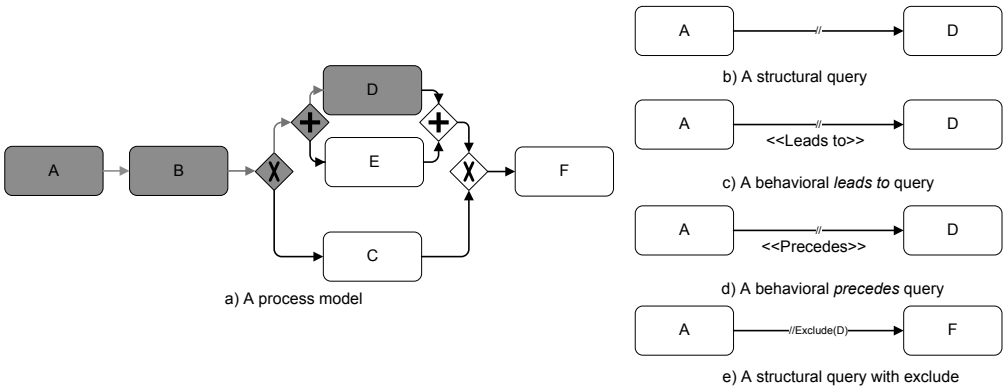


Figure 2: Difference between structural and behavioral BPMN-Q queries

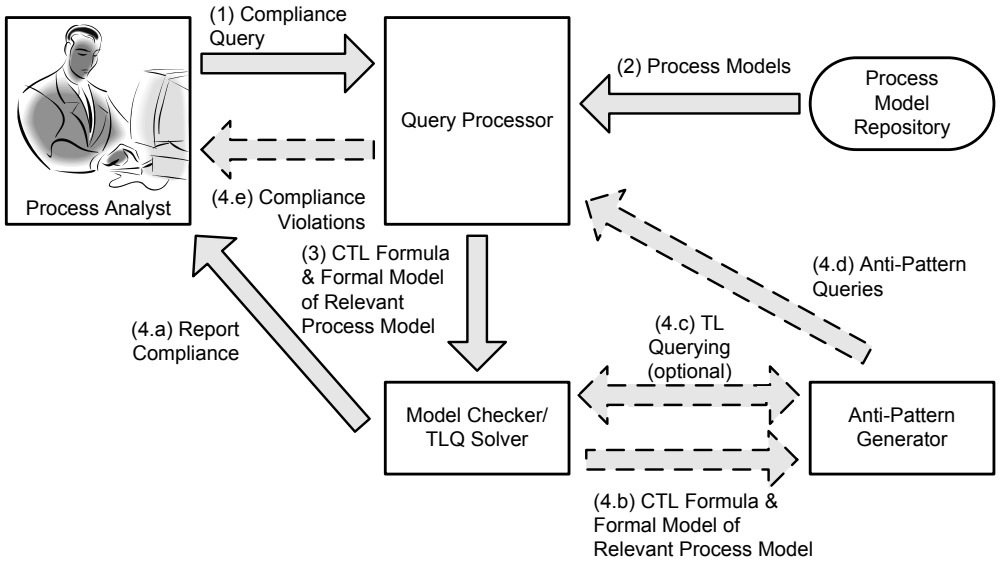


Figure 3: Overview of our compliance checking approach

time "D" executes, "A" must have had executed before. This corresponds to the requirement "A" `precedes` "D", which is visualized in Figure 2 d). Since these two behavioral relationships are different from each other and cannot be decided solely by a structural match, two stereotypes <<Leads to >>and <<Precedes >>are used to distinguish these concepts.

Finally, Figure 2 e) shows another structural query that looks for a path from "A" to "F" without visiting "D". A match to this query could be the whole process model except the edge from the AND-split to "D", the activity "D", and the edge from "D" to the AND-join. However, for the specific case of anti-patterns, if we look for an execution path from"A" to "F" without executing "D", then it is logical not include "E" either, as both "E" and "D" belong to a parallel block. For this reason, when we exclude a node, we exclude all its directly enclosing AND-splits. Therefore, we exclude all of its parallel branches.

3. The Compliance Checking Approach

Our approach of applying model checking techniques for automated compliance checking is summarized in Figure 3. First and foremost, a process analyst specifies compliance requirements as behavioral queries. As mentioned above, it is crucial for the applicability of the approach that process analysts can formulate these queries in a language

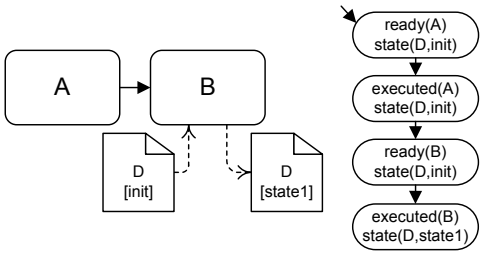


Figure 4: Process model and corresponding state transition system

they are familiar with. That is, the language to specify compliance queries should be based on the core concepts of process modeling languages discussed in Section 2.1, while a similar notation is used for visualization. Subsequently, the query is utilized in a *query processor* to select all process models that are relevant regarding the compliance query out of the set of process models stored in a repository. This selection is guided by the activities mentioned in the compliance query.

Each of the selected process models, as well as the compliance query are then forwarded to a *model checker*. A model checker allows for the verification of a temporal logic formula against a formal behavioral model [7]. Therefore, any selected process model has to be translated into a formal behavioral model. By leveraging the Petri net mapping referenced in Section 2.1, a state transition system (STS) is derived for a process model. States of this STS are build using the following elementary predicates, in order to allow for reasoning about the execution of activities and changes of data object states.

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

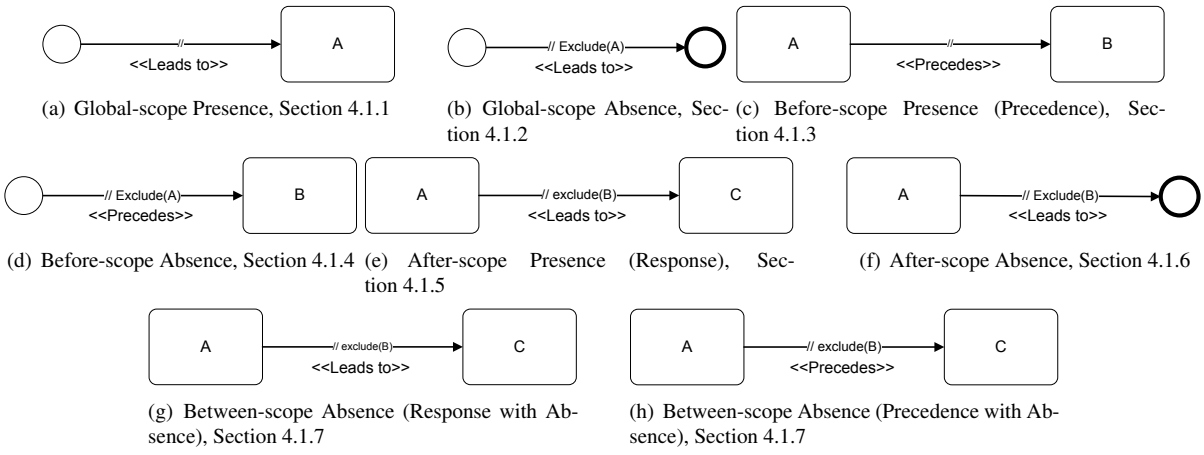
Figure 4 exemplifies the derivation of the STS for small process excerpt. The predicates used in the STS, are also applied in order to translate compliance queries into CTL formulas. Given the STS for a selected process model, the model checker decides whether the CTL formula representing the compliance rule is satisfied. This result is communicated to the process analysts. In case of a negative result, the source of violation is detected in a multi-step approach. First, the STS and the CTL formula are forwarded to the *anti-pattern generator*, which creates a set of a anti-pattern queries in order to localize the violation. The creation of these anti-patterns might involve temporal logic querying, i.e., extracting state predicates of a behavioral model under which a certain temporal logic formula is satisfied. Second, the anti-pattern query is forwarded to the query processor. As such an anti-pattern query is a structural query, it can directly be applied to the respective process model. Finally, the part of the process model matching this structural query is presented to the process analyst as the source of violation of the compliance query.

In this paper, we focus on two aspects of this approach. On the one hand, the gap between the visual specification of a compliance rule using concepts known from process modeling and the CTL formula is addressed in Section 4. On the other hand, the derivation of anti-pattern queries from the CTL formula representing the compliance rule in order to provide feedback on any violation is introduced in Section 5. This section also gives details on temporal logic querying.

4. Visual Specification of Compliance Requirements

This section introduces our approach of specifying compliance requirements visually using BPMN-Q. First, we discuss compliance rules that relate solely to constraints on the execution order of activities in Section 4.1. Second, Section 4.2 introduces the specification of compliance rules that formulate requirements in terms of states of data objects, if an activity is about to be executed. Third, Section 4.3 addresses the combination of both kinds of compliance rules, i.e., execution order constraints have to hold only if certain data requirements are met.

It is worth to mention that we introduce our approach with a focus on compliance rules that consider activities instead of events of a business process. However, all rules introduced for activities can also be applied for events.



(a) Global-scope Presence, Section 4.1.1

(b) Global-scope Absence, Section 4.1.2

(c) Before-scope Presence (Precedence), Section 4.1.3

(d) Before-scope Absence, Section 4.1.4

(e) After-scope Presence (Response), Section 4.1.5

(f) After-scope Absence, Section 4.1.6

(g) Between-scope Absence (Response with Absence), Section 4.1.7

(h) Between-scope Absence (Precedence with Absence), Section 4.1.7

Figure 5: Control Flow Compliance Patterns Expressed in BPMN-Q

4.1. Control Flow Patterns

Control flow patterns focus on activities and their order of potential execution. For instance, certain activities must always be executed or should never occur after the execution of another activity. The compliance requirement that *complaints must always be archived* would be an example for the former. Further on, two activities might also be required to be exclusive to each other, i.e., they are never executed both in one process instance.

Based on [10], compliance rules are further classified according to the concepts of *scope* and *order of occurrence*. A scope is either *global*, i.e., the whole process model, *before* some activity, *after* some activity, or *between* two activities. On the other hand, an activity may be required to occur (be *present*) or not to occur (be *absent*) within a scope. As we will see this classification also relates to the well-known notions of *precedence* and *response* of activity executions.

In the remainder of this section, we introduce the visual notation in BPMN-Q along with the corresponding CTL formula for each of the categories of compliance rules.

4.1.1. Global-scope Presence

A single activity might be required to be executed in all process instances, e.g., in a shipment process the received packets must be inspected in every case. Thus, we call such pattern a global presence. It is depicted in Figure 5(a).

Formalization

The CTL formula for global-scope presence is

$$\mathbf{AG}(\mathbf{start} \rightarrow \mathbf{AF}(\mathbf{executed}(A))) \tag{1}$$

4.1.2. Global-scope Absence

It might be the case that a certain activity must not execute at all, i.e., the activity does not occur in any instance of the process model. This case is called global absence and depicted in Figure 5(b).

Formalization

The CTL formula for global-scope absence is

$$\mathbf{AG}(\mathbf{start} \rightarrow \mathbf{A}[\neg\mathbf{executed}(A)\ \mathbf{U}\ \mathbf{end}]) \tag{2}$$

4.1.3. Before-scope Presence (Precedence)

In general, before-scope presence rules require that the execution of an activity B is preceded by the execution of another activity, A. Still, this does not imply that B must be immediately preceded by A. Instead, other activities might be executed in between. However, each time B is executed, A must have been executed before. Note that before-scope presence rules are also known as *precedence* rules. Figure 5(c) shows the respective visual notation.

Formalization
The CTL formula for precedence is

$$\neg\mathbf{E}[\neg\mathbf{executed}(A)\ \mathbf{U}\ \mathbf{ready}(B)] \tag{3}$$

4.1.4. Before-scope Absence

An activity A might be required to be absent before the execution of another activity B. Thus, if B executes than A must have never been executed before, which is addressed by the BPMN-Q query in Figure 5(d) It states that there must not be an execution sequence from the start of the process to activity A that eventually leads to the execution of activity B.

Formalization
The CTL formula for before-scope absence is

$$\neg\mathbf{EF}(\mathbf{start} \wedge \mathbf{EF}(\mathbf{executed}(A) \wedge \mathbf{EF}(\mathbf{ready}(B)))) \tag{4}$$

4.1.5. After-scope Presence (Response)

In general, the after-scope presence pattern states that after execution of an activity A another activity B has to be executed eventually. This pattern is also known as the *response* pattern. The representation of such a rule in BPMN-Q is shown in Figure 5(e).

Formalization
The CTL formula for response is

$$\mathbf{AG}(\mathbf{executed}(A) \rightarrow \mathbf{AF}(\mathbf{executed}(B))) \tag{5}$$

4.1.6. After-scope Absence

Similar to the before-scope absence pattern, it might be required to express that certain activities are forbidden to be executed after execution of another activity. Figure 5(f) depicts the BPMN-Q query that corresponds to this kind of compliance rule.

Formalization
The CTL formula for after-scope absence is

$$\mathbf{AG}(\mathbf{executed}(A) \rightarrow \mathbf{A}[\neg\mathbf{executed}(B)\ \mathbf{U}\ \mathbf{end}]) \tag{6}$$

4.1.7. Between-scope Absence

This kind of rule forbids the execution of a certain activity between the execution of two other activities. An example for this kind of rule would be the following requirement: 'When a new order is received; it is not allowed to forward it to finance department until its bill of material is calculated'.

Still, there are two variations of this kind of rule. In Figure 5(g), activity C is required to be executed after activity A, while activity B must not be executed in between. This could be also considered as a *response with absence* pattern. Similarly, Figure 5(h) illustrates that activity B might be forbidden to be executed, when an execution of activity C is requested to be preceded by an execution of activity A. This pattern, in turn, could be considered as a *precedence with absence* pattern.

Formalization
The CTL formula for response with absence is

$$\mathbf{AG}(\mathbf{executed}(A) \rightarrow \mathbf{A}[\neg\mathbf{executed}(B)\ \mathbf{U}\ \mathbf{executed}(C)]) \tag{7}$$

The CTL formula for precedence with absence is

$$\begin{aligned}&\neg\mathbf{E}[\neg\mathbf{executed}(A)\ \mathbf{U}\ \mathbf{ready}(C)]\wedge\\&\neg\mathbf{EF}(\mathbf{executed}(A) \wedge \mathbf{EF}(\mathbf{executed}(B) \wedge \mathbf{EF}(\mathbf{ready}(C))))\end{aligned} \tag{8}$$

4.2. Data Flow Patterns

In order to ensure compliance, it might be needed to ensure that data elements of a business process assume a certain state once a dedicated activity is about to be executed. As discussed in Section 2.1, explicit data association edges between data elements and activities might be applied to express preconditions and effects, respectively, in a process model. However, a compliance rule might refer to a dependency that is not explicitly mentioned in the model. To this end, BPMN-Q introduces the concept of implicit data input edges, cf., Section 2.3. Figure 6 shows how this kind of compliance rule can be expressed as a BPMN-Q query.

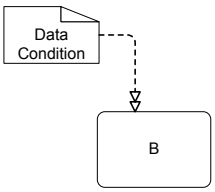


Figure 6: Data Flow Compliance Rule Expressed in BPMN-Q

Note that in case of a complex data dependency, multiple data objects along with implicit data input edges might be used to model the condition.

Formalization

Recalling the discussion of data objects (Section 2.1) and data access semantics (Section 3), a data condition on a single data element d and the set of data states (values) S can generally be expressed as

$$dataCondition = \bigvee_{s \in S} \mathbf{state}(d, s) \tag{9}$$

Consequently, the data flow compliance rule is formalized as

$$\mathbf{AG}(\mathbf{ready}(a) \rightarrow dataCondition) \tag{10}$$

4.3. Conditional Control Flow Patterns

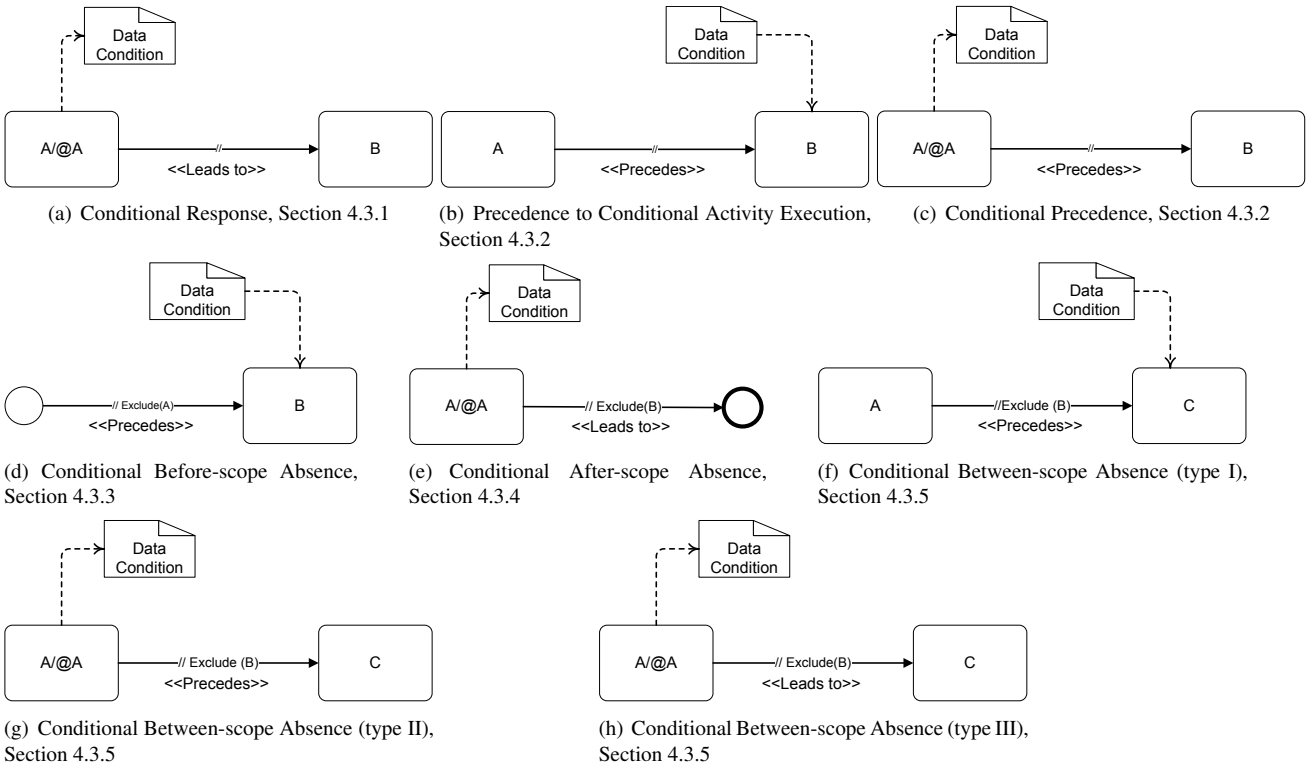
In the previous sections, we focused on control flow rules and data dependencies separately. This section shows how control flow rules can be refined, such that the order of activity occurrences is only required to hold under certain data conditions.

4.3.1. Conditional Response

First and foremost, it might be required to execute a certain activity once a data condition is met. For instance, consider a process in the banking sector. Here, it might be required that a respondent back is added to a black list once a due diligence evaluation fails. The visual representation of this kind of query is show in Figure 7(a). A data dependency is modeled by the respective data object associated to either a specific activity A or the anonymous activity $@A$. The use of the anonymous activity allows the modeler to abstract from any specific activity that actually sets the data value. Further on, activity B is required as a response of an execution of activity A or $@A$, respectively. Therefore, this pattern can be considered as a refinement for the response (alias global-scope presence) pattern discussed in Section 4.1.1.

Formalization

The formalization of this pattern is not straightforward as inappropriate modeling of the data dependency might lead to false alarms at model checking time. We illustrate this problem using the excerpt of a process depicted in Figure 8. Consider the following rule: Whenever activity A executes, such that data object d assumes state bad ($\mathbf{state}(d, bad)$ holds true), activity C must be executed thereafter. Formally, $\mathbf{AG}(\mathbf{executed}(A) \wedge \mathbf{state}(D, bad) \rightarrow \mathbf{AF}(\mathbf{executed}(C)))$.



(a) Conditional Response, Section 4.3.1

(b) Precedence to Conditional Activity Execution, Section 4.3.2

(c) Conditional Precedence, Section 4.3.2

(d) Conditional Before-scope Absence, Section 4.3.3

(e) Conditional After-scope Absence, Section 4.3.4

(f) Conditional Between-scope Absence (type I), Section 4.3.5

(g) Conditional Between-scope Absence (type II), Section 4.3.5

(h) Conditional Between-scope Absence (type III), Section 4.3.5

Figure 7: Conditional Control Flow Patterns Expressed as BPMN-Q Queries

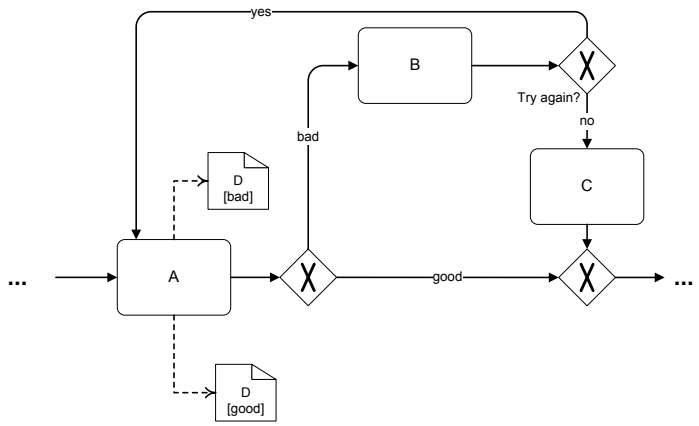


Figure 8: A process excerpt to illustrate a false alarm

Model checking this formula with the behavioral model of the process excerpt depicted in Figure 8 results in a false alarm. That is, the model checker decides that the formula is not satisfied. Consider the case where activity A is executed for the first time and assume that this results in data object d assuming state bad. At this execution state, the model checker records that the data dependency for the compliance rule is met. Afterwards, the process instance might continue with the execution of activity B and it may be decided to take the yes branch of the XOR decision (*Try again?*). Subsequently, activity A is executed again, which can result in data object d being set to state $good$. Subsequently, the process instance can continue execution. As activity C has not been executed, the model checker reports a violation of the compliance rule mentioned above.

We see that the original CTL formula is under-specified, the data dependency is not defined appropriately. Actually, we require that activity C is executed only when data object d assumes the state bad and never assumes the state $good$ afterwards. In order to cope with this requirement, we assume additional knowledge to be available. That is, states bad and $good$ are considered to be contradicting data states. Formally, for each data state s, there is a set of contradicting data states CON_s, which need to be taken into account explicitly in the formalization of the compliance rule.

$$contraDataCondition = \mathbf{AG}(\bigwedge_{s' \in CON_s} \neg \mathbf{state}(d, s')) \tag{11}$$

Based thereon, we can adapt the formalization of the data condition as follows

$$stableDataCondition = \bigvee_{s \in \mathcal{S}} (\mathbf{state}(d, s) \wedge \mathbf{AG}(\bigwedge_{s' \in CON_s} \neg \mathbf{state}(d, s'))) \tag{12}$$

The CTL formula for conditional response is

$$\mathbf{AG}((\mathbf{executed}(A) \wedge stableDataCondition) \rightarrow \mathbf{AF}(\mathbf{executed}(B))) \tag{13}$$

Here, $stableDataCondition$ refers to Formula 12.

Following on this argumentation, we can correct the formula for the example rule discussed above. This rules is captured as $\mathbf{AG}((\mathbf{executed}(A) \wedge \mathbf{state}(D, bad) \wedge \mathbf{AG}(\neg \mathbf{state}(D, good))) \rightarrow \mathbf{AF}(\mathbf{executed}(C)))$.

4.3.2. Conditional Precedence

The precedence pattern, cf., Section 4.1.3, can be refined with data dependencies in two different ways. A data condition can be attached to the source or the target activity. Thus, we distinguish two types of conditional precedence rules.

Precedence to conditional activity execution requires that an activity A has been executed before activity B executes under a certain data condition. For instance, before archiving a confirmed order, a copy of the order must have been sent to the marketing department. This kind of rule is captured in BPMN-Q as illustrated in Figure 7(b).

Conditional Precedence, in turn, requires that the execution of activity B must have been preceded by an execution of activity A, which resulted in a certain data condition. Again, we can abstract from the specific activity A using the anonymous activity, when the focus is on the pure data condition. This is captured by the BPMN-Q query shown in Figure 7(c).

Formalization

The CTL formula for precedence to conditional activity execution is

$$\neg \mathbf{E}[\neg \mathbf{executed}(A) \ \mathbf{U} \ \mathbf{ready}(B) \wedge dataCondition] \tag{14}$$

The CTL formula for conditional precedence is

$$\neg \mathbf{E}[\neg(\mathbf{executed}(A) \wedge \ stableDataCondition) \ \mathbf{U} \ \mathbf{ready}(B)] \tag{15}$$

In the above formulae, $dataCondition$ and $stableDataCondition$ refer to Formulae 9 and 12, respectively.

4.3.3. Conditional Before-scope Absence

Before-scope absence rules can be refined with data dependencies as follows. In case an activity is about to execute and a certain data condition holds, it is required that some other activity has not been executed before. For instance, in an order processing processes, it is required that at the time of archiving a canceled order, the activity for the shipment of the order has not been executed. Figure 7(d) shows the BPMN-Q query capturing this pattern.

Formalization

The CTL formula for conditional before-scope absence is

$$\neg \mathbf{EF}(\mathbf{start} \wedge \mathbf{EF}(\mathbf{executed}(A) \wedge \mathbf{EF}(\mathbf{ready}(B) \wedge dataCondition))) \tag{16}$$

4.3.4. Conditional After-scope Absence

Similar to the previous case, it might be forbidden to execute an activity once some data condition becomes true. The visual representation of this pattern is presented in Figure 7(e). While the modeling of the data condition equals the one introduced in Figure 7(a), the requirement of not executing a certain activity is represented by a *leads to* path to an end event with the *exclude* property set accordingly.

Note that this pattern can also be considered as being a refinement of the global-scope absence pattern, cf., Section 4.1.2, such that an activity must not be executed if a data condition is met. That is due to the potential usage of an anonymous activity.

Formalization

Following on the discussion on contradicting data states given above, the CTL formula for conditional after-scope absence is

$$\mathbf{AG}(\mathbf{executed}(A) \wedge stableDataCondition) \rightarrow \mathbf{A}[\neg\mathbf{executed}(B)\ \mathbf{U}\ \mathbf{end}]) \tag{17}$$

4.3.5. Conditional Between-scope Absence

Finally, the between-scope absence as discussed in Section 4.1.7 can also be refined with data conditions. Again, data conditions can be attached to the source or the target activity.

If the data condition is attached to the target activity as illustrated in Figure 7(f) (type I), a conditional execution of an activity C must be preceded by an execution of an activity A, while an activity B is never executed in between.

If the data condition is attached to the source activity, we distinguish two more cases. First, Figure 7(g) depicts the case (type II) that the execution of an activity C (not constrained by any data condition) must be preceded by the execution of an activity A or the anonymous activity, respectively, with a specific output data condition. In between, activity B is not allowed to be executed. Second, Figure 7(h) illustrates the case (type III) that after the execution of an activity A resulting in a specific data condition, activity C must be executed, such that activity B is not allowed to be executed in between.

Formalization

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

$$\begin{aligned}&\neg\mathbf{E}[\neg\mathbf{executed}(A)\mathbf{U}\ \mathbf{ready}(C) \wedge dataCondition]\wedge\\ &\neg\mathbf{EF}(\mathbf{executed}(A) \wedge \mathbf{EF}(\mathbf{executed}(B) \wedge \mathbf{EF}(\mathbf{ready}(C) \wedge dataCondition)))\end{aligned} \tag{18}$$

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

$$\begin{aligned}&\neg\mathbf{E}[\neg(\mathbf{executed}(A) \wedge stableDataCondition)\ \mathbf{U}\ \mathbf{ready}(C)]\wedge\\ &\neg\mathbf{EF}((\mathbf{executed}(A) \wedge stableDataCondition)\wedge\\ &\mathbf{EF}(\mathbf{executed}(B) \wedge \mathbf{EF}(\mathbf{ready}(C))))\end{aligned} \tag{19}$$

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

$$\begin{aligned}&\mathbf{AG}((\mathbf{executed}(A) \wedge stableDataCondition) \rightarrow\\ &\mathbf{A}[\neg\mathbf{executed}(B)\ \mathbf{U}\ \mathbf{executed}(C)])\end{aligned} \tag{20}$$

5. Visualization of Compliance Violations

The temporal logic formula that corresponds to a compliance rule can be verified against the behavioral model of a process model using model checking as introduced in Section 3. If the process model does not satisfy the compliance rule, we are interested in explaining the violation to the process analyst. As in case of modeling compliance rules, this feedback has to be given in a way that the process analyst is familiar with.

When a temporal logic formula is not satisfied, the model checker generates a counter-example, i.e., a sequence of states of the behavioral model that violates the rule. One might think of using this counter-example in order to explain compliance violations. However, this approach has several drawbacks. First, the counter-example is given

on the level of the behavioral model (i.e., the state space) of the process model. Therefore, such a sequence of states would have to be mapped back to the process model level in order to highlight the parts of the process model that caused the violation. This has to be seen as a serious pitfall as one state transition in the behavioral model can be mapped to many equivalent structural parts [24]. Second, although the identified counter-example is a dis-proof of the compliance rule, it is not complete. There might be other parts of the process model that cause violations as well. Therefore, we would have to fix the first found violation and then rerun the whole model checking approach. That, in turn, is ineffective. Instead, it would be more appropriate to identify all violations before any resolutions are applied to the process model. Finally, the identified counter-example depends on the behavioral model, i.e., the state transition system of the process model. In case the transition system is generated after reducing the original model for efficient reasoning (cf., [11, 25]), the counter-example cannot be used to draw conclusions on the original process model.

To avoid these shortcomings of using counter-examples, we decided to develop an alternative approach to visually explain violations to process analysts. For each compliance pattern introduced in Section 4, we analyze the process model under consideration and generate so-called anti-patterns. Such an anti-pattern is represented as a structural BPMN-Q query. Thus, for each of the compliance patterns, there is a set of anti-patterns that declaratively describe potential violations. The anti-pattern queries, in turn, can be matched to the original process model, such that the matching part represents the source of violation of the original compliance rule. While the anti-patterns can be created directly for some kinds of compliance patterns, certain compliance patterns require further investigation of the behavioral model of the process model in order to derive the anti-pattern queries. To this end, we rely on temporal logic querying [26].

In the remainder of this section, Section 5.1 first gives some details on temporal logic querying. Based thereon, we show how anti-patterns are derived for compliance patterns. Following on the classification of these patterns presented in Section 4, Section 5.2 introduces anti-patterns for control-flow patterns, while Section 5.3 focuses on patterns that specify data dependencies. Section 5.4 introduces the creation of anti-patterns for the conditional compliance patterns that combine control flow and data aspects. Section 5.5 elaborates on the relation between compliance rules and their anti-patterns in the light of our compliance checking approach introduced in Section 3. Finally, we give more details on how the temporal logic queries needed to specify certain anti-patterns are solved in Section 5.6.

5.1. Temporal Logic Querying

Temporal Logic Querying (TLQ) was first introduced by Chan in [26] in order to find software model invariants and gain better understanding of the model's behavior. In general, model checking can be seen as a sub-problem of temporal logic querying. In model checking, we issue Boolean queries only. In case of TLQ, we ask a TLQ solver (e.g. [27]) to find a propositional formula that would make our query hold true when seen as a temporal logic formula. The question mark '?' is used in a temporal logic query as a placeholder for such a propositional formula. Consequently, a query $\mathbf{AG}(?)$ might be used to find invariants in a model. We can even restrict our query by limiting it to formulae over certain predicates. For instance, the query $\mathbf{AG}(?\{p, q\})$ looks for invariants that are based on the predicates p and q.

We illustrate the application of temporal logic querying by an example CTL formula $\mathbf{AG}(x \wedge y \rightarrow \mathbf{AF}(b))$. It requires b to occur as a response to the occurrence of $x \wedge y$. Now, we assume that this formula is checked against some system and it fails. A temporal logic query $\mathbf{AG}(x \wedge y \wedge ? \rightarrow \mathbf{AF}(b))$ can be issued to investigate the reason for this failure. In this query, we ask for the condition to which b occurs as a response. Using a TLQ solver, it may turn out that this condition is given as $x \wedge y \wedge z$. Thus, model checking the CTL formula $\mathbf{AG}(x \wedge y \wedge z \rightarrow \mathbf{AF}(b))$ results in a positive answer.

5.2. Control Flow Anti-Patterns

For the case of control flow compliance patterns, the anti-patterns can be derived directly. In general, the anti-patterns are derived by negating the temporal logic formula of the original compliance pattern. For instance, the global scope presence pattern, cf., Figure 5(a), is equivalent to the CTL formula $AF(a)$. By negating this formula, i.e., $\neg AF(a)$, and using well-known equivalence rules we reach the formula $EG(\neg a)$. That means there is at least one possible execution of the process from the beginning to the end, such that activity a is never executed. This formula, in turn, can be represented as a BPMN-Q query, the actual anti-pattern.

5.2.1. Global-scope Violation

The global-scope *presence* requires that a certain activity must be executed in *all* instances of a process. Formally, the violation can be described by negating Formula 1. The formula that describes the violation for global-scope presence is

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

After applying well-known temporal logic equivalences, the formula is

$$\mathbf{EF}(\mathbf{start} \wedge \mathbf{EG}(\neg\mathbf{executed}(A))) \tag{21}$$

We see that Formula 21 declaratively describes the violation of the original compliance rule by identifying at least one possible execution of the process model, in which **start** occurs, whereas activity A is never executed thereafter. However, in the context of business process models, the execution has to eventually come to an end. Thus, it is sufficient to visualize the violation on the process model to find a path from the start of the process to its end without visiting activity A. This can be expressed in a structural BPMN-Q query, as illustrated in Figure 9. As a consequence, all process models that either lack any occurrences of activity A or provide means to skip the execution of any of the occurrences of activity A would be matched.

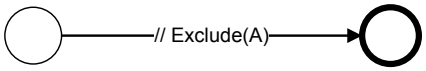


Figure 9: Global-scope Presence Anti-Pattern Expressed as a BPMN-Q query

For the case of a global-scope absence pattern, the corresponding anti-pattern has to match, if the respective activity is potentially executed. That is, the process model under consideration has at least one occurrence of the activity to be absent. Formally, the anti-pattern CTL formula is

$$\mathbf{EF}(\mathbf{start} \wedge (\mathbf{E}[\neg\mathbf{end}\ \mathbf{U}\ (\mathbf{executed}(A) \wedge \neg\mathbf{end})] \vee \mathbf{EG}(\neg\mathbf{end}))) \tag{22}$$

Formula 22 hints at two sources of a potential violation. First, activity A might be executed before the end of the process, i.e., $\mathbf{E}[\neg\mathbf{end}\ \mathbf{U}\ (\mathbf{executed}(A) \wedge \neg\mathbf{end})]$. Second, the process might not terminate at all, i.e, $\mathbf{EG}(\neg\mathbf{end})$. As we consider solely well-formed process models (cf., Section 2.1), the latter case can be neglected as processes are required to terminate eventually. Thus, we only have to consider the first source of potential violation. In particular, we have to match a path form the start of the process to an occurrence of activity A, the activity that should be absent according to the original compliance pattern, and from that activity to the process end. However, it is sufficient to show the path from the process start to the activity A as an evidence for the violation. This is captured by the query in Figure 10.

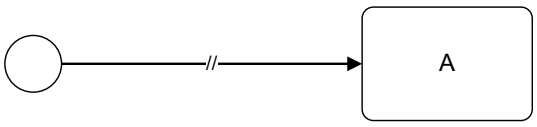


Figure 10: Global-scope Absence Anti-Pattern Expressed as a BPMN-Q query

5.2.2. Before-scope Violation

Before-scope presence patterns require that an activity A is always executed before another activity B. Therefore, a violation occurs when activity B might be executed without executing A before. Formally, the violation is defined as

$$\mathbf{E}[\neg\mathbf{executed}(A)\ \mathbf{U}\ \mathbf{ready}(B)] \tag{23}$$

To structurally locate this violation in the process model, we need to find at least one path from the start of the process to activity B without visiting activity A. This is shown in the anti-pattern query of Figure 11.

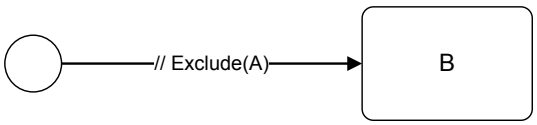


Figure 11: Before-scope Presence Anti-Pattern Expressed as a BPMN-Q query

For the before-scope absence pattern, the violation is defined as

$$\mathbf{EF}(\mathbf{start} \wedge \mathbf{EF}(\mathbf{executed}(A) \wedge \mathbf{EF}(\mathbf{ready}(B)))) \tag{24}$$

We see that the violation occurs when there is a chance to start the process, execute activity A, and eventually reach a state in which activity B can be executed. As the process is always started, any violation is actually caused by the dependency between executions of activity A and B. One might think of capturing this violation by a structural query containing a path from activity A to activity B as illustrated in Figure 12.

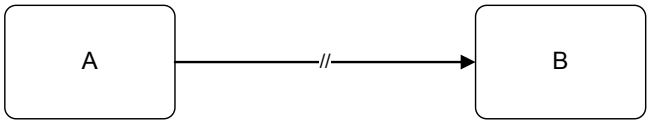


Figure 12: Incomplete Before-scope Absence Anti-Pattern

However, this anti-pattern does not match all process parts that might cause the violation. For instance, consider the process depicted in Figure 13, which violates the original compliance pattern, but does not match the structural query representing the anti-pattern in Figure 12. That is because the anti-pattern assumes that there is a structural path between activity A and activity B, whereas the example process defines a concurrent execution of the two activities. However, this must also be considered as a violation since they can be executed in an arbitrary order and, therefore, might violate the original compliance pattern. While it is not feasible to capture all structures that might lead to a concurrent execution, we can still approximate the violation with the anti-pattern illustrated in Figure 14.

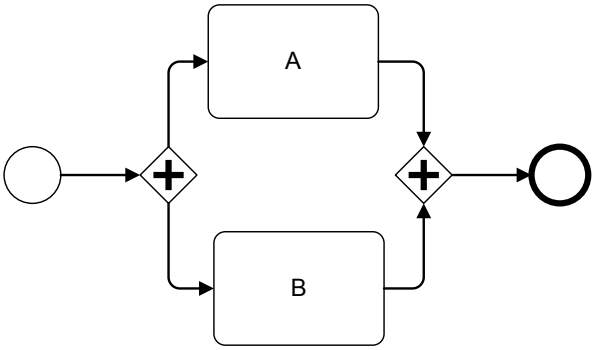


Figure 13: A Sample Non-compliant Process

This modified anti-pattern looks for a path form the start of the process to both activities. In case that there is a path to the target activity of the original query that contains the activity that should be absent, both paths of the anti-pattern query will overlap. On the other hand, if the two activities can be executed concurrently, both paths of the anti-pattern query will have some parts of the process model in common, whereas other parts represent the concurrent branches that lead to a potential violation.

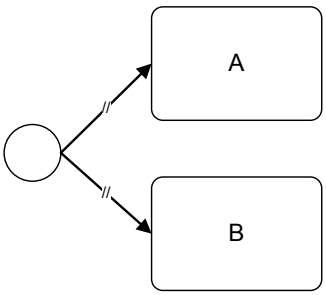


Figure 14: Approximate Anti-pattern for Before-scope Absence

5.2.3. After-scope Violation

For the after-scope presence pattern, violation occurs if in some process instance activity A is executed but never followed by an execution of activity B. The CTL formula for the after-scope presence anti-pattern is

$$\mathbf{EF}(\mathbf{executed}(A) \wedge \mathbf{EG}(\neg\mathbf{executed}(B))) \tag{25}$$

Any potential violation can be matched by the query depicted in Figure 15, where the subformula $\mathbf{EG}(\neg\mathbf{executed}(B))$ is structurally captured by a path from activity A to process end with the exclude property set to B. That is because the process terminates when it reaches the **end** state and no activities can be executed.

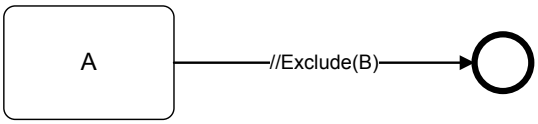


Figure 15: After-scope Presence Anti-Pattern

Further on, the after-scope absence pattern is violated, if activity B might be executed after activity A has been executed. Formally, the after-scope absence anti-pattern is

$$\mathbf{EF}(\mathbf{executed}(A) \wedge (\mathbf{E}[\neg\mathbf{end}\ \mathbf{U}\ (\mathbf{executed}(B) \wedge \neg\mathbf{end})] \vee \mathbf{EG}(\neg\mathbf{end}))) \tag{26}$$

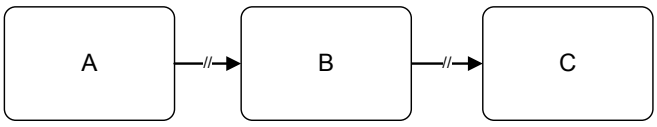
To detect the violation on the process model, we can reuse the anti-pattern query in Figure 14, as it detects A and B being executed either in sequence or in parallel.

5.2.4. Between-scope Violation

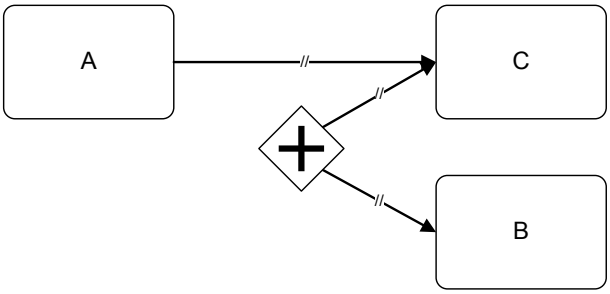
The violation of these patterns are variants of the anti-patterns discussed above. Violation of the between-scope absence (response with absence) pattern can occur in one of two cases. First, it might be the case that an execution of activity A is followed by an execution of activity B and C thereafter. Second, activity C may never be executed after activity A has been executed. Thus, the anti-pattern CTL formula is

$$\begin{aligned}\mathbf{EF}(\mathbf{executed}(A) \wedge (\mathbf{E}[\neg\mathbf{executed}(C)\ \mathbf{U}\\ (\mathbf{executed}(B) \wedge \neg\mathbf{executed}(C))] \vee \mathbf{EG}(\neg\mathbf{executed}(C))))\end{aligned} \tag{27}$$

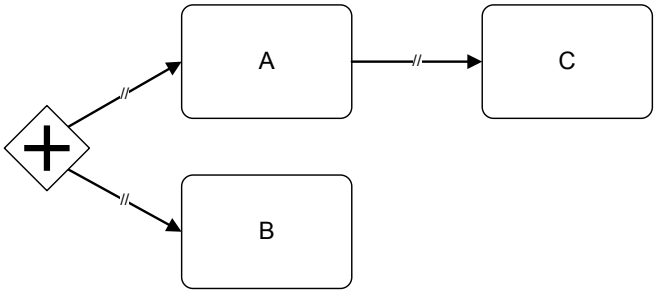
The first possibility of violation is declaratively described as the case that activity A executes; afterwards B executes, and finally C executes. There are many process model structures comprising A, B, and C that are capable of generating the respective execution trace. First, any process model containing the three activities A, B, and C in a sequence causes a violation. However, having activities A and B executing in parallel with C being executed afterwards, also satisfies the first part of the disjunction above. In the same vein, executing activities B and C in parallel with A being executed before, satisfies the first part of the disjunction. Finally, having all of them executed in parallel leads to the



(a) A, B and C execute in sequence



(b) B and C execute in parallel



(c) A and B execute in parallel

Figure 16: Between-scope Absence Type I Anti-Patterns

same result. However, executing activities A and C in parallel also satisfies the second part of the disjunction above. Thus, we can exclude this case from the set of potential causes for violation at this point.

Anti-pattern queries in Figure 16 detect the different cases of violation that are traced back to the first part of the disjunction. The anti-pattern query in Figure 16(a) detects the case where activities A, B, and C are part of a sequence. Figure 16(b) depicts an anti-pattern query that looks for a parallel execution of B and C, while activity A is executed before C. Note that this anti-pattern query matches process models in which activity A executes either before the parallel branches or in which it belongs to the same parallel branch as activity C. Finally, the anti-pattern query in Figure 16(c) detects the violation where A and B run in parallel and C executes after A. The matches to the last two anti-pattern queries are not disjoint. Moreover, the last anti-pattern matches any process model where C executes after joining all parallel branches or where it is executed within the same parallel branch as A.

The second possibility of violation is similar to the violation of the after-scope presence pattern. Hence, we can reuse the anti-pattern query in Figure 15. Note that this anti-pattern also matches processes where activities A and C are executed in parallel.

The case of the between-scope absence (precedence with absence) pattern is similar. First, the pattern is violated if activities A, B, and C are executed in a sequence. Second, if activity C is executed without having executed activity A before the original pattern is also violated. Formally, the between-scope absence (precedence with absence) anti-pattern is expressed in CTL as

$$\begin{aligned}&\mathbf{E}[\neg\mathbf{executed}(A)\ \mathbf{U}\ \mathbf{ready}(C)]\vee\\&\mathbf{EF}(\mathbf{executed}(A)\wedge\mathbf{EF}(\mathbf{executed}(B)\wedge\mathbf{EF}(\mathbf{ready}(C))))\end{aligned}\tag{28}$$

For the first source of potential violation, the anti-pattern query depicted in Figure 16 might be applied. Regarding

the second source of potential violation, the anti-pattern illustrated in Figure 11 can be applied, when replacing the activity B with activity C.

5.3. Data Flow Anti-Patterns

A data flow compliance pattern is violated, if there exists a state in which the respective activity is ready to execute, but the data condition is not fulfilled. Formally the data flow anti-pattern is

$$\mathbf{EF}(\mathbf{ready}(a) \wedge \neg dataCondition) \tag{29}$$

Thus, the data object that is referred to by the original compliance pattern assume a state that is different to those specified by the condition $dataCondition$. To discover these *undesired* data states, we issue the following temporal logic query against the behavioral model of the process model.

$$\mathbf{AG}(\mathbf{ready}(a) \rightarrow \mathbf{state}(d, ?_s)) \tag{30}$$

The temporal logic query in Formula 30 retrieves all states the data object assumes when $\mathbf{ready}(a)$ holds. Here, the symbol d is a placeholder for the data object that was mentioned in the original compliance pattern while $?_s$ is the placeholder for its states. In general, such a query delivers the different bindings of data object states that make the statement hold true. The general form of the query result is $\bigwedge_{d \in \mathcal{D}_\mathcal{Q}} (\bigvee_{s \in \mathcal{L}(d)} \mathbf{state}(d, s))$. Recall that each data object d is able to assume a finite set of states, i.e., S. Also, for each data object we assume a known $initial$ state.

Based on the set of states for a data object d at the time $\mathbf{ready}(a)$ holds, we can derive structural BPMN-Q anti-pattern queries. To explain this, let us assume a banking business process where the 'Risk' data object is used to assess the risk of opening a bank account. The 'Risk' object has the possible values $\{initial, low, high\}$. Now, consider the data flow compliance rule

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

Let us assume that the above rule is violated by the banking process model. Then, we issue the following temporal logic query

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

Solving this query leads to the following answer

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

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

The other two data states, namely *initial* and *high*, cause the violation of the original compliance rule. Here, having the value *initial* in the query answer tells us that there are execution path(s) where the "Risk" data object was not updated at all. That is, it keeps its initial state until the "Open Correspondent Account" activity is reached. To capture this situation on the process structure, we need to formulate an anti-pattern query where there is an execution path from the start of the process to the open account activity that excludes all activities updating the "Risk" object. It is easy to investigate the process model for those activities that update the "Risk" object. We call this set of activities $Update_{risk}$. With this information in hand, we can formulate an anti-pattern BPMN-Q query as shown in Figure 17. For our example, d is replaced with "Risk", A is replaced with "Open Correspondent Account", and $Update_{risk}$ corresponds to $update(d)$.

The other cases of violation could occur due to setting the data object to some *undesired* state that is kept until the activity in the compliance rule is reached. Regarding our example, the "Risk" object could be set to *high* and the value is kept until the activity "Open Correspondent Account" is reached. Thus, we need to find a path in the process model from some activity that sets the "Risk" to *high* to the open account activity. However, to highlight correct parts of the process, we have to be sure to exclude any activity on the path that sets the "Risk" to *low*. We refer to the set of activities updating a data object to a certain state as $Update_{(risk,low)}$. The anti-pattern for such case is shown in Figure 18 in its generic form.

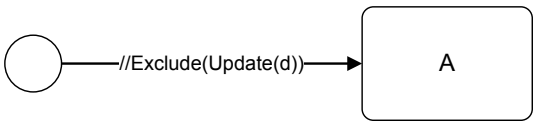


Figure 17: Anti-Pattern Query for Data Flow Violation

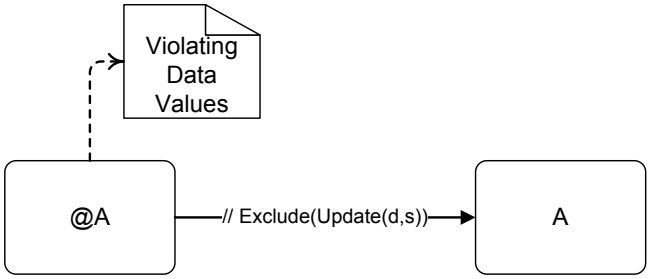


Figure 18: Anti-Pattern Query for Data Flow Violation

5.4. Conditional Flow Anti-Patterns

For conditional control flow compliance patterns, the generation of the anti-patterns might investigate process-specific violation conditions. However, in some cases, derivation of anti-patterns follows directly on the derivation of control flow anti-patterns as introduced above.

5.4.1. Conditional Response Violation

The violation of the conditional response pattern is similar to the violation of the global-scope presence and the after-scope presence patterns. That is, the violation occurs whenever the execution of the response activity might be skipped. Formally the anti-pattern can be expressed as

$$\mathbf{EF}((\mathbf{executed}(A) \wedge stableDataCondition) \wedge \mathbf{EG}(\neg\mathbf{executed}(B))) \tag{31}$$

We notice that the anti-pattern formula above detects a similar violation as Formula 21. Thus, an anti-pattern query would look for a path from the point where the condition occurred, i.e., $\mathbf{executed}(A) \wedge \bigvee_{s\in\mathcal{S}}(\mathbf{state}(d,s) \wedge \mathbf{AG}(\bigwedge_{s'\in CON_s} \neg\mathbf{state}(d,s')))$ to the end of the process that does not contain activity B. Such a query is illustrated in Figure 19.

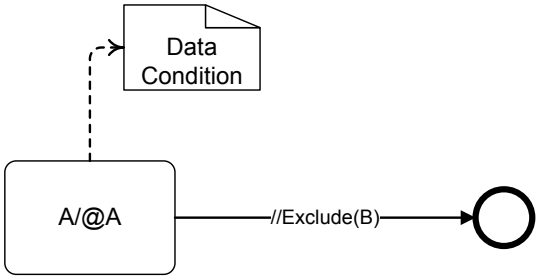


Figure 19: Anti-Pattern Query for Conditional Presence Violation

5.4.2. Conditional Precedence Violation

As discussed in Section 4.3.2, there are two variants of the conditional precedence. We referred to the first case as precedence to conditional activity execution (cf., Formula 14). Here, detection of potential violations resembles the one introduced for the before-scope presence pattern. That is, the violation occurs when the precedent activity

is not executed before at least one occurrence of the conditionally executed activity. Formally, the anti-pattern for precedence to conditional activity execution is defined as

$$\mathbf{E}[\neg\mathbf{executed}(A)\ \mathbf{U}\ \mathbf{ready}(B) \wedge dataCondition] \tag{32}$$

The anti-pattern query to detect this violation is shown in Figure 20. The path edge, excluding A, between the start event and activity B matches the part of the process model causing the violation. The other path connecting the anonymous activity with the data condition to activity B highlights the part of the process model where the conditional activity execution occurs.

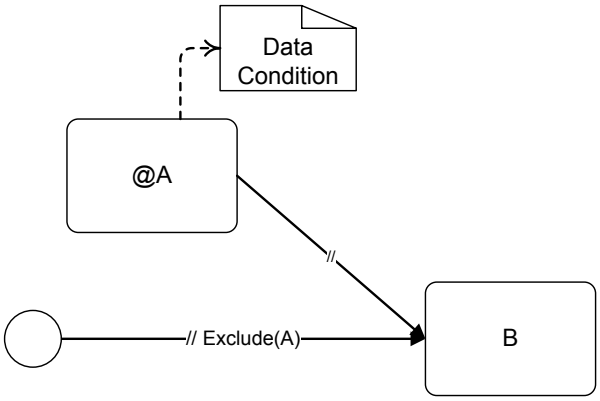


Figure 20: Anti-Pattern Query for Conditional Activity Execution Violation

Unlike precedence to conditional activity execution, detection of violations of conditional precedence (cf., Formula 15) requires querying the behavioral model of the investigated process model. Formally, the violation is defined as follows

$$\mathbf{E}[\neg(\mathbf{executed}(A) \wedge stableDataCondition)\ \mathbf{U}\ \mathbf{ready}(B)] \tag{33}$$

Recalling the meaning of the $stableDataCondition$ (cf. Formula 12) and according to Formula 33, the violation could occur in any of the following cases.

1. $\mathbf{executed}(A)\ \wedge\ (\bigvee_{s\in\mathcal{L}(d)} \mathbf{state}(d,s))$ did not occur before activity B is reached. That, in turn, might be traced back to one of the following reasons:
   (a) Activity A was not executed at all.
   (b) The data condition $\bigvee_{s\in\mathcal{L}(d)} \mathbf{state}(d,s)$ was not fulfilled.
2. $\mathbf{G}(\bigwedge_{s'\in CON_s} \neg\mathbf{state}(d,s'))$ was not fulfilled. That is, the state of the data object had been altered before activity B was ready to execute.

Please note that in case an anonymous activity was used in the rule, case 1a does not need to be investigated. In order to identify the exact reason for the violation, we have to issue a sequence of temporal logic queries. Depending on their results, we derive the query that shows how the violation occurred.

First and foremost, we check whether activity A is always executed before B. To this end, we check before-scope presence as introduced in Section 4.1.3. If it is satisfied, we know that data conditions are the cause of violation. On the other hand, in case of violation, we can use the anti-pattern of Figure 11 to highlight the violation on the level of the process model structure.

Second, we investigate potential violations of data conditions, i.e., cases 1b and 2 listed above. We first query the data values produced as a result of executing activity A by the following temporal logic query

$$\mathbf{EF}(\mathbf{executed}(A) \wedge \mathbf{state}(d,?_s) \wedge \mathbf{EF}(\mathbf{ready}(B))) \tag{34}$$

The temporal logic query of Formula 34 simply asks for states that can be assumed by data object d at the moment activity A executes, when eventually reaching activity B. The answer for this query would be of the form shown in Formula 9. For each of the identified data states that is not included in the original compliance pattern, we formulate an anti-pattern query as shown in Figure 21.

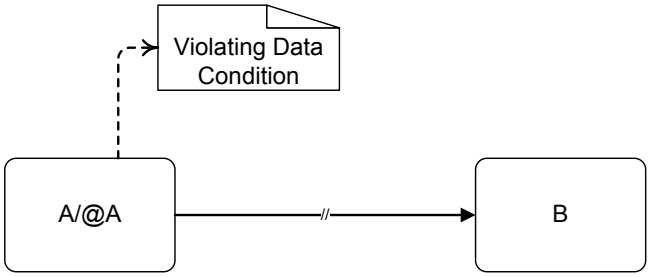


Figure 21: Anti-Pattern Query for Conditional Precedence

Finally, the violation could be due to the data object changing its state to any of the contradicting states. To discover those contradicting states, we issue the following temporal logic query.

$$\mathbf{EF}(\mathbf{executed}(A) \wedge \bigvee_{s \in \mathcal{S}} (\mathbf{state}(d, s) \wedge \mathbf{EF}(\mathbf{state}(d, ?_{s'}))) \wedge \mathbf{EFready}(B)) \tag{35}$$

This query asks the temporal logic query solver to find specific values for s' that would make the whole formula true. For each of the identified values for s', we can issue the anti-pattern query shown in Figure 22.

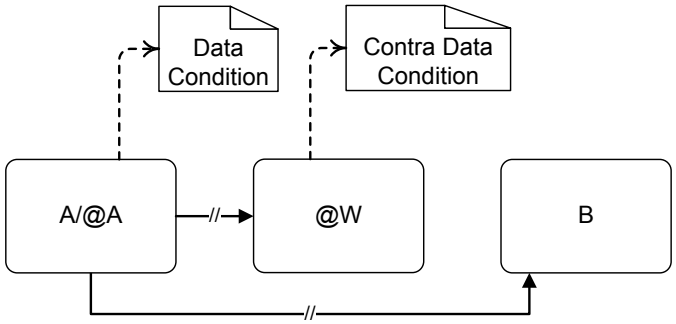


Figure 22: Anti-Pattern Query for Conditional Precedence

5.4.3. Conditional Before-scope Violation

The case of conditional before-scope presence violation was already covered in Section 5.4.2. Here, we discuss the violation of the conditional before-scope absence pattern. It requires that whenever an activity B is ready to execute under a certain condition, another activity A must have never been executed before. Thus, the violation is similar to the general before-scope absence control flow pattern, such that activity A might be executed before activity B. Formally, the anti-pattern CTL formula is

$$\mathbf{EF}(\mathbf{start} \wedge \mathbf{EF}(\mathbf{executed}(A) \wedge \mathbf{EF}(\mathbf{ready}(B) \wedge dataCondition))) \tag{36}$$

Following on the discussion given in Section 5.4.2, we see that activity A might not be in sequence with activity B. Rather, both activities might be executed concurrently. Therefore, the anti-pattern query in Figure 23 captures the cause of violation on the level of the process model structure. The two path edges from the start event to activities A and B, respectively, describe the paths that can cause the violation. The anonymous activity $@A$ attached to the data condition has a path edge to activity B in order to highlight the path in the process that leads to the execution of activity B under the data condition.

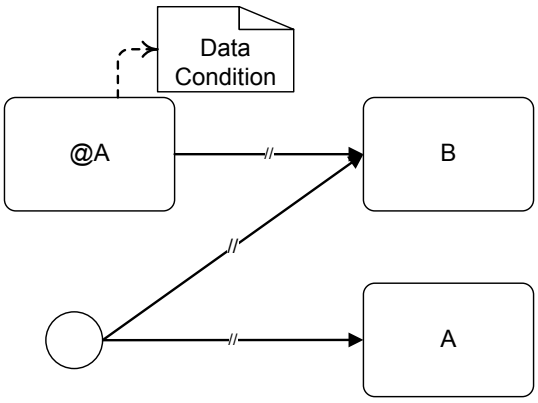


Figure 23: Anti-Pattern Query for Conditional Before-scope Absence

5.4.4. Conditional After-scope Violation

Violation of the conditional after-scope presence pattern was already discussed in Section 5.4.1. In this section, we discuss the case of the conditional after-scope absence violation. The CTL formula for such anti-pattern is

$$\begin{aligned}&\mathbf{EF}((\mathbf{executed}(A) \wedge stableDataCondition) \wedge \\ &(\mathbf{E}[\neg\mathbf{end}\ \mathbf{U}\ (\mathbf{executed}(B) \wedge \neg\mathbf{end})] \vee \mathbf{EG}(\neg\mathbf{end})))\end{aligned} \tag{37}$$

We see that the above formula resembles the one introduced for the violation of the control flow after-scope absence pattern, cf., Formula 26. The difference is that in the conditional case we require activity B to be absent only if activity A results in a certain data condition. Thus, we can derive the same anti-pattern query to detect the violation on the structure of the process, see Figure 14. The minor difference is that we need to highlight the resulting data condition of activity A, by attaching the data condition to the activity in the anti-pattern query.

5.4.5. Conditional Between-scope Violation

There are three different patterns to express conditional between-scope absence compliance rules (cf., Section 4.3.5).

The conditional between-scope absence type I pattern (cf., Formula 18) requires that when an activity C is ready to be executed and some $dataCondition$ holds, another activity A must have been executed before. Moreover, between activities A and C, activity B must have never been executed. The CTL formula for this anti-pattern is

$$\begin{aligned}&\mathbf{E}[\neg\mathbf{executed}(A)\ \mathbf{U}\ \mathbf{ready}(C) \wedge\ dataCondition] \vee \\ &\mathbf{EF}(\mathbf{executed}(A) \wedge \mathbf{EF}(\mathbf{executed}(B) \wedge \mathbf{EF}(\mathbf{ready}(C) \wedge\ dataCondition)))\end{aligned} \tag{38}$$

To detect the violation on the level of the process model, we reuse the anti-pattern queries for precedence to conditional activity execution, see Figure 20, and between-scope absence, see Figure 16, to detect the two possibilities of violation above respectively. However, we need to highlight the part of the process that makes the $dataCondition$ hold in the case of between-scope violation. We can do this in the same way as we did in Figure 23.

The violation of the conditional between-scope absence type II pattern (cf., Formula 19) occurs if the data condition does not hold once before reaching activity C or if activity B might be executed in between. Formally, the anti-pattern is captured in CTL as

$$\begin{aligned}&\mathbf{E}[\neg(\mathbf{executed}(A) \wedge stableDataCondition)\ \mathbf{U}\ \mathbf{ready}(C)] \vee \\ &\mathbf{EF}(\mathbf{executed}(A) \wedge stableDataCondition) \wedge \\ &\mathbf{EF}(\mathbf{executed}(B) \wedge \mathbf{EF}(\mathbf{ready}(C))))\end{aligned} \tag{39}$$

Thus, in part, the violation of this pattern is similar to the violation of the conditional precedence pattern discussed in Section 5.4.2, i.e, the first part of Formula 39. Therefore, the discussion on the derivation of anti-pattern queries for

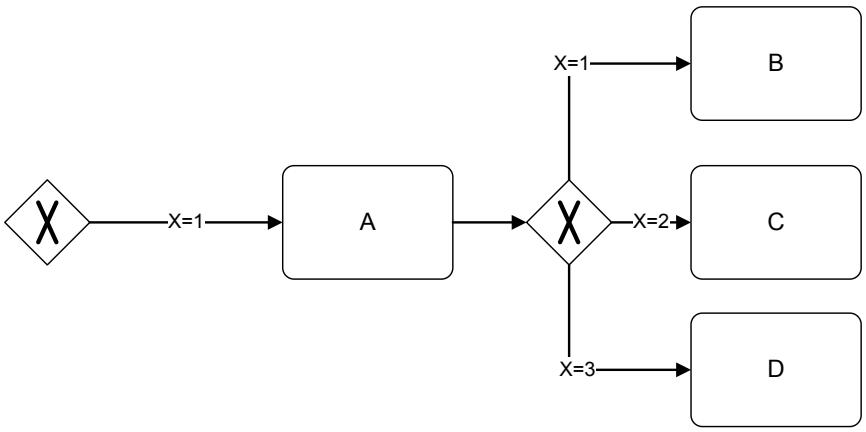


Figure 24: A process excerpt

the conditional precedence pattern is also valid in the current case. Consequently, all anti-pattern queries derived for the conditional precedence can be employed to detect potential violation of the conditional between-scope absence type II pattern.

To detect the other possibility of violation, i.e., the chance that activity B is executed in between, we can also reuse the anti-pattern queries in Figure 16. The change we need to do is to attach the data condition to activity A.

Finally, the violation of a conditional between-scope absence type III pattern is captured by the following CTL formula.

$$\begin{aligned}&\mathbf{EF}(\mathbf{executed}(A) \wedge stableDataCondition) \wedge \\ &(\mathbf{E}[\neg\mathbf{executed}(C)\ \mathbf{U}\ (\mathbf{executed}(B) \wedge \neg\mathbf{executed}(C))] \vee \\ &\mathbf{EG}(\neg\mathbf{executed}(C))))\end{aligned} \tag{40}$$

Informally, either activity B is executed in between or there is no chance to execute activity C after the $dataCondition$ holds. Again, the anti-pattern query in Figure 16 can be reused to detect the first case of violation. That is, activity A is executed and, thereafter, activities B and C might be executed in any order. In addition, we need to check for violations that relate to the data condition attached to activity A. Here, the anti-pattern for conditional presence can be reused, see Figure 19.

5.5. On the Relation between Compliance Rules and their Anti-Patterns

We see that all control flow anti-patterns are derived by negating the pattern's CTL formula and then describing this violation by means of a structural BPMN-Q query. Our approach of checking compliance as introduced in Section 3 proposes to use these structural queries solely after a violation is determined in the model checking phase. Thus, a question might be: Why not to match anti-patterns directly to the process model without doing model checking? Applying anti-patterns directly might lead to false alarms, which we illustrate with an example. Consider the process excerpt shown in Figure 24. Assume that we want to check a *A leads to B* compliance rule. Assuming that activity A does not change the value of x, the rule is satisfied by that process. However, if we match the anti-pattern from Figure 15 directly without model checking first, it would highlight the paths to activities C and D. To prevent these false alarms, we conduct model checking first. Once a violation is reported, we start with generating anti-patterns and matching them to the process [28].

On the other hand, imagine that activity A in Figure 24 changes the value of x to 2. In this case, applying model checking will indicate that the *A leads to B* rule is violated. Looking at the process model, we know that the violation occurs because the branch with condition $x = 2$ is executed. Hence, activity B is not executed. However, when matching the anti-pattern to the process model, both branches, for $x = 2$ and for $x = 3$, are highlighted. In this case, the branch with $x = 3$ is a false alarm. Still, it is not possible to have a structural match that is *only* a false alarm. There is always at least one match to the anti-pattern that captures the violation of the compliance rule.

In order to eliminate such false alarms, we need to conduct complex behavioral analysis to determine the branches that actually caused the violation. We aim at addressing this point in future work.

5.6. Solving Temporal Logic Queries

We have seen that for some data-dependent compliance rules it is necessary to investigate the process behavior for data conditions causing violation. To this end, we rely on temporal logic queries. In this section, we shed light on the evaluation of temporal logic queries.

Unfortunately, the temporal logic query solver reported on in [27] was not publicly available. However, according to [26, 29], it is possible to reduce the temporal logic query solving problem to $2^{2^{|AP|}}$ model checking problems with AP is the set of atomic propositions used to express properties. We took that direction to implement a problem-specific temporal logic query solver for the explanation of data-dependent rules violations as discussed in Sections 5.3 and 5.4. We reduce the problem of solving a temporal logic query solving problem into a linear number of model checking problems.

We need to issue temporal logic queries to explain violations of the following compliance rules:

1. Data flow rules.
2. Conditional precedence rules.
3. Conditional between-scope absence type II .

The cases 2 and 3 are similar, so that we restrict the discussion to cases 1 and 2.

5.6.1. Evaluating Data Flow Temporal Logic Queries

In Section 5.3 we showed that in order to discover violating data conditions we have to issue the temporal logic query of Formula 29. We have to find the values (states) a data object d may assume each time some activity A is ready to execute. Recall that at the execution of the business process, a data object can assume exactly one state. If an activity A is ready to execute once a data object d assume either the state $s1$ or $s2$, we conclude that $\mathbf{AG}(\mathbf{ready}(A) \rightarrow \mathbf{state}(d, s1) \vee \mathbf{state}(d, s2)$.

Following on this argument, we might issue up to $2^{|state_d|}$ model checking questions with $state_d$ representing the set of states the data object d can assume. These model checking questions would represent the different disjunctions of data object states of d. They are formalized as

$$\mathbf{AG}(\mathbf{ready}(a) \rightarrow \bigvee_{s \in sd} \mathbf{state}(d, s))$$

However, there is a chance to simplify this problem due to expressiveness of CTL as well as the characteristics of data states in our setting. Exploiting this knowledge, we issue much simpler model checking questions to answer the temporal logic query of Formula 29. The simpler form is

$$\mathbf{EF}(\mathbf{ready}(a) \wedge \mathbf{state}(d, s)) \tag{41}$$

With Formula 41, we check whether there is a reachable state where both propositions $\mathbf{ready}(a)$ and $\mathbf{state}(d, s)$ hold true. For each $s \in state_d$ where the corresponding formula is satisfied, $\mathbf{state}(d, s)$ is added to the disjunction representing the final answer to the temporal logic query (cf., Formula 9). Thus, we can answer the data flow anti-pattern temporal logic query by a linear number of model checking problems rather than the doubly exponential number in the general case.

5.6.2. Evaluating Conditional Precedence Temporal Logic Queries

For the case of conditional precedence (as well as conditional between-scope absence type II), we have to answer two temporal logic queries (cf., Formulae 34 and 35).

In order to solve the query of Formula 34, we follow the argument given in the previous section. Hence, we answer the query by means of issuing a linear number of model checking problems.

For the temporal logic query of Formula 35, we rely on the domain knowledge. Given a data state $\mathbf{state}(d, s)$, we know its set of contradicting data states. For each of these contradiction data states, we also issue a model checking problem.

6. Application

This section introduces a process model along with a set of compliance requirements related to this process model in order to illustrate the applicability of our approach. We focus on a process model from the financial domain. Figure 25 shows the process of opening a correspondent bank account. The process is expressed in BPMN.

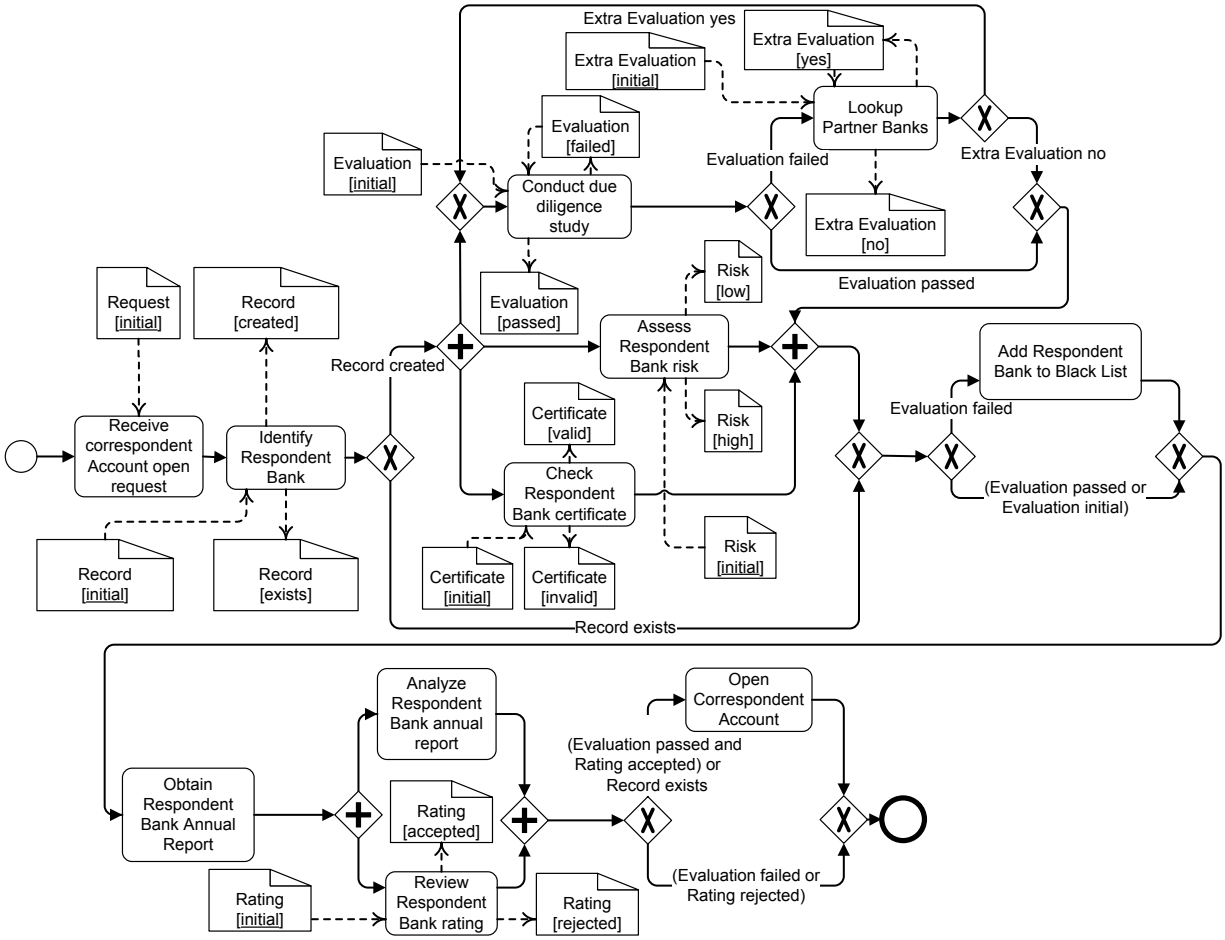


Figure 25: A process model to open a bank account

In general, this type of account is opened by a bank (the so-called respondent bank) in some country at another bank in another country to ease and speed up financial transactions. The process starts with the activity "Receive correspondent account open request". Then, the identity of the bank is looked up. If this is the first time that the respondent bank requests to open an account, a new record for that bank is *created* and some checks must take place. The bank that wants to open the account needs to conduct a study about the respondent bank due diligence ("Conduct due diligence study"), which might be *passed* or *failed* by the respondent bank. In case the respondent bank fails the evaluation of due diligence, the bank inquires one of its partner banks about the respondent bank ("Lookup Partner Banks"). Then, a decision is made whether to make an extra study. If so, the process loops back and repeats activity "Conduct due diligence study". Otherwise, processing proceeds directly. Concurrently to these checks, the risk of opening an account for that respondent bank is assessed ("Assess Respondent Bank risk") with the resulting risk categorized as either *high* or *low*. In the mean time, the respondent bank certificate is checked for *validity* in order to proceed with opening the account. Further on, if the due diligence study evaluation fails, the respondent bank is added to a black list.

If a respondent bank already has a record with the bank, all these checks are skipped. In any case, the bank has to obtain a report about the performance of the respondent bank ("Obtain Respondent Bank Annual Report"). This report is analyzed by the bank. Subsequently, the respondent bank rate is reviewed. If the respondent bank passes the checks, i.e., it passes the due diligence evaluation and its rating is accepted or there is already a record for the respondent bank, the account is finally opened.

The process described in Figure 25 is subject to the following compliance requirements regarding anti money laundering [14]:

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

In the remainder of this section, we show the application of the patterns described in Section 4 to express these compliance requirements. Subsequently, we check whether the given process is compliant with respect to these rules. We also show how violations of some of the rules can be explained following the approach introduced in Section 5.

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

This rule is a combination of multiple elementary compliance patterns. First, $R1$ requires both activities, "Obtain Respondent Bank Annual Report" and "Analyze Respondent Bank Annual Report", to occur in the process model. Second, it requires a certain order between them.[1] Thus, we visualize these compliance requirements as depicted in Figure 26.

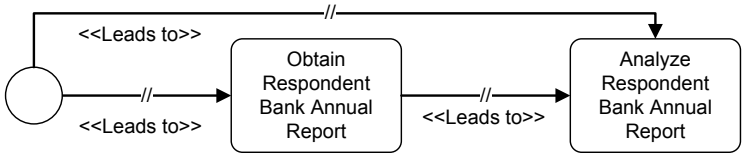


Figure 26: Representation of R1 in BPMN-Q

$R1$ is represented by applying a global-scope presence and after-scope presence pattern. The corresponding CTL formula is

$$\mathbf{AG}(\mathbf{start} \rightarrow \mathbf{AF}(\mathbf{executed}(Obtain\ Respondent\ Bank\ Annual\ Report)))\wedge$$
$$\mathbf{AG}(\mathbf{start} \rightarrow \mathbf{AF}(\mathbf{executed}(Analyze\ Respondent\ Bank\ Annual\ Report)))\wedge$$
$$\mathbf{AG}(\mathbf{executed}(Obtain\ Respondent\ Bank\ Annual\ Report) \rightarrow$$
$$\mathbf{AF}(\mathbf{executed}(Analyze\ Respondent\ Bank\ Annual\ Report)))$$

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

$R2$ can be represented by a conditional response pattern as shown in Figure 27. Activity "Add Respondent Bank to Black List" is the response to the condition that the "Evaluation" *fails*.

By investigating the process model in Figure 25, we find out that activity "Conduct due diligence study" is responsible for updating the "Evaluation" data object to state *fails*. So, the CTL formula for $R2$ is

$$\mathbf{AG}(\mathbf{executed}(\text{Conduct due diligence study}) \wedge \mathbf{state}(Evaluation, failed)\wedge$$
$$\mathbf{AG}(\neg\mathbf{state}(Evaluation, passed)) \rightarrow \mathbf{AF}(\mathbf{executed}(Add\ Respondent$$
$$Bank\ to\ Black\ list)))$$

[1] Whether or not the order is required is subject to interpretation. However, such interpretation of the informal requirements, e.g., legislation text, into formal requirements is beyond the scope of this paper.

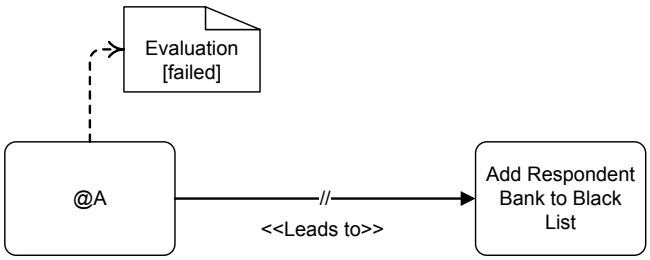


Figure 27: Representation of R2 in BPMN-Q

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

This rule can be represented as a data flow pattern, see Section 4.2. It is represented by the BPMN-Q query as shown in Figure 28, which requires the "Risk" data object to assume state low when the activity "Open Correspondent Account" is about to execute. To this end, an implicit data input edge is modeled between the data object and the activity.

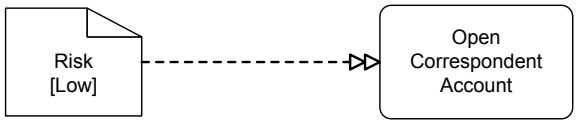


Figure 28: Representation of R3 in BPMN-Q

The CTL formula for $R3$ is

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

$R4$: *If it is the first time to deal with the respondent bank, an advanced due diligence study must be conducted*

This rule can be also represented as a conditional response pattern. Dealing with the respondent bank the first time is reflected in the state *created* of the data object "Record".

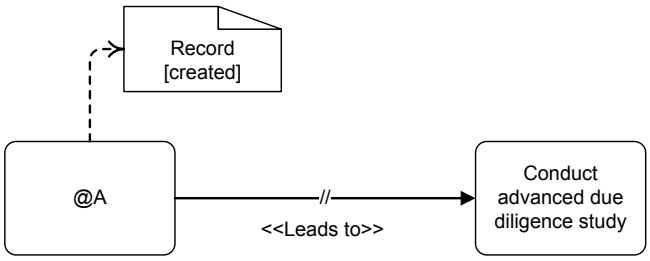


Figure 29: Representation of R4 in BPMN-Q

By finding out that activity "Identify Respondent Bank" is responsible to setting the "Record" data object to *created*, The CTL formula for $R4$ is

$$\mathbf{AG}(\mathbf{executed}(\text{Identify Respondent Bank}) \wedge \mathbf{state}(Record, created) \rightarrow \mathbf{AF}(\mathbf{executed}(Conduct\ advanced\ due\ diligence\ study)))$$

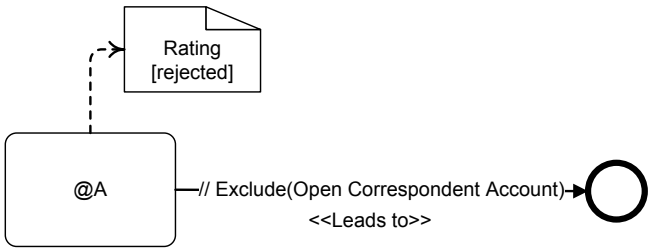


Figure 30: Representation of R5 in BPMN-Q

$R5$: *If the respondent bank rating is rejected, an account must never be opened*

This rule can be modeled using the conditional after-scope absence pattern, see Section 4.3.4. Figure 30 represents this rule as a BPMN-Q query.

For $R5$, the CTL formula is

$$\mathbf{AG}(\mathbf{executed}(\text{Review Respondent Bank rating}) \wedge \mathbf{state}(Rating, rejected) \wedge \mathbf{AG}(\neg\mathbf{state}(Rating, accepted)) \rightarrow \mathbf{A}[\neg\mathbf{executed}(Open\ Correspondent\ Account)\ \mathbf{U}\ \mathbf{end}])$$

$R6$: *Before opening an account, the respondent bank certificate must be valid*

This rule is modeled using the conditional precedence pattern. Figure 31 visualizes the respective BPMN-Q query.

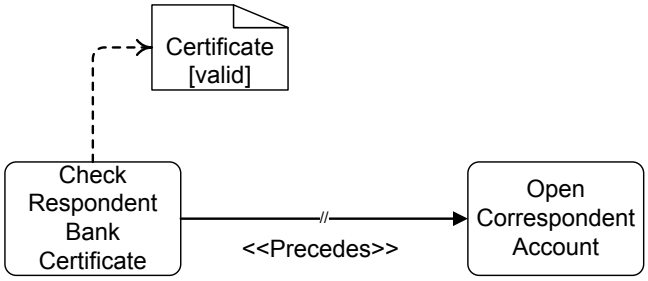


Figure 31: Representation of R6 in BPMN-Q

The CTL formula for $R6$ is

$$\neg\mathbf{E}[\neg(\mathbf{executed}(Check\ Respondent\ Bank\ certificate) \wedge \mathbf{state}(Certificate, valid) \wedge \mathbf{AG}(\neg\mathbf{state}(Certificate, invalid)))) \mathbf{U}\ \mathbf{ready}(Open\ Correspondent\ Account)]$$

Checking the Rules Against the Process Model

Following on the processing as introduced in Section 3, we model check the given compliance requirements against the behavioral model of the process model depicted in Figure 25. This yields the following result.

- $R1$ is satisfied.
- $R2$ is satisfied.
- $R3$ is not satisfied.
- $R4$ is not satisfied.
- $R5$ is satisfied
- $R6$ is not satisfied.

We see that three out of six compliance rules are not satisfied. In the remainder of this section, we show how the violation of these rules is explained by deriving appropriate anti-patterns.

Explaining Violations of $R3$

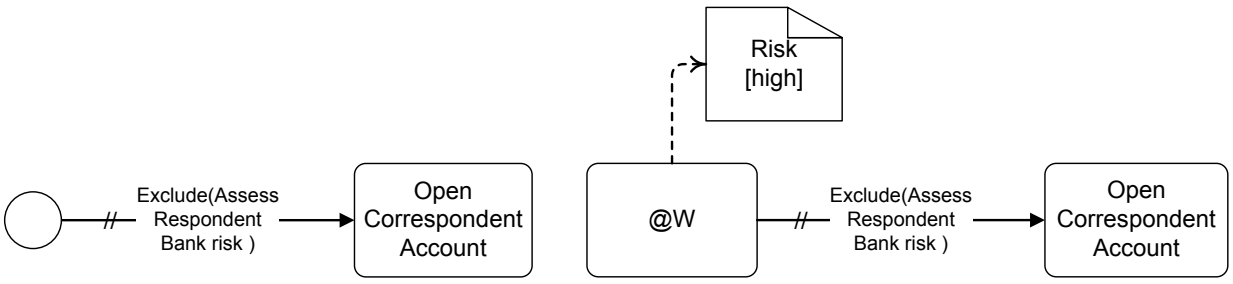
The rule $R3$ states that "Opening an account must be of low risk" and has been represented by the BPMN-Q query shown in Figure 28. In order to explain why this data flow pattern is not satisfied by the model in Figure 25, we derive anti-patterns as discussed in Section 5.3. Therefore, we issue the following temporal logic query.

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

Evaluating this temporal query against our example process yields the following answer.

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

We exclude the value $\mathbf{state}(Risk, low)$ from the answer above, since it is the one originally included in the compliance rule. Based on the remaining two values, i.e., $\mathbf{state}(Risk, initial)$ and $\mathbf{state}(Risk, high)$, we generate two anti-pattern queries that are depicted in Figure 32.



(a) Anti-Pattern Query for $(Risk, initial)$ value

(b) Anti-Pattern Query for $(Risk, high)$ value

Figure 32: Anti-Pattern Queries for $R3$

The structural matching of the query in Figure 32(a) to the open account process model illustrated in Figure 25 is visualized in Figure 33. This match indicates that it is possible to open a correspondent account where the "Risk" object is kept to its *initial* value. This is possible in the case that a request is received from a respondent bank, for which there exists a record already.

Further on, matching of the query illustrated in Figure 32(b) highlights the another possibility of violating the compliance rule $R3$. In this case, the risk is assessed. However, it is still possible to open the account while "Risk" is *high*, as highlighted in the process model in Figure 34.

Note that this result nicely shows the drawbacks of applying a model checking counter-example directly for explaining violations. The highlighted part represents two execution scenarios of the process that are non-compliant. Any counter-example, however, would only represent one of these scenarios. Thus, the feedback would be incomplete, which might lead to a second model checking iteration after the first flaw has been fixed.

Explaining Violations of $R4$

Rule $R4$ has been expressed using the conditional response pattern. Thus, an anti-pattern should match a path from the point where the condition holds to the process end that does not contain the activity "Conduct advanced due diligence study". The corresponding anti-pattern is shown in Figure 35. The anti-pattern would match the whole process after activity "Identify Respondent Bank" indicating that all execution paths from that point do not visit the "Conduct advanced due diligence study" activity. Due to space restrictions, we do not show the matching part of the process model.

Explaining Violations of $R6$

Compliance rule $R6$ is modeled as a conditional precedence. The rule is violated. Thus, following the approach discussed in Section 5.4.2, we have to check several conditions in order to derive the respective anti-patterns.

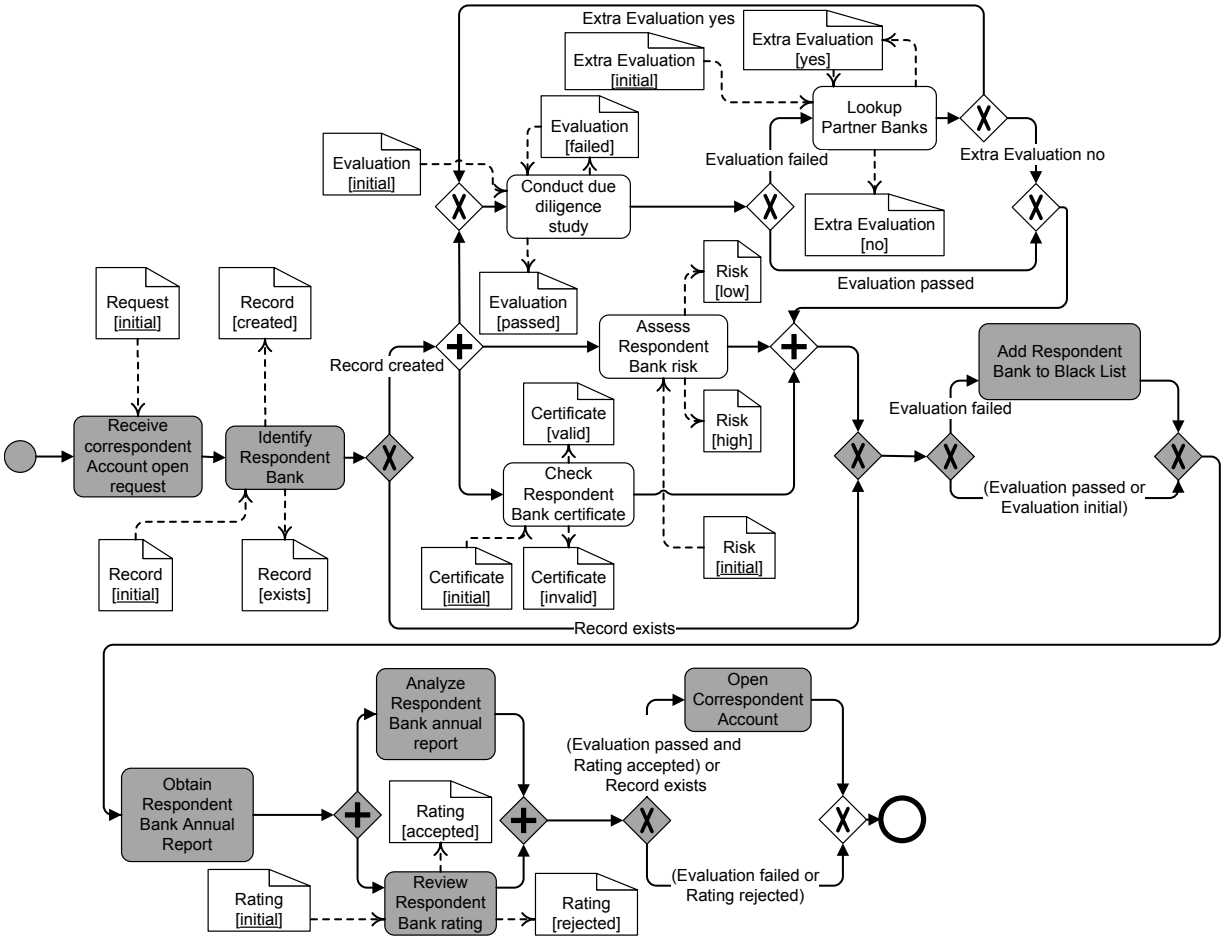


Figure 33: An Execution Scenario that violates R3

First, we check whether the activity "Open Correspondent Account" is always preceded by the "Check Respondent Bank certificate" activity. As this check fails, we derive the anti-pattern query that is depicted in Figure 36. It matches the process part that represents the skipped execution of activity "Check Respondent Bank certificate" before reaching "Open correspondent Account".

Second, we also check for data-dependent violations. Hence, we issue the following temporal logic query.

$$\mathbf{EF}(\mathbf{executed}(Check\ Respondent\ Bank\ certificate) \wedge \mathbf{state}(Certificate, ?_s) \wedge \mathbf{EF}\ \mathbf{ready}(Open\ Correspondent\ Account))$$

Applied to our example, this temporal logic query yields the following result.

$$\mathbf{state}(Certificate, valid) \vee \mathbf{state}(Certificate, invalid)$$

Based thereon, we derive the anti-pattern query that is depicted in Figure 37.

The matching part of our example process model to this anti-pattern query indicates that it is possible to execute activity "Check Respondent Bank certificate" with a result of an *invalid* "Certificate". Yet, the activity "Open Correspondent Account" activity is still reachable. Again, the matching part is not visualized due to space restrictions.

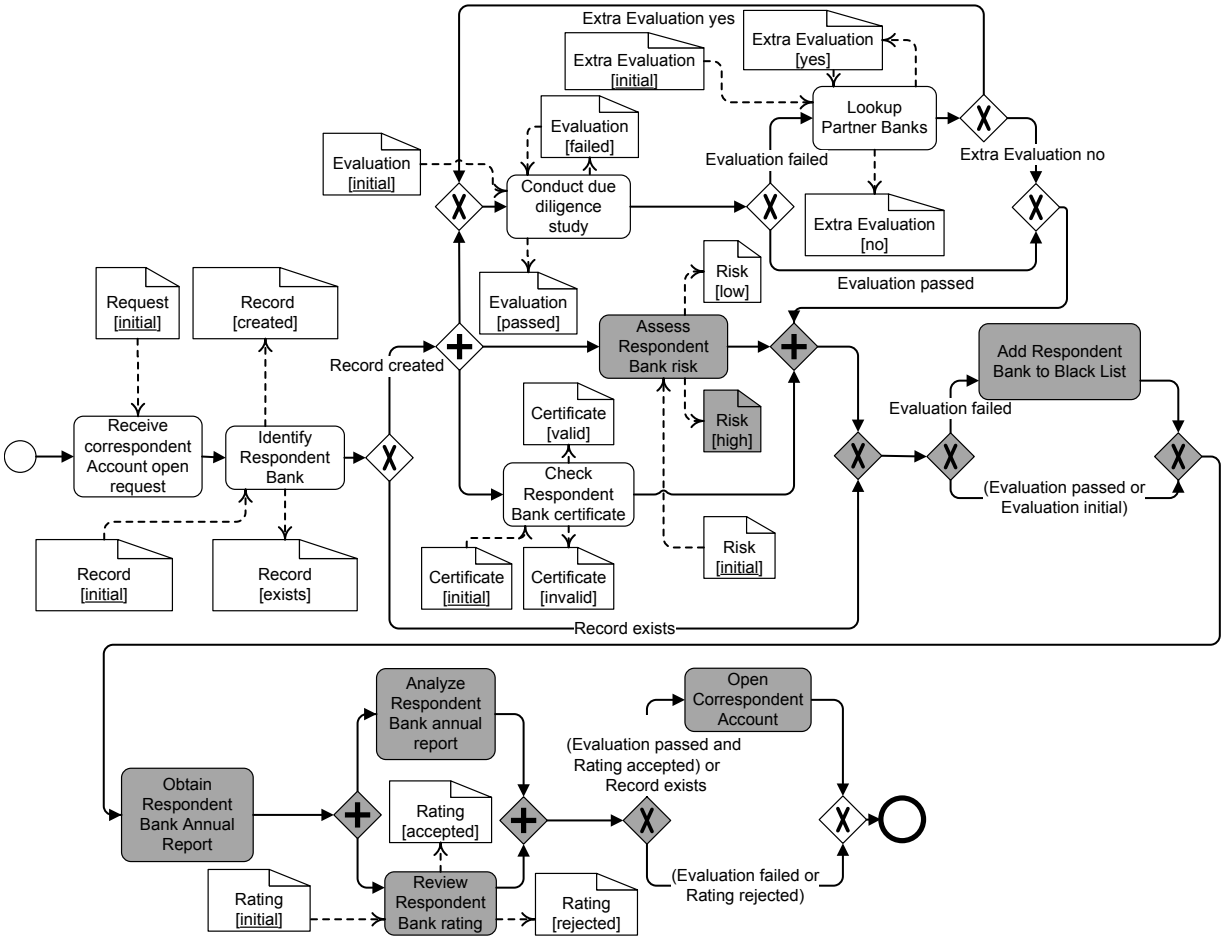


Figure 34: Another Execution Scenario that violates R3

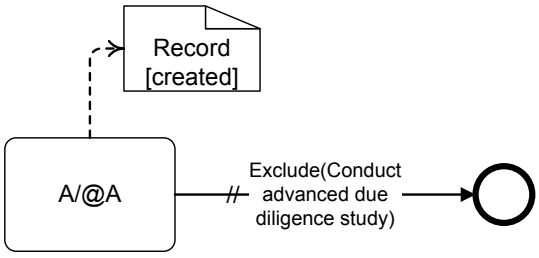


Figure 35: Anti-Pattern Query for R4

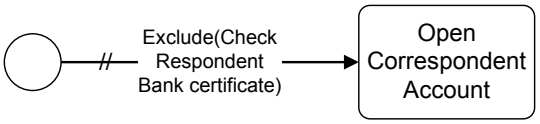


Figure 36: Anti-Pattern Query for R6

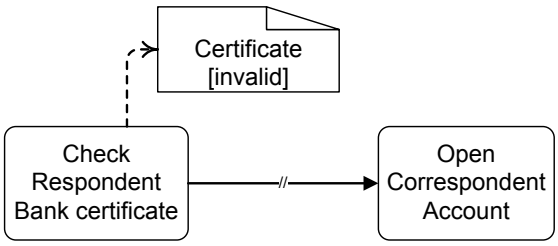


Figure 37: Another Anti-Pattern Query for R6

7. Implementation

The approach introduced in this paper has been implemented and integrated within Oryx[2], a web based process modeling and repository platform. At the front-end side, we developed a stencil set to model BPMN-Q queries visually. At the back-end side, we implemented the BPMN-Q query processor which uses the LoLA[3] model checker and controls the analysis. The latter follows the steps outlined in Figure 3.

Starting with the BPMN-Q stencil set, the BPMN-Q query processor implements the mapping of the introduced rules into their corresponding CTL formulae. On the other hand, we allow for checking these rules against BPMN process models within the model repository provided by Oryx. To prepare these process models for model checking, the mapping proposed in [19, 22] is used to generate a corresponding Petri net that captures the control flow along with the data aspects. The generated Petri net is then passed to LoLA along with the CTL formulae for model checking.

From the user perspective, our implementation works as follows. A user opens the Oryx editor within her browser. The user then models a compliance rule as a BPMN-Q query. In a next step, the user triggers the compliance check by using a toolbar button. At that time, the query is serialized and transfered to the back-end, where the identification of relevant process models and the actual check are done. Finally, the user receives the feedback in the form of small previews on the compliance status of each process model of the repository that was considered to be relevant to the query.

Figure 38 shows a screenshot of the BPMN-Q query editor. Figure 39 provides a detailed feedback of the violation of a process model to the rule in Figure 38.

The implementation has been used for checking compliance of process models with up to 100 activities. For this class of process models, our implementation works efficiently. That is, verification of a compliance rules and visualization of potential violation is done within milliseconds.

8. Related Work

There is a large amount of research concerned with compliance checking of business process models. In the following, we focus on approaches that support visual specification of compliance requirements, support automated checking, and provide feedback to process analysts in case of violation.

In [30, 31], Förster et al. proposed the Process Pattern Specification Language (PPSL) as an approach to visually express quality, compliance, constraints regarding the process behavior. PPSL is an extension of UML Activity Diagrams [17] that extends edges with the <<after>>stereotype to indicate that two activities are not necessarily are required to be in strict order. Moreover, they are able to express complex constraints with all types of logical connectors. Similar to our work, PPSL patterns are translated into temporal logic that is model checked against the process model. Regarding the expressiveness of specification language, however, PPSL does not provide means to specify absence patterns. Moreover, the approach covers only control flow rules, whereas data aspects and conditional control flow rules are neglected. Finally, PPSL does not support the notion of anti-patterns. In contrast to our work, explanation of violation to the process analyst is not tackled.

[2] http://oryx-project.org/
[3] http://service-technology.org/lola/

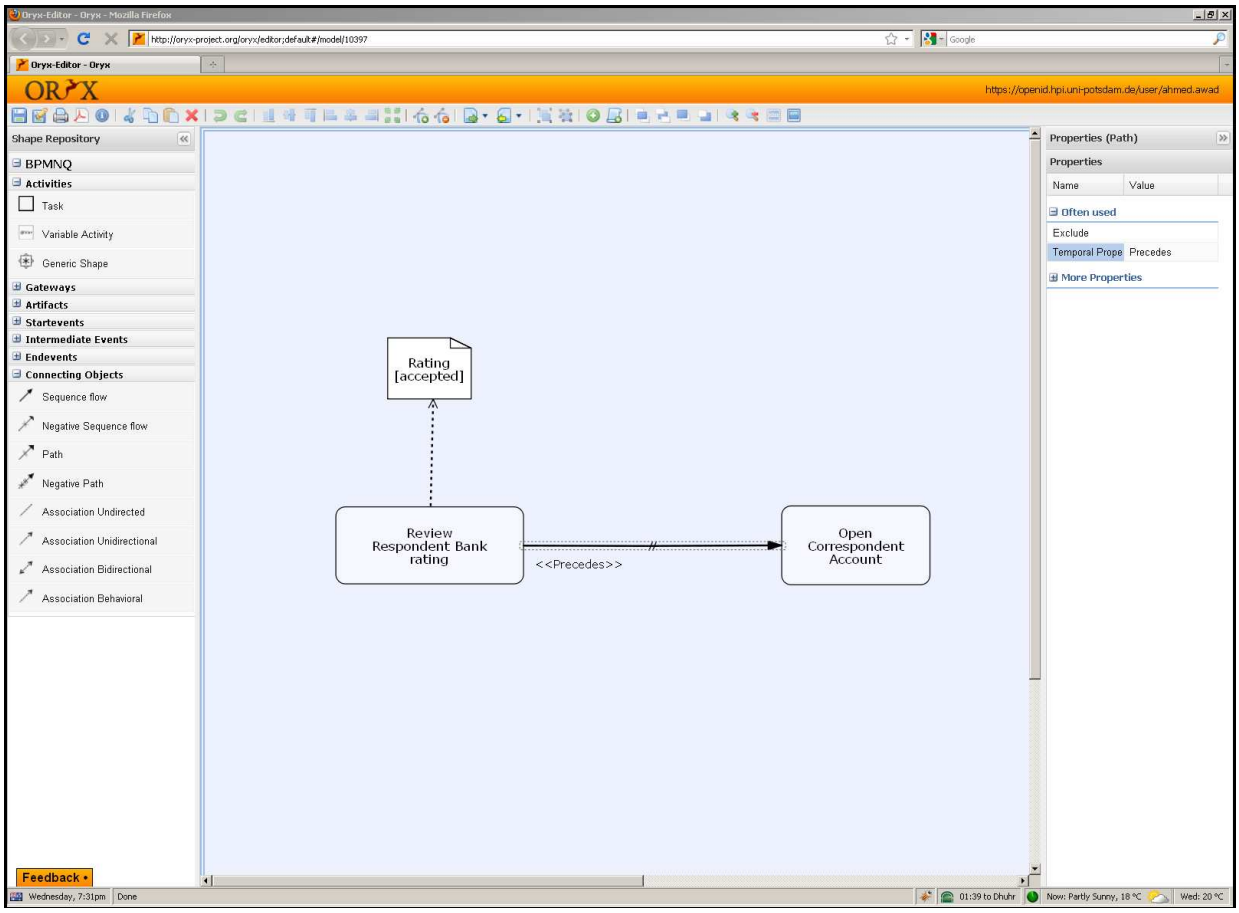


Figure 38: Screenshot of the BPMN-Q query editor

Moreover, the Business Property Specification Language (BPSL) [32] is another proposal for a visual specification language that has been developed by IBM. In [33], Liu et al. use BPSL to express compliance requirements for business process models. Although the language allows for expressing compliance rules that consider both, data and control flow aspects, the notion of anti-patterns and explanation of compliance violation is not addressed.

In [34], Feja et al. provide a visual language to express semantic correctness criteria of business process models. Their language is merely a visualization of CTL operators. That is, temporal operators, logical operators, and path quantifiers are given a visual notation. The process analyst is intended to use this visual notation to compose the temporal logic formulae. We follow on a different approach that aims at hiding the complexity of temporal logic languages from the process analyst. We assume our pattern based approach to enable the process analyst to focus on specifying properties. G-CTL, in turn, assumes process analysts to be aware of the details of CTL.

Work on declarative business process modeling languages [35, 36, 37, 38] is also related to our work. These languages are intended for the creation of business process models and provide the modeler with a set of visual patterns, which are then mapped into an LTL formula. The collection of pattern realizations, so-called constraints, declaratively describe how a business process could be executed. While the objective behind our approach is different, there are many commonalities. However, data-related aspects that are addressed by our approach are typically beyond the scope of declarative business process modeling languages. Instead, these languages focus on control flow constraints only.

Approximate compliance checking for annotated process models is discussed in [39, 40]. The authors provide an approach that is in general neither sound nor complete. That is, in some cases they are able to find *some* of the violations. In some other cases they cannot identify non-compliance. Also, the verification algorithms are limited to process models without loops. The verification is in a form of theorem proving and thus needs to propagate the effects

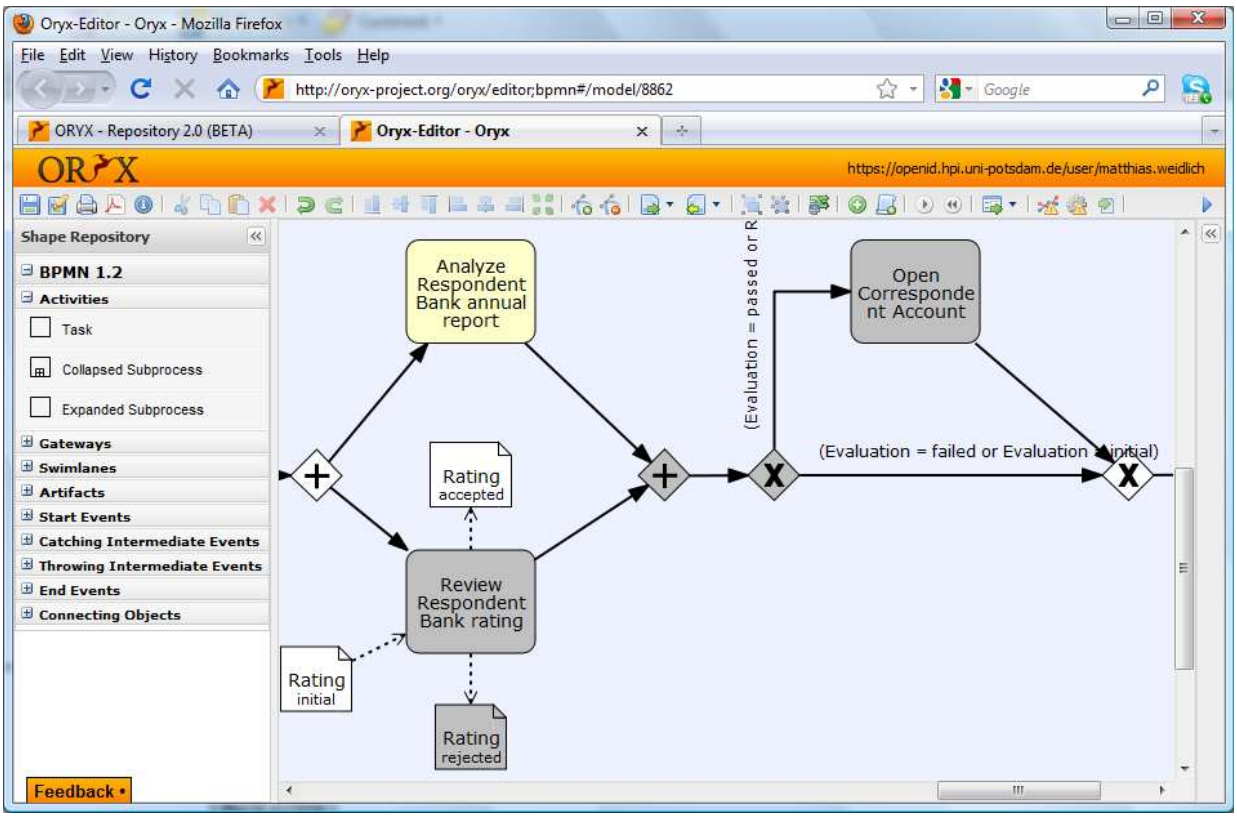


Figure 39: A screenshot showing the highlighting of the model fragment that causes compliance violation

of activities throughout states. One merit of that approach is the ability to trace back from the violating state to the activity that caused the violation. However, this is only useful to compliance rules of local nature, e.g., restrictions on resource allocations like separation of duty. However, violation to execution ordering compliance rules cannot benefit from this diagnosis feature. The approach seeks compliance rules with effects local to specific activities within the process.

Regarding the above mentioned approaches, we see that the unique contribution of our approach based on BPMN-Q is not only the way complexity of temporal logic formulae is hidden in a graphical notation. Rather, its application as a query language to highlight parts of the process model that caused violation is a feature that, to the best of our knowledge, has not been presented before.

Finally, visualization of violations has also been addressed in the context of domain-independent verification criteria. To this end, soundness of process models has received a considerable attention in the area of business process management. Soundness of a process guarantees the absence of behavioral anomalies, such as deadlocks and livelocks. Further on, there exist approaches that explain, i.e., visualize, problematic areas of the process in case it is identified to be not sound. In [41], Flender and Freytag translated the output of verification tools back to the level of the process structure. Moreover, BPMN-Q has also been used to develop structural anti-pattern queries, which, based on heuristics, declaratively describe process modeling anomalies [42, 43].

9. Conclusion

In this paper, we presented an end-to-end pattern based approach for the specification of compliance requirements. Compliance patterns are visually represented as BPMN-Q queries. Each pattern also represents a temporal logic formula to enable automated verification of compliance requirements against process models. BPMN-Q uses a visual representation that is similar to that used to develop process models. This visual representation has two major

advantages. First, process analysts can define compliance requirements without detailed background knowledge on formal specification languages. Second, the visual constructs for patterns are quite similar to those used for process modeling. Although an empirical test is beyond the scope of this paper, we can expect the learning curve of BPMN-Q to be quite low. That is due to the fact that the query constructs are very close to the constructs used to specify process models.

When a compliance requirement is not satisfied, we introduced an approach to provide feedback in a comprehensible way. That is, anti-patterns are derived from the compliance patterns. Each anti-pattern is also represented by a BPMN-Q query and is structurally matched to the process model in order to highlight parts whose execution causes the violation.

We can notice that specification of compliance patterns depends on the *all* paths quantifiers in CTL. That is because, compliance properties are required to hold in every possible execution. On the other hand, the *existential* path quantifier was sufficient to capture the anti patterns formally. With respect to the set of temporal operators used, we see that all but the next time operator $\mathbf{X}$ have been applied in our approach. That is due to the fact that compliance requirement typically do not require statements about immediate execution dependencies (execution of one activity is immediately followed by the execution of another activity).

The support for reasoning about data aspects makes our approach capable of addressing a wider range of compliance rules compared to other approaches. Most existing approaches consider solely control flow aspects.

The patterns introduced in this paper are not meant to be complete. Although we followed existing classifications of compliance requirements [10], there might be compliance requirements that have not yet been discussed in BPMN-Q. However, it is important to see that complex compliance requirements can be specified by combining multiple of the introduced patterns. If it turns out that certain real-world compliance rules cannot be covered by the current set of rules, however, we assume that our approach is easy to extend. In future work, we aim at investigating potential limitations of our approach in terms of expressiveness in more detail. Moreover, we want to address a process modeling perspective that has been neglected in our approach. In particular, we want to address resource assignment compliance rules in future work.

References

[1] Sarbanes-Oxley Act of 2002, Public Law 107-204, (116 Statute 745), United States Senate and House of Representatives in Congress, 2002.

[2] Basel Committee on Banking Supervision, Basel II Accord (2004).

[3] W. van der Aalst, A. Weijters, Process-Aware Information Systems: Bridging People and Software through Process Technology, Wiley & Sons, 2005, Ch. Process Mining, pp. 235–255.

[4] W. v. d. Aalst, Verification of Workflow Nets, in: Application and Theory of Petri Nets 1997, volume 1248 of LNCS, Springer Verlag, Berlin, 1997, pp. 407–426.

[5] J. Dehnert, P. Rittgen, Relaxed Soundness of Business Processes, in: K. R. Dittrich, A. Geppert, M. C. Norrie (Eds.), CAiSE, Vol. 2068 of Lecture Notes in Computer Science, Springer, 2001, pp. 157–170.

[6] F. Puhlmann, M. Weske, Investigations on soundness regarding lazy activities, in: BPM 2006, LNCS 4102, Springer, 2006, pp. 145–160.

[7] E. M. Clarke, O. Grumberg, D. A. Peled, Model Checking, MIT Press, 1999.

[8] A. Awad, BPMN-Q: A Language to Query Business Processes, in: EMISA, Vol. P-119 of LNI, GI, 2007, pp. 115–128.

[9] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design patterns: elements of reusable object-oriented software, Addison-Wesley Professional, 1995.

[10] M. B. Dwyer, G. S. Avrunin, J. C. Corbett, Patterns in Property Specifications for Finite-State Verification, in: ICSE, 1999, pp. 411–420.

[11] A. Awad, G. Decker, M. Weske, Efficient Compliance Checking Using BPMN-Q and Temporal Logic, in: M. Dumas, M. Reichert, M.-C. Shan (Eds.), BPM, Vol. 5240 of Lecture Notes in Computer Science, Springer, 2008, pp. 326–341.

[12] A. Awad, M. Weidlich, M. Weske, Specification, Verification and Explanation of Violation for Data Aware Compliance Rules, in: L. Baresi, C.-H. Chi, J. Suzuki (Eds.), ICSOC/ServiceWave, Vol. 5900 of Lecture Notes in Computer Science, 2009, pp. 500–515.

[13] A. Awad, M. Weske, Visualization of compliance violation in business process models, in: S. Rinderle-Ma, S. W. Sadiq, F. Leymann (Eds.), Business Process Management Workshops, Vol. 43 of Lecture Notes in Business Information Processing, Springer, 2009, pp. 182–193.

[14] F. S. Commission, Guidelines On Anti-Money Laundering & Counter-financing of Terrorism (2007).

[15] Business Process Modeling Notation 1.2 (BPMN 1.2) Specification, Final Adopted Specification., technical report, OMG, 2009.

[16] G. Keller, M. Nüttgens, A. Scheer, Semantische Prozessmodellierung auf der Grundlage "Ereignisgesteuerter Prozessketten (EPK)", Tech. Rep. 89, Institut für Wirtschaftsinformatik, Saarbrücken (1992).

[17] OMG, UML 2.0 superstructure specification, Tech. rep. (2004).

[18] W. Reisig, Petri Nets, EATCS Monographs on Theoretical Computer Science Edition, Springer, 1985.

[19] R. M. Dijkman, M. Dumas, C. Ouyang, Semantics and analysis of business process models in BPMN, Inf. Softw. Technol. 50 (12) (2008) 1281–1294. doi:http://dx.doi.org/10.1016/j.infsof.2008.02.006.

[20] W. M. P. van der Aalst, Workflow verification: Finding control-flow errors using petri-net-based techniques, in: W. M. P. van der Aalst, J. Desel, A. Oberweis (Eds.), Business Process Management, Vol. 1806 of Lecture Notes in Computer Science, Springer, 2000, pp. 161–183.

[21] N. Lohmann, E. Verbeek, R. M. Dijkman, Petri Net Transformations for Business Processes - A Survey, T. Petri Nets and Other Models of Concurrency 2 (2009) 46–63.
[22] A. Awad, G. Decker, N. Lohmann, Diagnosing and repairing data anomalies in process models, in: S. Rinderle-Ma, S. Sadiq, F. Leymann (Eds.), BPM 2009 Workshops, Vol. 43 of Lecture Notes in Business Information Processing, Springer-Verlag, 2009, pp. 5–16.
[23] E. A. Emerson, J. Y. Halpern, Decision Procedures and Expressiveness in the Temporal Logic of Branching Time, J. Comput. Syst. Sci. 30 (1) (1985) 1–24.
[24] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, A. Yakovlev, N. R. England, Petrify: a Tool for Manipulating Concurrent Specifications and Synthesis of Asynchronous Controllers, IEICE Transactions on Information and Systems 80 (1997) 315–325.
[25] J. Mendling, 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 (May 2007).
[26] W. Chan, Temporal-Logic Queries, in: CAV, Vol. 1855 of LNCS, Springer, 2000, pp. 450–463.
[27] M. Chechik, A. Gurfinkel, TLQSolver: A Temporal Logic Query Checker, in: CAV, Vol. 2725 of LNCS, Springer, 2003, pp. 210–214.
[28] S. Sakr, A. Awad, A framework for querying graph-based business process models, in: M. Rappa, P. Jones, J. Freire, S. Chakrabarti (Eds.), WWW, ACM, 2010, pp. 1297–1300.
[29] G. Bruns, P. Godefroid, Temporal logic query checking, in: LICS '01: Proceedings of the 16th Annual IEEE Symposium on Logic in Computer Science, IEEE Computer Society, Washington, DC, USA, 2001, p. 409.
[30] A. Förster, G. Engels, T. Schattkowsky, Activity Diagram Patterns for Modeling Quality Constraints in Business Processes, in: L. C. Briand, C. Williams (Eds.), MoDELS, Vol. 3713 of Lecture Notes in Computer Science, Springer, 2005, pp. 2–16.
[31] A. Förster, G. Engels, T. Schattkowsky, R. V. D. Straeten, Verification of Business Process Quality Constraints Based on Visual Process Patterns, in: TASE, IEEE Computer Society, 2007, pp. 197–208.
[32] K. Xu, Y. Liu, C. Wu, BPSL Modeler – Visual Notation Language for Intuitive Business Property Reasoning, Electron. Notes Theor. Comput. Sci. 211 (2008) 211–220. doi:http://dx.doi.org/10.1016/j.entcs.2008.04.043.
[33] Y. Liu, S. Müller, K. Xu, A static compliance-checking framework for business process models, IBM Syst. J. 46 (2) (2007) 335–361.
[34] S. Feja, D. Fötsch, S. Stein, Grafische validierungsregeln am beispiel von epks, in: W. Maalej, B. Brügge (Eds.), Software Engineering (Workshops), Vol. 122 of LNI, GI, 2008, pp. 198–204.
[35] W. M. P. van der Aalst, M. Pesic, DecSerFlow: Towards a Truly Declarative Service Flow Language, in: M. Bravetti, M. Núñez, G. Zavattaro (Eds.), WS-FM, Vol. 4184 of Lecture Notes in Computer Science, Springer, 2006, pp. 1–23.
[36] M. Pesić, W. M. P. van der Aalst, A Declarative Approach for Flexible Business Processes Management, in: J. Eder, S. Dustdar (Eds.), Business Process Management Workshops, Vol. 4103 of Lecture Notes in Computer Science, Springer, 2006, pp. 169–180.
[37] M. Pesić, H. Schonenberg, W. M. P. van der Aalst, DECLARE: Full Support for Loosely-Structured Processes, in: EDOC, IEEE Computer Society, 2007, pp. 287–300.
[38] M. Pesić, Constraint-Based Workflow Management System: Shifting Control to Users, Ph.D. thesis, Technische Universiteit Eindhoven (2008).
[39] I. Weber, G. Governatori, J. Hoffmann, Approximate Compliance Checking for Annotated Process Models, in: Proceedings of the first international workshop of governance, risk and compliance in information systems GRCIS, Vol. 339, CEUR-Workshop, 2008, pp. 46–60.
[40] J. Hoffmann, I. Weber, G. Governatori, On compliance checking for clausal constraints in annotated process models, Information Systems Frontiers Special Issue on Governance, Risk, and Compliance, available online.
URL http://www.springerlink.com/content/u340646302- q7hm7w/
[41] C. Flender, T. Freytag, Visualizing the Soundness of Workflow Nets, in: 13th Workshop Algorithms and Tools for Petri Nets, AWPN 2006, 2006, pp. 47–52.
[42] A. Awad, F. Puhlmann, Structural Detection of Deadlocks in Business Process Models, in: W. Abramowicz, D. Fensel (Eds.), BIS, Vol. 7 of Lecture Notes in Business Information Processing, Springer, 2008, pp. 239–250.
[43] R. Laue, A. Awad, Visualization of Business Process Modeling Anti Patterns, in: P. Bottoni, E. Guerra, J. de Lara, T. Margaria, J. Padberg, G. Taentzer (Eds.), Proceedings of the 1st International Workshop on Visual Formalisms for Patterns (VFfP 2009), Corvallis, OR (USA), 2009.
URL http://journal.ub.tu-berlin.de/index.php/eceas- st/article/viewFile/344/331