Cluster Comput (2015) 18:1503–1526
DOI 10.1007/s10586-015-0494-0

Runtime self-monitoring approach of business process compliance in cloud environments

Ahmed Barnawi[1] · Ahmed Awad[2] · Amal Elgammal[2] · Radwa El Shawi[3] · Abdullah Almalaise[1] · Sherif Sakr[4]

Received: 31 July 2015 / Revised: 12 September 2015 / Accepted: 14 September 2015 / Published online: 13 October 2015

Abstract Recently, several industrial studies have concluded that compliance management is one of the major challenges companies face nowadays. In practice, runtime compliance monitoring is of utmost importance for compliance assurance as during the design-time compliance checking phase, only a subset of the imposed compliance requirements can be statically checked due to the absence of required variable instantiation and contextual information. Furthermore, the fact that a business process model has been statically checked for compliance during design-time does not guarantee that the corresponding running business process instances are usually compliant due to human and machine errors. The problem of runtime monitoring of business process compliance becomes more challenging when business processes are executed in cloud computing environments. In this context, the compliance process can not rely on external components as the whole execution environment is mainly controlled by the cloud providers. In this article, we propose a novel approach to tackle this problem by adopting and configuring the business process models into a form that augment the associated compliance rules so that they can be monitored without the need to rely on external monitoring components. Compared to approaches that depend on an external monitoring component, our approach requires less sophisticated infrastructure when hosted on the cloud as well as less traffic footprint as communication with an external component for monitoring is no longer needed.

Keywords Cloud computing · Cloud monitoring · Business process compliance

✉ Ahmed Awad
a.gaafar@fci-cu.edu.eg

Ahmed Barnawi
ambarnawi@kau.edu.sa

Amal Elgammal
a.elgammal@fci-cu.edu.eg

Radwa El Shawi
rmelshawi@pnu.edu.sa

Abdullah Almalaise
aalmalaise@kau.edu.sa

Sherif Sakr
sakrs@ksau-hs.edu.au

[1] King Abdulaziz University, Jeddah, Saudi Arabia

[2] Cairo University, Giza, Egypt

[3] Princess Nourah Bint Abdulrahman University, Riyadh, Saudi Arabia

[4] King Saud bin Abdulaziz University for Health Sciences, Riyadh, Saudi Arabia

1 Introduction

Compliance management has become a pressing concern for organizations operating in all industrial sectors, especially heavily regulated domains such as the financial, pharmaceutical and manufacturing sectors. In today's business environment, organizations are continuously required to cope with an increasing number of compliance constraints originating from various sources, including laws and regulations (such as Sarbanes-Oxley, US patriot act, HIPAA), standards and code of practice (such as ISO 9001), internal policies, and business partner contracts (such as service level agreements-SLAs).

This is causing significant problems for organizations in almost all industrial sectors, as the complexities of hard and soft regulations are little understood or appreciated [7]. For example, banking regulations such as anti-Money Launder-

ing directives are generally complex and far-reaching, with a raft of major banks found to be not in compliance in 2012. Standard Chartered Bank, London, and HSBC Holdings Plc. for example, were fined a total of $459 million and $1.92 billion, respectively, in 2012.[1] These incidents were preceded by scandals and business failures such as Enron, and WorldCom back in 2001. Subsequently, much attention has been paid to compliance management from both the academic and the industrial communities. Many research efforts have contributed to the compliance management of business processes. The vast majority of these contributions focus on compliance checking at process design time [4,19,20,32], to name just a few. Although compliance checking at design-time is of crucial importance to identify and resolve as many as non-compliance scenarios, however, it does not provide a guarantee that the execution of cases based on *compliant* models will also be compliant. This is due to the complexity of the runtime phenomenon and the existence of several external factors such as human performers of activities within a process. Enforcing compliance at process configuration time has gained little focus [39] and still does not guarantee compliant execution especially to compliance requirements related to timing constraints. Post-mortem analysis of execution history automates the auditing and identification of violations but with no chance to remedy these violations [8,55,57].

In practice, compliance monitoring at process execution time is of crucial importance as it complements the design- and configuration-time checking with techniques to detect violations that are hard or even impossible to address at the earlier stages of the process lifecycle, e.g., time span constraints between tasks, and it also saves the effort of running several checks over the process logs to identify violations. Moreover, it allows for proactive management of violations where a violation scenario could be avoided. Recently, compliance monitoring of business processes has begun to gain more attention from the research community [3,10,31,33–35,56] where various approaches were introduced to accomplish the task of compliance monitoring. In most of these approaches, an external monitoring component needs to be built and linked to the process execution environment. The monitoring component usually listens to event streams originating from the process execution environment where technologies like complex event processing (CEP) [30] are used to check the compliance status of the running instances against different compliance rules.

As the cloud computing model is gaining in prominence and increasingly being adopted by organizations of all sizes, it is inevitable that business processes supporting these organizations will also be executed on the cloud in a distributed environment. Yet, these processes have to adhere to compliance requirements from various and diverse compliance sources. Moreover, large volumes of events are expected to be generated as a direct impact of running these processes. According to the NIST definition [38], cloud computing support three levels of *service models*: (1) *Infrastructure as a Service (IaaS)* which describes the provisioning process of computing resources such as servers, network bandwidth, storage, and related tools which are necessary to build an application environment from scratch. (2) *Platform as a Service (PaaS)* which provides a higher-level environment where developers can write customized software applications. (3) *Software as a Service (SaaS)* which refers to special-purpose softwares that can be made available through the Internet. In these models, especially, PaaS and SaaS, the monitoring facilities of the cloud users are quite limited, if any. In particular, the full access control on the cloud computing resources and software is ultimately on the provider side. Thus, the possibility of achieving the business process compliance via external monitoring components becomes quite limited.

In this paper, we propose a novel approach that allows reporting compliance status of *timing* and *resource assignment* compliance rules directly by the process instance itself without the need to have an external monitoring component. Compared to approaches that depend on an external monitoring component [2,10,31,33–35,56], our proposed approach requires less sophisticated infrastructure when hosted on the cloud as well as less traffic footprint as communication with an external component for monitoring is no longer needed. In particular, our approach is targeted to BPMN processes as they have a rich set of constructs that can be exploited to report about compliance status as will be shown later. The contributions of this paper are:

- An automatic pattern-based approach to rewrite process models to include violation monitoring logic within the process, where *compliance patterns* are high-level abstractions of frequently used compliance requirements, which help non-technical users to abstractly represent desired properties and constraints.
- Coverage of patterns that lend them selves to runtime detection like timing and resource assignment constraints,
- The approach is mainly designed for BPMN 2.0 processes but applicability to BPEL 2.0 is also discussed as they are the two standard process execution languages,
- An integrated tool-suite has been developed as a proof-of-concept that ascertains the implementability of the proposed approach,
- Validation of the approach using a case study from the financial sector about anti money laundering,

[1] http://www.accuity.com/industry-updates/free-resources/trends-in-aml-compliance-infographic/.

The rest of this paper is structured as follows: Background concepts are discussed in Sect. 2. The contribution of the paper through a pattern-based process rewriting is discussed in details in Sect. 3. Implementation and evaluation on a case study are discussed in Sect. 4. Related work is presented and discussed in Sect. 5, by comparing to the approach proposed in this paper. Finally, Sect. 6 concludes the paper with a critical discussion of the presented approach.

2 Background

This section introduces the main concepts and techniques that form the groundwork for our approach.

2.1 Compliance patterns

In general, pattern-based modeling of compliance rules is well accepted in the community and several studies have provided a comprehensive set of patterns that cover the different aspects as control flow, data flow, resource allocation and timing as summarized by Ly et al. [31]. In this work, we build on top of those patterns. In particular, Fig. 1 summarizes the set of compliance patterns which are supported by our framework. We use *pattern* and *rule* interchangeably where a rule is an instantiation of a pattern.

In practice, any pattern can optionally be limited to a scope in which the rule is required to hold. The scope represents a time window that is bounded by case or task instance-related events. In principle, the default scope is the whole process instance execution. Also, the pattern can be refined by a condition where the rule is required to hold only when this condition is true. The condition may refer to process execution data that are reflected in the event data payload. With each pattern, two actions are defined. The *Violation Action* describes the action taken by the monitoring component when the violation occurs whereas the *Prediction Action* describes the action taken when there is a possibility of violation. The nature of the action depends on how the monitoring component is integrated with the execution environment. For instance, the simplest action that can be taken is to alert administrators.

Definition 1 (*Atomic Compliance Rule*) Let PM be the set of all process models to be monitored. A compliance rule is a tuple $(pattern, model, antecedent, consequent, condition, scope\ start, scope\ end, multiplicity, WA, time\ span, alert\ time\ span, isWithin, violation\ action, predictiveaction, role, user)$ where:

- $pattern \in \{Exists, Absence, Sequence, Next, Precedes, One\ to\ one\ precedes, Response, One\ to\ one\ response, SoD, BoD, Performed\ by\ role, Performed\ by\ resource\}$ defines the pattern from which the rule is instantiated,
- $model \in PM$ is a reference to the process model against which the rule has to be monitored,
- $antecedent \in \{ex, not(ex) | ex \in RE\}$ where $not(ex)$ means that event ex has not been observed,
- $consequent \in \{ex, not(ex) | ex \in RE\}$ where $not(ex)$ means that event ex has not been observed,
- condition is the data condition that is to be examined at the occurrence of the rule's $antecedent$
- $scope\ start \in RE$ defines the delimiting start event of the rule's scope,
- $scope\ end \in RE$ defines the delimiting end event of the rule's scope,
- $multiplicity$ is a constraint on the number of occurrences of the rule's $antecedent$,
- $WA \subset RE$ is the set of events that must not occur between the $antecedent$ and $consequent$,
- $time\ span$ is the time window in/out of which the $consequent$ event must be observed,

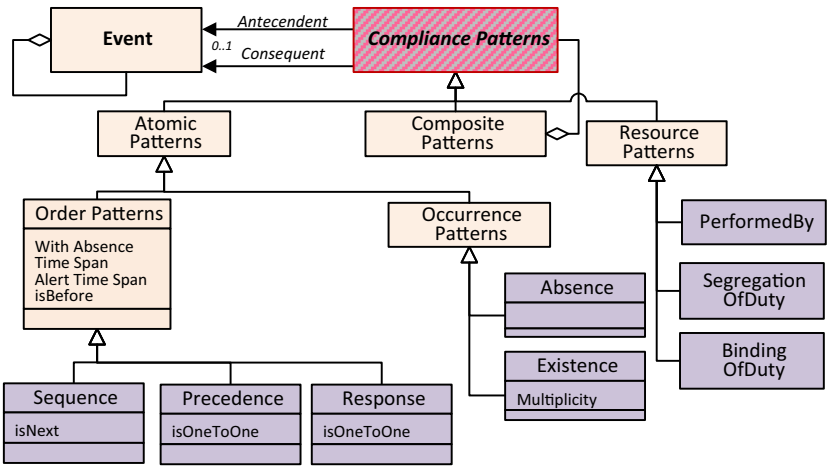


Fig. 1 Compliance patterns

– $alert\ time\ span$ is the time window after which there is a possibility of violation if the *consequent* was not observed,
– $isWithin \in \{true, false\}$ is a Boolean value indicating wether the *consequent* event must be observed *before* or *after* the end of the $time\ span$
– $violation\ action \in \{alert, suspend\}$ defines the action to be taken upon the occurrence of a violation,
– $predictive\ action \in \{alert, suspend\}$ defines the action to be taken when there is a possibility of a violation,
– $role$ defines the user role that is referred to when $pattern = Performed\ by\ role$,
– $user$ defines the user that is referred to when $pattern = Performed\ by\ resource$.

When any property does not apply to a rule pattern, it is represented as $\perp$. We define CR as the set of all compliance rules registered with the monitoring component.

As per Definition 1, there might be a *time span* window that puts further constraints on the observation of the *consequent* event with respect to the *antecedent* event. This is also further controlled by the *isWithin* property. So, if $isWithin = true$, the pattern requires that the *consequent* event to be observed *before* time span elapses otherwise there is a violation. Whereas, if $isWithin = false$, then *consequent* has to be observed *after* the time span elapses.

Composite patterns are used to logically connect other patterns by Boolean operators AND, OR, NOT, etc. This is used to define complex rules that can not be expressed merely by atomic patterns, which is especially helpful when sub-ideal level of compliance is also needed [31].

2.2 Process runtime APIs

Our approach relies on the following set of application programming interface (APIs) which are commonly supported by the runtime business process execution environments:

– `GetProcessInstanceID(TaskInstance)`: Returns the process instance identifier in which a task instance is currently running.
– `GetCompletionTimestampOfTask(Task, Instance)`: Returns from execution history the `Date Time` value at which the specified `Task` was completed within the specified `Instance`.
– `GetPerformerOfTask(Task, Instance)`: Returns the user object who has completed an instance of task `Task` in process instance `this`.
– `GetRolesOfUser (User)`: Returns the set of roles assigned to the user `User`.

Examples for these APIs are available in the documentation of the open source BPMN execution engine, Activiti,[2] and the commercial execution engine, Camunda.[3]

3 Rewriting processes to alert for violations

In this section, we describe our approach for adapting the business process model design and configuration in order to support self-alerting for compliance violations at runtime in cloud environments. Section 3.1 provides an overview of our approach and the flow of steps that link between business process models and compliance rules in order to produce a violation-aware business process models which are ready for execution. In Sect. 3.2, we describe in detail our contribution with respect to adopting the business process model for alerting compliance violations when they occur at runtime with respect to *timed order pattern*. In Sect. 3.3, we describe adopting the business process models to alert for runtime violations of resource-related patterns.

3.1 Overview

We follow a pattern-based approach which is based on the compliance patterns discussed in Sect. 2. As can be noted, the patterns can be verified in two phases, design time and runtime. We assume that design time verification has been done and the model is statically compliant by design. However, in practice, not all compliance requirements can be checked during design-time due to the lack of some contextual information and variable instantiation that is only available during the runtime phase of the business process lifecycle [2,22]. Therefore, in this article, we address only patterns that need further verification at runtime. To be more specific, we address *order patterns* that have either the *time span* or the *alert time span* in addition to resource patterns (See Sect. 2.1). Both timing information and user assignments to tasks and activities are usually not known until runtime.

As shown in Fig. 2, there are basically two threads, the rule modeling and the process modeling threads. A compliance expert can visually specify the compliance rules based on patterns described in Sect. 2.1. These rules are then stored in the repository. Similarly, a process/business expert defines the logic of a BPMN process model that is also saved to the repository. Both the rule and the process model are the inputs to the component "*Self-Alerting Logic of BP Augmentation*". Actually, an arbitrary number of compliance rules can be taken as input in addition to the process model. This component applies changes to the input process model based

[2] http://www.activiti.org/javadocs/index.html.
[3] http://docs.camunda.org/latest/api-references/javadoc/.

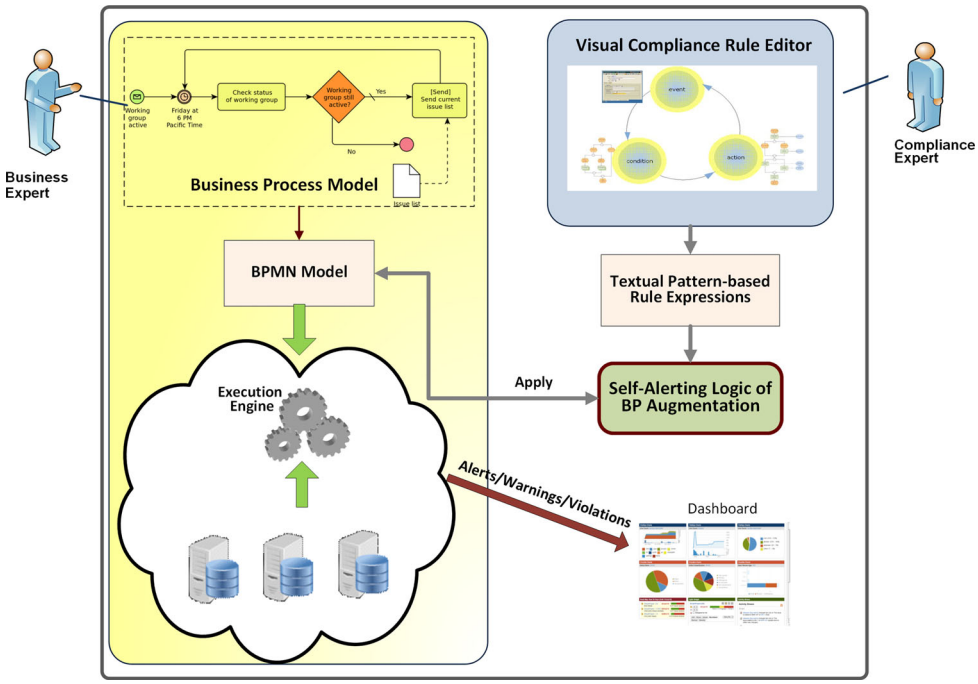


Fig. 2 Framework overview

on the input rule(s) in order to make the process model alert for/prevent violations to the rules at runtime. This component is the core contribution of this article and the details of how it works are discussed in the following subsections.

3.2 Timed order patterns

In general, order patterns, cf. Fig. 1, are concerned with the execution sequence of activities within a process instance. Moreover, this execution ordering might be required to occur within a time window, *time span* property, when the property *isWithin* is set to *true* or that the time window has to elapse before the consequent of the rule may occur, when the *isWithin* is set to *false*. In practice, verifying that a process design complies with the execution order between a rule's antecedent and consequent can be accomplished by design-time compliance checking techniques such as [4–6,23]. In this case, a faithful execution of the process by a central execution engine shall guarantee compliance to the rule. However, if the rule is decorated with timing constraints, design-time-only does not provide a sufficient guarantee that enacted instances of the process model will obey that constraint. Thus, we need a runtime support to monitor and detect violations to such timing constraints on order patterns.

Timed Response Pattern

As shown in Fig. 1, the time constraint imposed on the occurrence of the rule's consequent might occur either:

– *Before* the time span elapses from the time the rule's antecedent occurs or,
– *After* the time span elapses from the time the rule's antecedent occurs.

For the former case, an example rule, based on Definition 1, could be $Response(antecedent = A, consequent = B, condition = T, scopeStart = ProcessStart, scopeEnd = ProcessEnd, multiplicity = N/A, WA =, timeSpan = 5days, isWithin = true, alertTimeSpan = 4days, violationAction, predictiveAction)$

To enable execution awareness of possible violations, we basically do three steps

– Insert a throwing signal event right after the rule's antecedent task signaling antecedent task completion,
– Insert a throwing signal event right after the rule's consequent task signaling consequent task completion,
– Insert a non-interrupting event subprocess that is triggered by the antecedent's signal event and that awaits for the consequent signal event or a timer to occur.

A *signal* event [48] broadcasts a predefined event that is of a business value. The throwing signal event generates and broadcasts an instance of that event whereas the catching event *listens* to the occurrence of those event instances. The dissemination of the event instance is via throwing the signal event that crosses over the process instance boundary and even cross process definition. That is, any catching signal event that listens to that specific event will get notified by the underlying event delivery infrastructure. The non-interrupting event subprocess is actually triggered by a catching signal event for the rule's antecedent completion

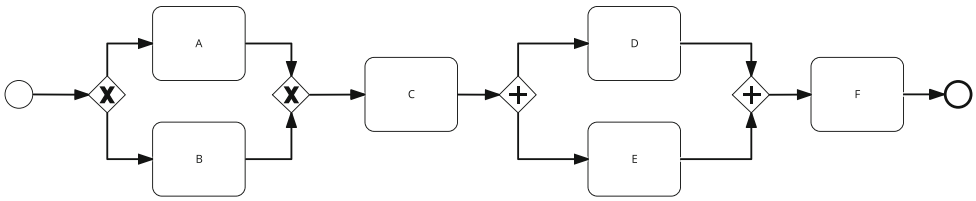


Fig. 3 An example process model

event. Once an event is caught, the process instance, case, id stored within the incoming event is compared to the case id of the catching instance. The comparison is made to avoid catching an irrelevant event from another case. If they are different, then the sub-process exits without further action. On the other hand, if they belong to the process instance, the subprocess proceeds and awaits, via an event-based gateway, for one of two events to occur, namely a signal event indicating the completion of the consequent task or a timer event indicating the expiry of the time span of the rule. The violation detection depends also on the nature of the time span, i.e., the $isWithin$ property of the rule. If it is set to $true$, then a violation occurs if the time span expires and the completion event of the rule's consequent is not observed. On the other hand, if $isWithin = false$, a violation occurs if the completion event is observed *before* the time span expires.

For the former case of $isWithin = false$, if the signal indicating the completion of the consequent task is caught first and it is originating from the same process instance, then the event subprocess completes and no extra action is needed. This is because the consequent task was completed within the specified time window and the compliance rule is satisfied. On the other hand, if the timer event expires first; this means that the time span allowed for the consequent task to complete its action has elapsed before observing that the consequent task has completed. In this case, a task with the logic to respond to the violation is invoked. Note that the subprocess initiating the catch event is *non-interrupting*. This means that the flow of the main process can continue normally without interruption. The role of the *timer* event is to make the subprocess idle until the rule's time span expires. As a result, the timer condition expression is `GetCompletionTimeOfTask(Antecedent, this)` + $Rule.timeSpan$. The first operand of that expression is an invocation of the runtime API to get the completion time of the antecedent task within $this$ process instance, cf. Sect. 2.2. The second operand is the rule's $timeSpan$ property. The result is a point in time, usually a `DateTime` value, until which the sub process will be waiting. The *violation action* task can also be replaced with a subprocess that contains a more sophisticated logic to respond to the violation. If the response to the violation might result in terminating the current instance, a *terminate* end event can be employed. For the latter case $isWithin = true$, what needs to be changed is to move the violation action task from the timer event branch to the catching signal event branch of the consequent task.

To better illustrate how our approach works, consider the process model in Fig. 3. Also, consider the following compliance rules:

- $R1 = Response(antecedent = C, consequent = F, timeSpan = 5days, isWithin = true, alertTime Span = 4days)$,
- $R2 = Response(antecedent = A, consequent = C, timeSpan = 5days, isWithin = false)$

Obviously, the process is compliant with the *ordering* constraint of the different rules by design. For instance, with respect to $R1$ whenever task C is executed, task F will be executed afterwards within the same process instance. Similarly for rule $R2$, when task A is executed, task C will be executed in the same instance. This also applies for $R2$. To make the process aware of possible time-based violations, for $R1$ we need to: 1) insert a signal event right after task C, 2) insert another throwing signal event right after task F and 3) insert an event subprocess that will include the logic to decide if there was a violation to the time window constraint. The modified process is shown in Fig. 4. We also added explicitly a data object *Instance ID* that carries the specific process instance ID. This data object is written at the creation time of the process instance by the start event. The value stored in this data object is obtained by invoking the runtime API `GetProcessInstanceID()`, cf. Sect. 2.2. Then this object is made as the input for the two throwing signal events. Rules $R1$ has also its $alertTimeSpan$ set. To address this part of the rule, we need to add an extra subprocess where the expression of the timer event is changed to refer to the rule *alert time span* property. Also, the violation action task needs to be replaced with the respective action to be taken to alert for the *possible* violation upstream. In that case, the actual violation can be avoided if the performer of task F is informed to speedup the processing of the task before the time span expires.

Figure 5 shows the changes made to the original model from Fig. 3 to alert for violations of $R2$. For $R2$, $isWithin = false$. That is the consequent is allowed to occur only *after* the time span from the antecedent occurrence has elapsed, cf. Definition 1. To monitor violations for such rules, the violation action task is put on the signal event branch.

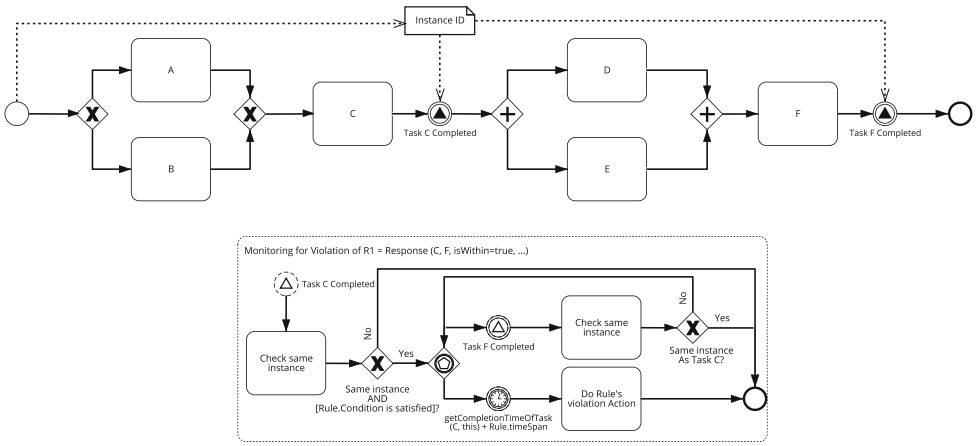


Fig. 4 The process from Fig. 3 after making changes to alert for violations of $R1$

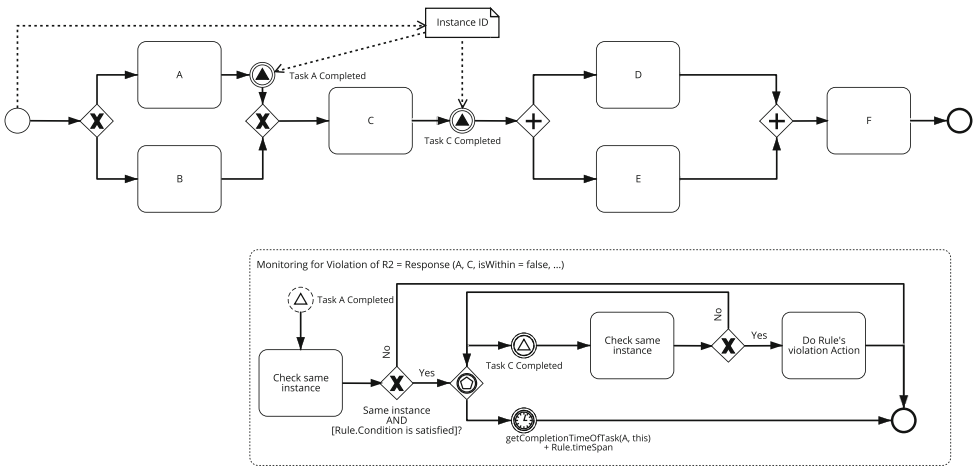


Fig. 5 The process from Fig. 3 after making changes to alert for violations of $R2$

As shown in Figs. 4 and 5, signal events are used within more than one event subprocess for different purposes. For instance, when the event $C\ Completed$ is thrown right after the task C completion, it is caught by two different catching events. One is a start non-interrupting event used to initiate the violation handling event subprocesses for rules $R1$. The other one is an intermediate catching event in the subprocess monitoring violations for rule $R2$. Also, in case that the rule's antecedent was part of a loop, the event sub-process will get instantiated as many times as the antecedent task executes as per the specification of BPMN 2.0 [48].

Timed Precedence Pattern

indicates that whenever the rule's antecedent occurs, the rule's consequent must have occurred in the instance execution history either *before* or *within* the specified timespan. Consider the following rules which are imposed against the process model in Fig. 3.

- $R3 = Precedence(antecedent = F, consequent = D, timeSpan = 1days, isWithin = true)$
- $R4 = Precedence(antecedent = F, consequent = E, timeSpan = 1days, isWithin = false)$

The process is compliant with ordering constraints imposed by the *precedence* pattern. That is, whenever task F is executed task D was executed before, $R3$, and also task E was executed, $R4$. To enforce the monitoring for the timing part, we need to 1) insert a throwing signal event after the rule's antecedent task in the process 2) add an event subprocess that is initiated by the corresponding catch signal event that includes the monitoring logic. By enforcing the monitoring of $R3$ on the process in Fig. 3 the modified process is shown in Fig. 6. The event subprocess is triggered by the completion of the antecedent task, task F for $R3$. Next, a check is made to be sure that the event was thrown from the same process instance. In that case, runtime API `GetCompletionTimeofTask(Task, Instance)`, cf. Sect. 2.2, is invoked for both the antecedent and the consequent tasks of the rule, C and F for $R3$. Then a check is made to see if the completion time of the consequent obeys the constraint enforced by the rule. That is, we check if the completion time of the consequent was within the time window ending with the completion time of the

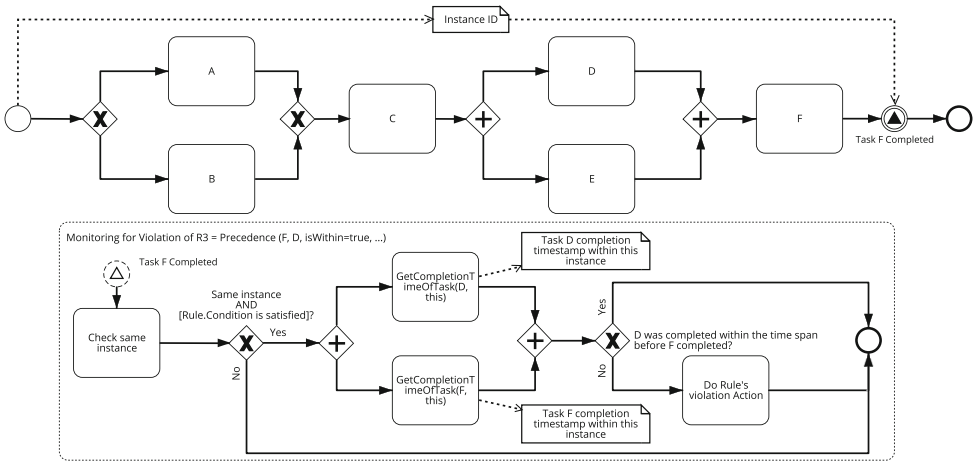


Fig. 6 The process from Fig. 3 after making changes to alert for violations of $R3$

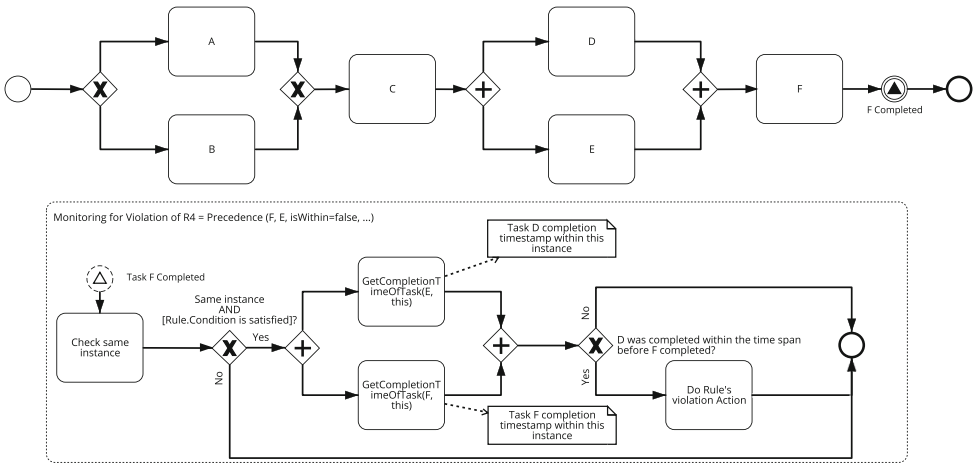


Fig. 7 The process from Fig. 3 after making changes to alert for violations of $R4$

antecedent and going backward with the amount specified in the `TimeSpan` property of the rule. If this is *not* the case and the rule's `isWithin` was set to $true$, this means that there is a violation and the rule's *violation action* has to take place. Otherwise, the event subprocess completes silently.

Figure 7 shows the changes made to the process from Fig. 3 to enable the monitoring for violations of $R4$. Note that for $R4$ `isWithin` is set to $false$. That is, in order for the process to be compliant, the consequent must have had occurred before the time window ending with the antecedent completion time and lasting backward for the rule's `TimeSpan`. So, if it happens that the completion time of the consequent was within the time window then there is a violation.

3.3 Resource patterns

Employed resources involves mainly the task allocations, access control and authorization constraints, and constitutes one of the important structural facets of BP compliance. As shown in Fig. 1, 'Resource Patterns' class is one of the three sub-classes of the 'Compliance Patterns' super-class. Resource patterns typically involve some basic business process concepts, in particular: role, user (actor), and task (or BP activity). We assume that tasks are assigned to roles and users perform the tasks through the roles they are assigned to. As shown in Fig. 1, we are considering three typical resource patterns: 'PerformedBy', 'SegregationOfDuties' and 'BindingOfDuties', which is described as follows:

- t PerformedBy R: This means that No other role than R is allowed to perform activity t
- t_1 SegregatedFrom t_2: Activities t_1 and t_2 must be performed by different roles and users
- t_1 BoundedWith t_2: Activities t_1 and t_2 must be performed by the same user

To be able to verify resource allocation and authorization constraints, it is necessary to formally represent some important business process elements along with their relationships, i.e., *Roles*, *Users* (or Actors) and *Activities* (or tasks). We are following the approach proposed in [62] for this purpose. More precisely, we assume the existence of three sets, i.e., $Users$, $Roles$ and $Activities$. The set

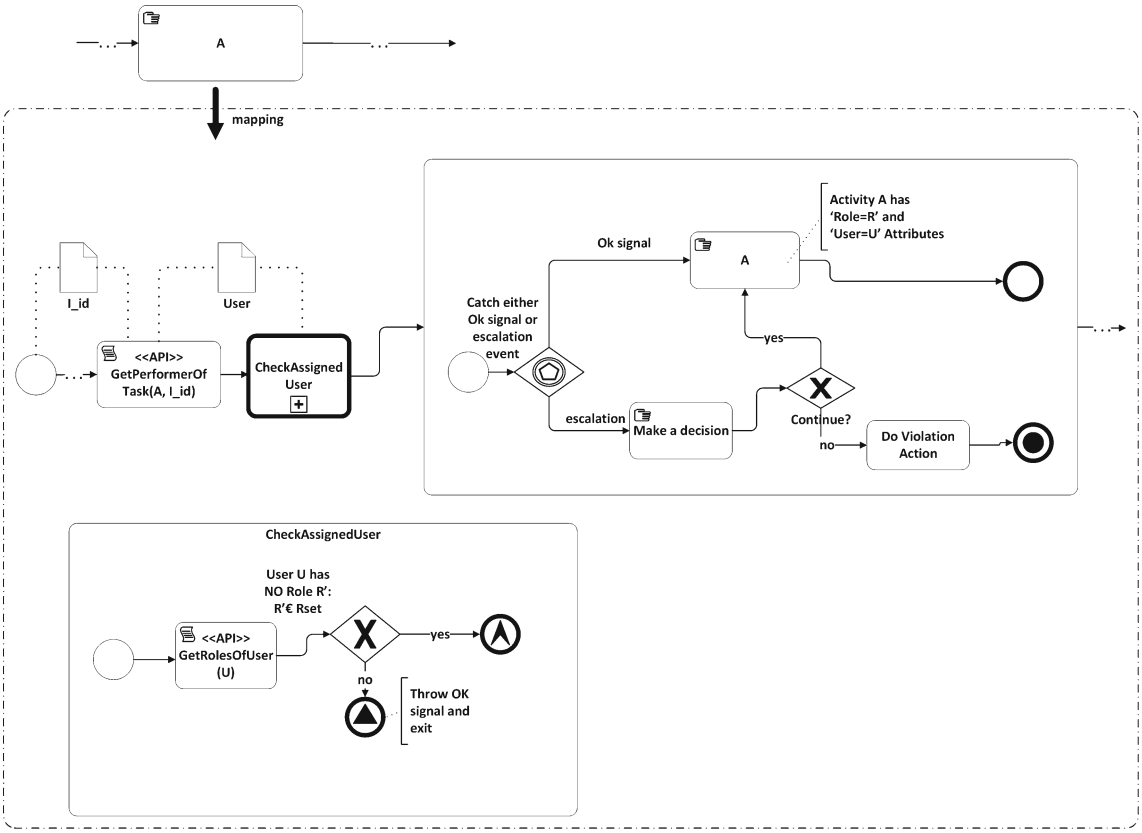


Fig. 8 Re-structuring BPMN model to detect, prevent and alert for the 'PerformedBy' pattern violation

($Users \times Roles \times Activities$) represents the allocation of activities to particular users assuming specific roles. The set $Roles$ is a partial order set forming a role hierarchy. This means that a role inherits the privileges assigned to its related lower-level roles in the role hierarchy. Similarly, the set ($Roles \times Users$) represents the M:N relationship between $Roles$ and $Users$.

Compliance requirements on employed resources typically require run-time information to be verified. That is, many of such requirements can only be partially verified for compliance at design-time. We assume that the binding of roles to specific tasks is known during design-time and has been statically verified using approaches such as in [22,61]. Therefore, the main focus here is to augment the respective BPMN models to detect and prevent employed resources violations on the $Actors$ level. In the following, the automated structural augmentation of BPMN models is discussed to enable corresponding running instances to detect, alert and prevent the violations of *PerformedBy*, *SegregatedFrom* and *BoundedWith* resource patterns.

PerformedBy Pattern

Figure 8 shows the structural augmentation of a standard BPMN model to enable the detection, prevention and alerting of this constraint violation, pro-actively. The mapping of the BPMN model takes advantage of the *Call Activity* construct of BPMN v2.0 [48], and other BPMN v2.0 advanced event types. A *Call Activity* is a reference to a globally defined (Sub-)Process. The construct represents a 'wrapper' for the call of the referenced process while executing the underlying process, the control flow is handed over to the called process. The call of a globally defined Process through Call Activity resembles the execution semantics of a conventional (sub-) process. The notation of a call activity is a rounded rectangle like a conventional activity but with a thick outline. The call activity may also be annotated with the subprocess marker ('+' sign), e.g., 'CheckAssignedUser' call activity in Fig. 8. The mapping also used two types of events in BPMN v2.0:

- *Escalation event*: escalating to a higher level of responsibility. An escalation event is represented by a thick upper arrow inside conventional start, intermediate or end events.
- *Signal event*: signalling across different processes. A signal thrown can be caught multiple times. A signal event is represented as a triangle inside conventional start, intermediate or end events. A filled triangle is used with end events, while a hollow triangle is used with start and intermediate events.

Given this background, assume that in Fig. 8 the rule $R_1 = A\ PerformedBy\ R$ is imposed on a BPMN model (upper part of Fig. 8). The goal during runtime is to ensure that the User U assigned to activity A has the role R. The structural mapping of the BPMN model involves the following steps:

1. Define a new global process called 'CheckAssignedUser' (lower part of Fig. 8). 'CheckAssignedUser' process starts by calling the `GetRolesOfUser (User)` API (described in Sect. 2.2), which returns a set $UR = \{UR_1, UR_2, \ldots, UR_n\}$ representing the roles assigned to User U. Then an exclusive gateway checks whether $UR \cap \{R\} = \phi$, which means that Role R is not assigned to User U, and represents a violation. If this condition evaluates to true then an end escalation signal is thrown (to escalate the potential violation before its occurrence to a higher role). If no potential violation is detected an 'OK' end signal is thrown
2. For every occurrence of Activity A, insert an activity that calls `GetPerformerOfTask(A, I_id)` API as described in Sect. 2.2, such that `I_id` represents the instance ID and the API returns the assigned user to Activity A.
3. Then insert a call activity 'CheckAssignedUser' just before Activity A and after the `GetPerformerOfTask (A, I_id)` API call.
4. Add Activity A in a sub-process, which starts with exclusive event-based gateway to catch either the escalation event (indicating that a violation is about to occur), or an 'OK' signal thrown from the 'CheckAssignedUser' global process.
5. If an 'OK' signal is caught, activity A is executed and the flow proceeds normally
6. If an escalation event is caught, a defined high-level role makes an informed decision whether to override the PerformedBy rule. Based on this decision, either a defined violation recovery action is performed, and the instance terminates, or activity A is executed and the flow proceeds normally

SegregatedFrom Pattern

Segregartion of duties (SoD) is a well-recognized resource allocation constraints in the regulatory domain to minimize the possibilities of fruad [28]. Following the same rationale of the mapping of the PerfomedBy resource Pattern discussed above, we assume that the SoD constraint has been checked on the Role level during design-time, which means that it has been ensured that the two activities involved in the SoD rule (e.g., A and B) are performed by different roles. Therefore, the objective of our mapping is to ensure that: (i) Activities A and B are performed by different Users, and then (ii) if the assigned users to activities A and B are different, ensure that they do not hold the same role. SoD is a binary operator that does not impose an order constraint on its operands. For simplicity, the mapping discussed here considers if activity A occurs before activity B in the BPMN model, however, our mapping and implementation discussed in Sect. 4.1 also considers the case of B precedes A. If either (both) A or (and) B is (are) absent, then no mapping is needed.

Figure 9 shows the augmentation of a standard BPMN model to detect, prevent and self-alert for potential violation before its occurrence. The mapping also takes advantage of 'Call Activity', 'Signal' and 'Escalation' events of BPMN v2.0 as described in the PerformedBy pattern mapping (Sect. 3.3)

Assume that in Fig. 9 the rule $R_1 = A\ SegregatedFromB$ is imposed on a BPMN model (upper part of Fig. 9). The structural mapping of the BPMN model involves the following steps:

1. Define a new global process called 'CheckSegregationOfDuties' (lower part of Fig. 9). 'CheckSegregationOfDuties' process starts by checking whether users $U1$ and $U2$ sent from the call activity are different. If this condition is evaluated to false, an escalation signal is thrown and the 'CheckSegregationOfDuties' process terminates. If the condition is satisfied, two API calls are invoked concurrently to the API `GetRolesOfUser (User)` API (described in Sect. 2.2) to return the roles assigned to User $U1$ and User $U2$, respectively. That's `GetRolesOfUser (U1` returns a set $UR1 = \{UR1_1, UR1_2, \ldots, UR1_n\}$ representing the roles assigned to User $U1$. And `GetRolesOf User (U2` returns a set $UR2 = \{UR2_1, UR2_2, \ldots, UR2_n\}$ representing the roles assigned to User $U2$.
2. Then an exclusive gateway checks whether $UR1 \cap UR2 = \phi$, which means that User $U1$ and User $U2$ do not play the same role and that the SegregatedFrom constraint is satisfied. If this condition is satisfied, an OK signal is thrown and the 'CheckSegregationOfDuties' terminates. If this condition evaluates to false then an end escalation signal is thrown (to escalate the potential violation before its occurrence to a higher role).
3. After every occurrence of Activity A, add an activity that calls `GetPerformerOfTask(A, I_id)` API as described in Sect. 2.2, such that `I_id` represents the instance ID and the API returns the assigned user `U1` to Activity A.
4. Then, before activity B, add an activity that calls `GetPerformerOfTask(B, I_id)` API as described in Sect. 2.2, such that `I_id` represents the instance ID and the API returns the assigned user `U2` to Activity B.
5. Then add a call activity 'CheckSegregationOfDuties' just before Activity B and after the `GetPerformerOf Task(B, I_id)` API call.

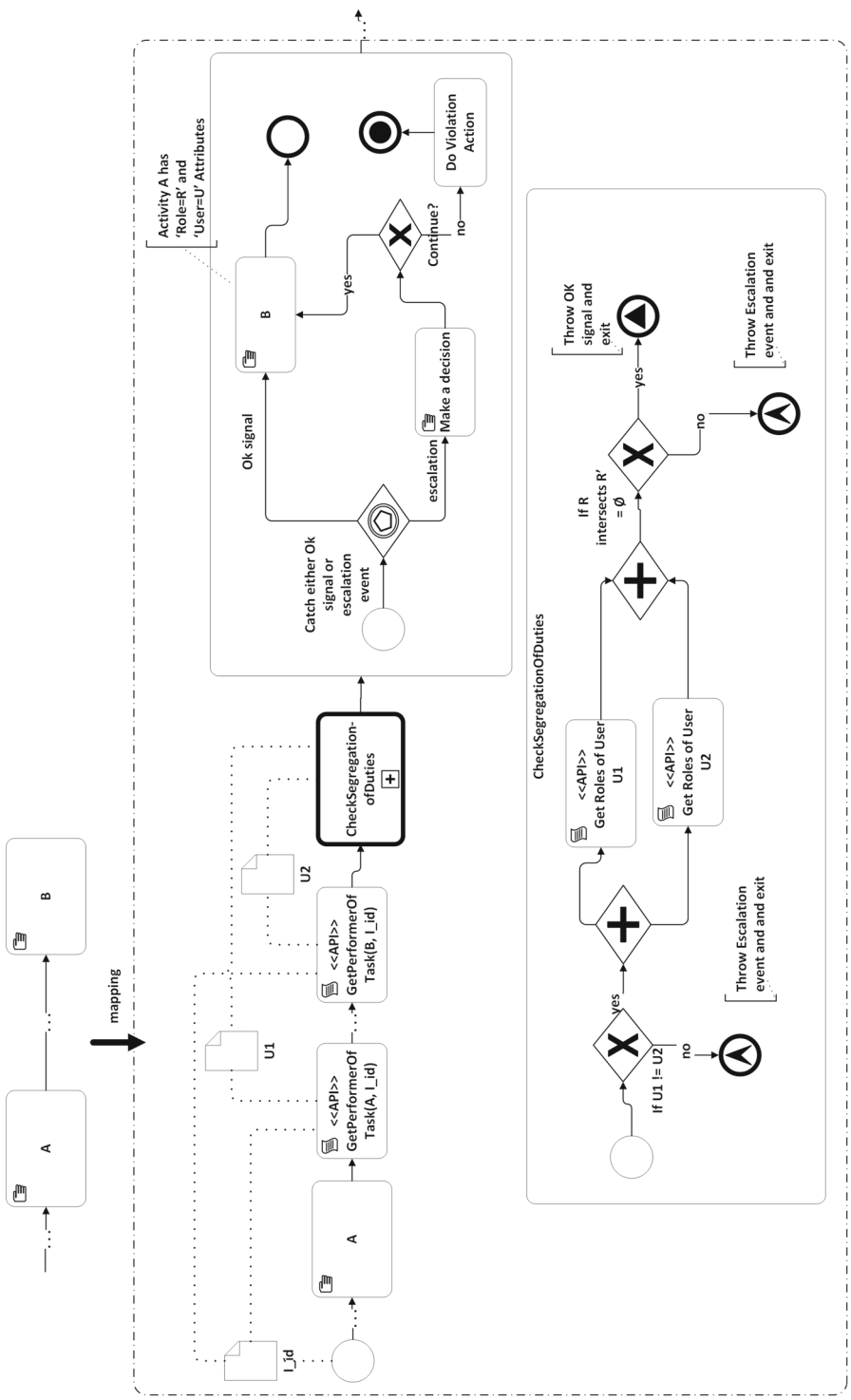


Fig. 9 Re-structuring BPMN model to detect, prevent and alert for the 'SegregatedFrom' pattern violation

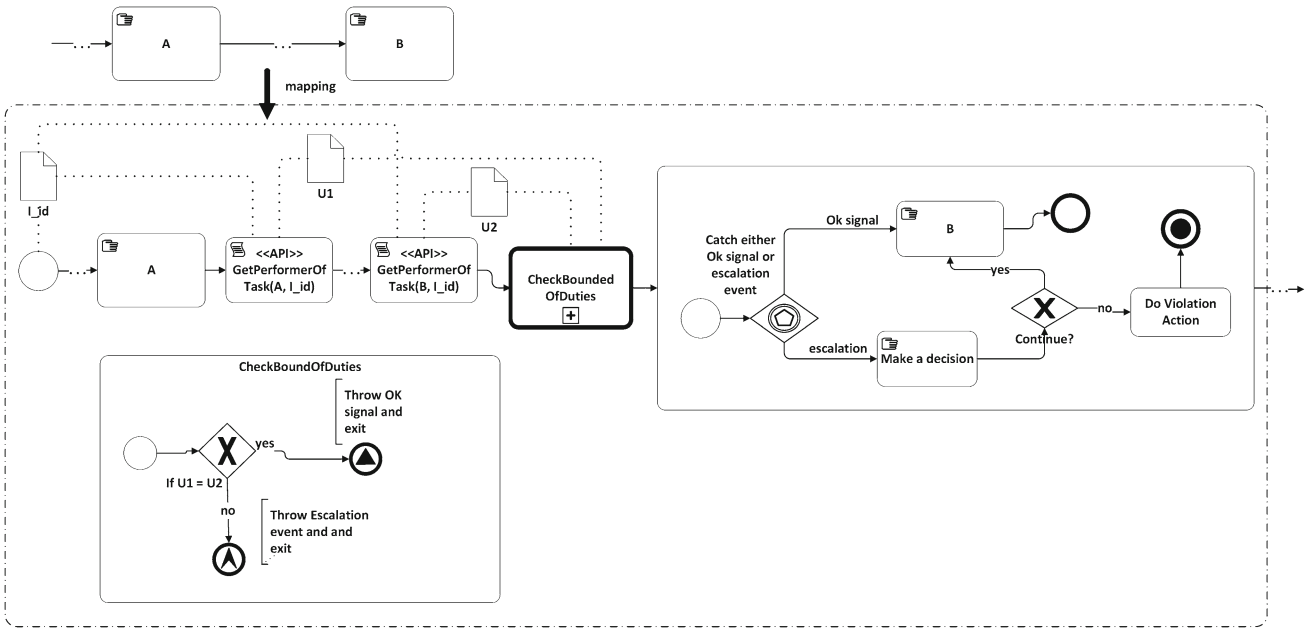


Fig. 10 Re-structuring BPMN model to detect, prevent and alert for the 'BoundedWith' pattern violation

6. Add Activity *B* in a sub-process, which starts with exclusive event-based gateway to catch either the escalation event (indicating that a violation is about to occur), or an 'OK' signal thrown from the 'CheckSegregationOfDuties' global process.
7. If an 'OK' signal is caught, activity 'B' is executed and the flow proceeds normally.
8. If an escalation event is caught, a high-level role makes an informed decision whether to override the SegregatedFrom rule. Based on this decision, either a defined violation recovery action is performed, and the instance terminates, or Activity *B* is performed and then the flow proceeds normally.

BoundedWith

BoundedWith compliance pattern only imposes constraint on the users that are allowed to perform two binary activities, and therefore it can only be checked during runtime. BoundedWith is a binary operator that does not impose any order constraint on its operands. For simplicity, the mapping discussed in this Section considers the case where activity *A* occurs before activity *B* in the BPMN model, however, our mapping and implementation, cf. Sect. 4.1 also considers the case of *B* precedes *A*. If either (or both) *A* or *B* is absent, then no mapping is needed. Figure 10 shows the augmentation of a standard BPMN model to detect, prevent and self-alert for potential violation before its occurrence. The mapping also take advantage of 'Call Activity', 'Signal' and 'Escalation' events of BPMN v2.0 as described in the PerformedBy pattern (Sect. 3.3). The structural mapping of the BPMN model involves the following steps:

1. Define a new global Process called 'CheckBoundedOfDuties' (lower part of Fig. 10). 'CheckBoundedOfDuties' process starts by checking whether users $U1$ and $U2$ sent from the call activity are the same. If this condition is false, an escalation signal is thrown and the 'CheckBoundedOfDuties' process terminates. If the condition is satisfied an 'OK' signal is thrown and the 'CheckBoundedOfDuties' terminates.
2. After every occurrence of Activity *A*, add an activity that calls `GetPerformerOfTask(A, I_id)` API as described in Sect. 2.2, such that `I_id` represents the instance ID and the API returns the assigned user `U1` to Activity *A*.
3. Then, before activity *B*, add an activity that calls `GetPerformerOfTask(B, I_id)` API as described in Sect. 2.2, such that `I_id` represents the instance ID and the API returns the assigned user `U2` to Activity *B*.
4. Then add a call activity 'CheckBoundedOfDuties' just before Activity *B* and after the `GetPerformerOfTask (B, I_id)` API call.
5. Add Activity *B* in a sub-process, which starts with exclusive event-based gateway to catch either the escalation event (indicating that a violation is about to occur), or an 'OK' signal thrown from the 'CheckBoundedOfDuties' global process.
6. If an 'OK' signal is caught, activity 'B' is executed and the flow proceeds normally
7. If an escalation event is caught, a high-level role makes an informed decision whether to override the SegregatedFrom rule. Based on this decision, either a defined violation recovery action is performed, and the instance

terminates or Activity B is performed and then the flow proceeds normally.

3.4 Applicability to BPEL

In this section, we discuss the applicability of our approach to BPEL 2.0 [46] as it is a standard execution language for business processes. With the BPEL4People [47] and Human Task [45] extensions, BPEL is able to invoke activities that are accomplished by humans. Thus, resource-based patterns need to be monitored over BPEL4People processes. Also are timed ordering patterns. The presented approach in Sect. 3 depends on constructs that are found in BPMN 2.0 [48]. Namely it depends on:

- Signal events, throwing and catching,
- Escalation events, throwing and catching,
- Non-interrupting event subprocesses,
- Call activities

Looking at the family of BPEL standards [45–47], we can notice that the signal and escalation events have no counterparts in BPEL at the standard specification level. However, some commercial tools that support BPEL have made their own extensions to support such constructs. Yet, BPEL 2.0 provides an `Extension Activity` which is a way to provide user-defined types of activities. With this in hand, two extensions to cover the missing signal and escalation events can be defined. Event handlers of BPEL can be used as the equivalent for the event subprocesses in BPMN. By default, event handlers in BPEL are non-interrupting. So, if the rule just requires sending out notifications nothing more is needed. On the other hand, if the rule requires a suspension or a termination of the instance, the event handler can have an `Exit` activity invoked in its event handling logic. Call activities in BPMN can be defined as standalone BPEL processes that are invoked as needed from within the BPEL process to be monitored. This solution shall provide a self-contained BPEL process that includes both the business logic and the monitoring logic in the same way as was shown in Sect. 3 for BPMN. However, BPEL has a limitation when it comes to implementing the event-based exclusive gateway. The counterpart construct in BPEL is the `pick` activity. The limitation is that `pick` can handle either timer or message events only. Currently, there is no support for other types of events not even via an extension. So, a solution for this limitation is to implement the *signal* extension activity in a way that triggers a message to be sent to the same process instance. Thus, the correlation is based on the process instance identifier.

For the sake of clarity, we introduce the example rule $R1$ above and its monitoring logic implemented in a snippet of a BPEL process. First, we use BPEL `extensionActivity` to define a signal event as shown in Listing 1.

```
<bpel:extensionActivity>
<signal .../>
</bpel:extensionActivity>
```

Listing 1 A BPEL extension to define the signal activity

Listing 2 is a pseudo BPEL code to show how the signal activity is used to implement the changes from Fig. 4.

```
<process>
<eventHandlers>
<onEvent name="Task␣C␣Completed">
<sequence>
<invoke name="Check␣same␣instance">
<if>
<condition>$Result = 'Yes'␣and␣$Orider.Amount␣>
5000␣and␣$Risk.threshold␣<␣2</condition>
<repeatUntil␣>
<pick>
<onMessage␣name="Task␣F␣Completed">
<scope>
<invoke␣name="Check␣same␣instance">
<if>
<condition>$SameInstanceAsAntecedentTask='Yes'
</condition>
<assign>
<copy><from>'No'</from>
<to>$MoreRound</to></copy>
</assign>
</if>
</scope>
</onMessage>
<onAlarm>
<until>$getCompletionTimeOfTask(C)+$Rule.timeSpan
</until>
<scope>
<invoke␣name="Rule␣violation␣action"/>
<assign>
<copy><from>'No'</from>
<to>$MoreRound</to></copy>
</assign>
</scope>
</onAlarm>
</pick>
<condition>$MoreRound='Yes'</condition>
</repeatUntil>
<else>
<empty/>
</else>
</if>
</sequence>
</onEvent>
</eventHandlers>
```

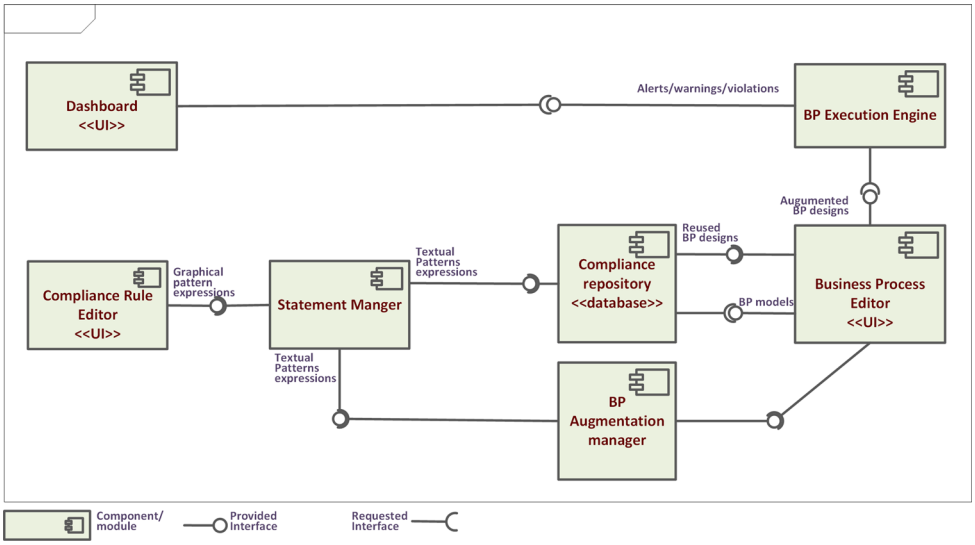


Fig. 11 Self-alerting BP compliance approach implementation architecture represented as a UML diagram

```
<sequence>
<if>
<condition/>
<invoke name="A"/>
<else>
</else>
<invoke name="B">
</if>
<invoke name="C">
<signal name"Task C Completed">
<flow>
<invoke name="D"/>
<invoke name="E"/>
</flow>
<invoke name="F">
<signal name"Task F Completed">
</sequence>
</process>
```

Listing 2 A BPEL process with monitoring logic for $R1$

4 Evaluation

In this section, we discuss the implementation of the proposed approach as well as evaluating the implemented approach against a case study from the financial sector regarding anti money laundering (AML).

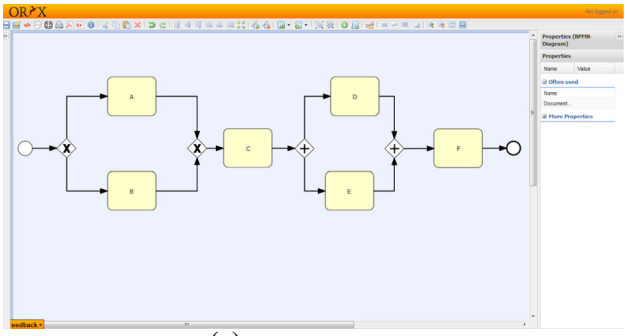
4.1 Implementation

The pattern-based process rewriting steps discussed in Sect. 3 have been implemented on top of the process model editor Oryx [16]. Oryx is an open source extensible model editor and repository. Originally, it supports BPMN 2.0 constructs. We extended Oryx in two ways. First, we defined an editor to visually compose compliance rules based on the patterns presented in Sect. 2.1. The second extension was by means of defining a plugin for the native BPMN 2.0 editor. Through the plugin, the user can choose an arbitrary number of compliance rules, previously defined via the rule editor. Then the plugin applies the changes discussed earlier and the modified model, that is now equipped with monitoring logic, is saved to a new model and opened for review by the user.

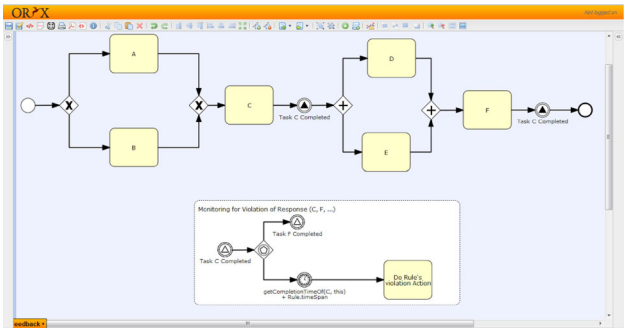
Figure 11 illustrates the implementation architecture of the self-alerting business process monitoring approach proposed in this paper. The implementation architecture consists of the following main components:

- *Compliance repository*: This is a central repository that stores and maintains business process and compliance-related specifics, where business and compliance concepts are semantically aligned, as well as augmented BP model for self-monitoring of timed and employed resources constraints.
- *Compliance rule editor*: This editor provides a graphical representation of the compliance patterns (as discussed in Sect. 2). The visual editor component has been implemented as a plugin on top of the Oryx editor.[4] Figure 12 illustrates screenshots for our compliance editor environment.
- *Statement manger*: This module is responsible for automatically compiling the visually modelled compliance rule into textual patterns expressions (e.g., Eq. 1 and Eq. 3 in Sect. 4.2 below)
- *Business process editor*: Provides the end users with a user-friendly modelling environment where the users can model their business process using the standard BPMN

[4] https://code.google.com/p/oryx-editor/.



(a) Original Process Model



(b) Adopted Process Model with Embedded Compliance Logic

Fig. 12 System screenshots of the compliance modeling environment

2.0 language. We employ the open source BPM platform Activiti[5] as a realization of this component where the user can model and enact business processes.

– *BP augmentation manager*: the augmentation manger implements the BP augmentation logic for monitoring timed and employed resources constraints as discussed in Sect. 3. BP Augmentation Manager augments the BPMN model from the 'Business Process Editor' component to prevent/alert against the applicable timed and employed resources constraints from the 'Statement Manager'.
– *BP execution engine*: Through another plugin, an augmented BPMN process model with the self-monitoring logic can be deployed to the Activiti engine which is running in cloud-based environment.[6]
– *Monitoring dashboard*: The dashboard is a user-friendly interface that enables the end-user to monitor the compliance status of the applicable set of compliance rules. The dashborad also contains statistical and aggregated information in tabular format and graphs that can help the experts to get a profound insight into any subtle flaws and assists in taking informed decisions that may yield to the modification of the respective business process model.The monitoring dashboard has been implemented as a desktop application using Microsoft C# .Net technology. Figure 13 illustrates a screenshot for our dashboard environment.

4.2 Case study: money-laundering detection process

Anti-money laundering is a pressing concern to any organization operating in the financial industry, as it is tightly adjunct to terrorism and proliferation financing. Despite the fact that it is not possible to precisely quantify the amount of money laundered every year, in [50], it has been shown that billions of US dollars were laundered annually. As part of a previous work [21], we have built an end-to-end business process encoded in the BPMN v2.0 standard that captures money laundering detection and reporting of the AML practices. The BPMN model is established based on best practices recommended by the Financial Action Task Force (FATF) 40 [24].

Figure 14 presents the money laundering detection and reporting BPMN process encoded in BPMN v2.0. The process proceeds as follows: it starts by a customer initiating a money transfer. Once the order is received by the bank, and if the order amount is greater than a given threshold (interpreted as five thousands Euros in our BPMN model), an automated check is carried out to detect if the transaction is suspicious. If the automated module detects that the transaction is suspicious, an authorized personnel is required to double check the transaction manually by reviewing clearance records and all other available records, and, if necessary, contacting the customer for further information.

If the transaction is proved to be suspicious, the transaction is flagged as suspicious and then deferred, and a Suspicious Activity Report (MSB) is sent to FinCEN (The Financial Crimes Enforcement Network).[7] The customer will be notified in both cases on the status of the transaction, while retaining all supporting documents in case they are requested by FinCEN during its investigation.

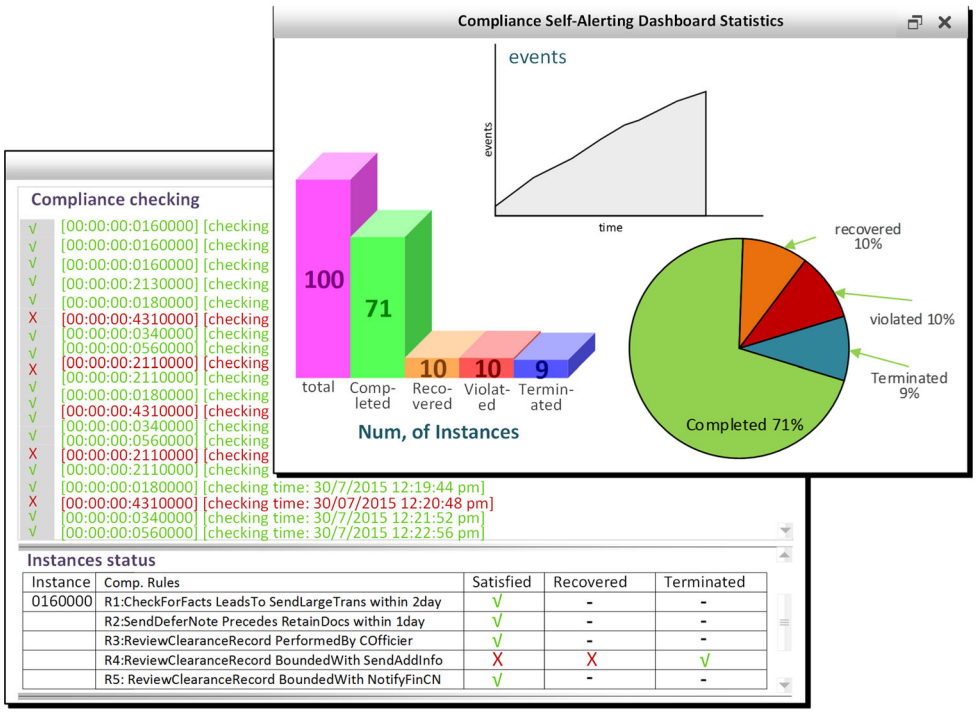
Table 1 presents a selection of the compliance requirements including risks, controls and sources applicable to the suspicious transaction reporting scenario. The first and second columns of the table allocate a unique reference and an organization-specific interpretation of the requirement, respectively. The third column lists the risks associated with these compliance constraints. Finally, the fourth column refers to the associated compliance sources.

In the rest of this section, we show how our approach, discussed in Sect. 3, is applied to the case study shown in Fig. 14.

[5] http://activiti.org/.

[6] It should be noted that our approach is agnostic towards the underlying business process execution engine and it can be adopted to any business process execution environment or SaaS platform.

[7] FinCEN: http://www.fincen.gov/.



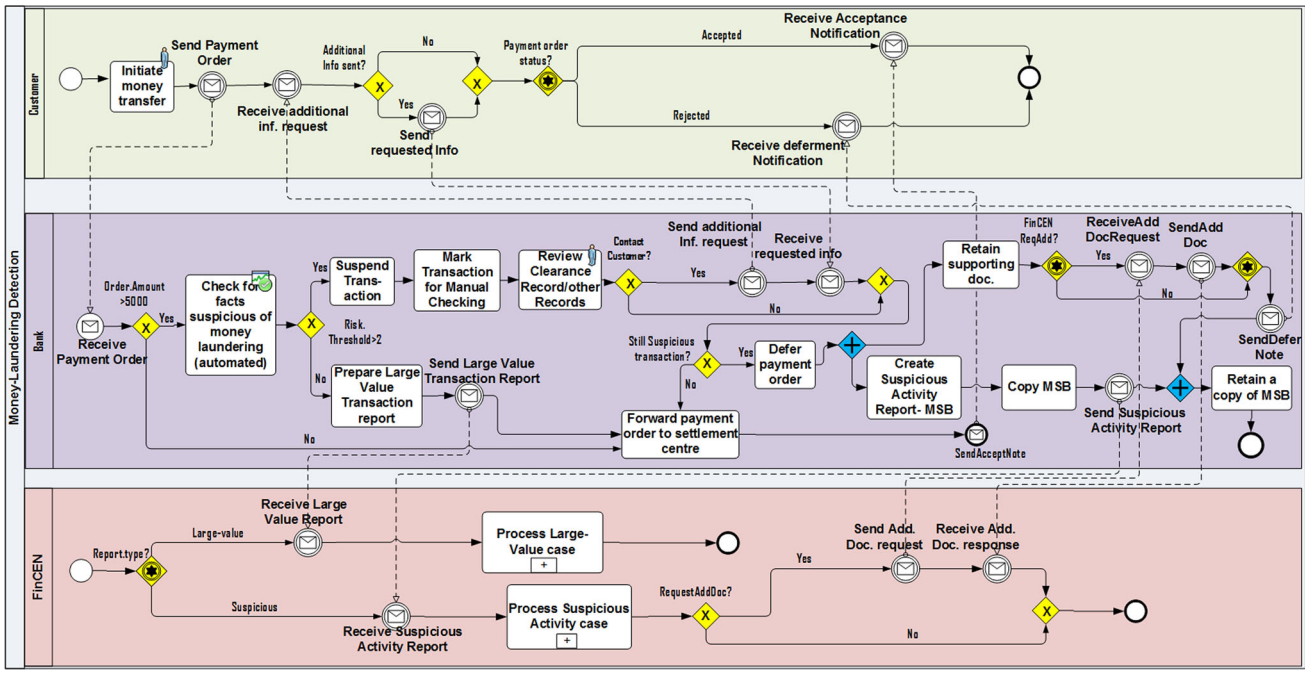


Fig. 13 Screenshot of system dashboard

Fig. 14 Money laundering detection and reporting BPMN process

Compliance Requirement $R1$ from Table 1 can be represented by a *timed response* pattern as shown in Formula 1.

$$\begin{aligned} R1\colon\ & Response(antecedent = Check\ for\ facts\ suspicious\ldots,\\ & consequent = Send\ LArge\ Value\ Transaction\ Report,\\ & condition = Order.Amount > 5000\ AND\ Risk.\\ & Threshold < 2,\ldots,\\ & timeSpan = 2days,\ldots,\\ & violation\ action = Notify\ for\ delayed\ reporting) \end{aligned} \quad (1)$$

Referring to the AML process in Fig. 14, we can notice that the ordering constraint for $R1$ above is satisfied. That is, task *Check for facts suspicious of money laundering* is executed and the risk threshold is greater than two and as the

Table 1 Excerpt of the Comp. Req. relevant to the AML scenario

ID	Comp. Req.	Risk	Comp. source	Pattern
R1	It is obligatory that the financial institution reports any suspicious transaction that involves or aggregates funds of at least $5,000 no more than 2 days from the logging of the transfer request	Fraud/misuse -Financial loss -Anti-money laundering -Terrorism financing	US Patriot Act. 1022. 210 1022.320	Timed response
R2	It is obligatory that documents support a suspicious activity are retained before the defer notice to the client in no more than 1 day.	-Loss of customers -Loss of reputation	Internal policy	Timed precedence
R3	It is obligatory that a designated compliance officer to double check each suspicious transaction manually by reviewing clearance records and all other available records	-Legal penalty due to non-compliance with laws -Fraud/misuse -Financial loss	US Patriot Act. 1022.210 (d)(2)	Performed by role
R4	It is obligatory that the manual checking of suspicious transactions and contacting the customer for additional information are bounded to the compliance officer	-Fraud/misuse -Financial loss	US Patriot Act. 1022.210 (d)(2)	Bind of duty

amount of the transfer is greater than $5000 the transaction will be reported to FinCEN, via the the respective intermediate message event. However, what needs to be monitored is the timing part of the rule. Figure 15 shows a snippet of the Bank's business process after enforcing $R1$ as discussed in Sect. 3. In Fig. 15, we showed only the part that is affected by the process. Two throwing signal events have been inserted after the task *Check for facts of suspicious money laundering*, $R1$ antecedent, and after the message event *Send Large Value Transaction Report*, $R1$ consequent, respectively. The event sub process catches the event of the rule's antecedent and compares the process instance identifier and also the rule's condition, `Order.Amount > 5000 AND Risk.Threshold < 2`, is checked. If the check passes, the sub-process begins the actual monitoring logic by waiting for the first of the two events to occur, either the completion of *Send Large Value Transaction Report* or a timer event based on the timestamp at which the antecedent was completed and adding 2 days as specified in the time span property of $R1$. Note that having an intermediate message event instead of a task as the rule's consequent does not affect our approach as in both cases it is possible to obtain the completion timestamp.

$R2$ from Table 1 can be represented as an instantiation of the *timed precedence* pattern. The formal representation of $R2$ is shown in Formula 2

$$\begin{aligned} R2: \; & Precedence(antecedent = Send\ Defer\ Note, \\ & consequent = Retain\ supporting\ documents, \ldots \\ & timeSpan = 1days, isWithin = false \ldots, \\ & violation\ action = Notify\ for\ quick\ defer\ note) \end{aligned} \tag{2}$$

$R2$ antecedent is the message event *Send Defer Note* whereas the consequent is the task *Retain supporting documents*. It is obvious that the AML process from Fig. 14 is compliant with the *ordering* part of $R2$. That is, whenever a defer note is sent, supporting documents are retained before. However, we still need to make sure that the timing part of the rule is monitored. As the rule requires that the send report action to be taken after at least 1 day before sending the defer note, $isWithin = false$, we make changes to the process as in the template shown in Fig. 7. Again, we show only parts of the process from Fig. 14 that have been affected in Fig. 16. As can be seen in Fig. 16, the check in the left most XOR split is only

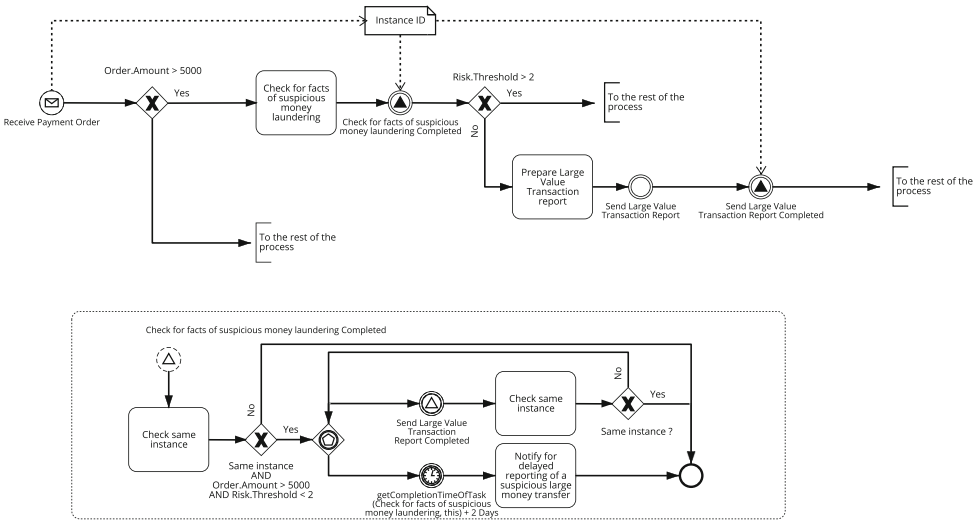


Fig. 15 Augmenting the AML detection process to detect the violation of $R1$

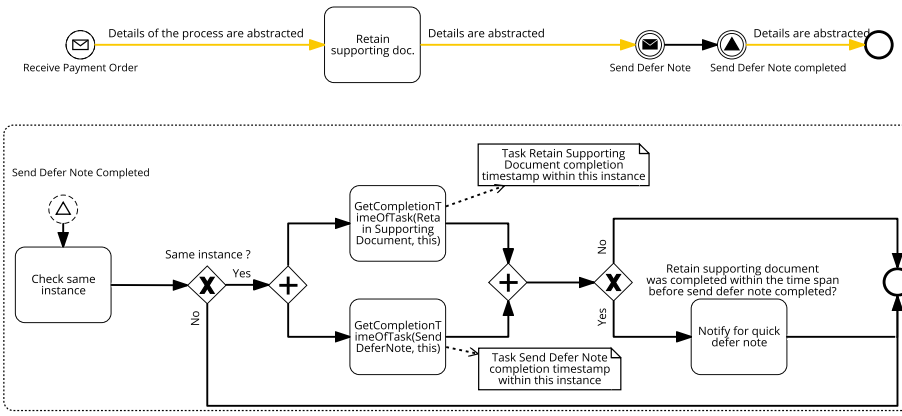


Fig. 16 Augmenting the AML detection process to detect the violation of $R2$

about checking having the same process instance for the event sub-process and the signal event. That is because the rule has no specific condition, i.e., the condition is equal to $true$. Next, both the timestamps of task *Retain supporting documents* and message event *Send Defer Note* are obtained from execution history and compared with respect to the the time span from the rule. If it is found that the message was sent earlier than specified, task *Notify for quick defer note* is invoked.

Compliance Requirement $R3$ in Table 1 can be represented by the *performed by* pattern as shown below. This can be easily represented using the *PerformedBy* pattern as:

$$R3: \; Performed\ by\ role(antecedent = Review\ Clearance\ role\ Record, \ldots, = Compliance\ Officier) \tag{3}$$

Figure 17 shows how the AML BPMN shown in Fig. 14 is automatically augmented following the above the approach discussed in Sect. 3 to enable the BPMN engine during runtime to detect, alert, prevent the violation of compliance requirement $R3$. To avoid a cluttered diagram, we only show in Fig. 17 the affected parts of the AML BPMN model in Fig. 14 within the scope of the 'Bank' lane. As shown in Fig. 17, Just before every occurrence of 'ReviewClearanceRecord' activity (which occurred only once in the AML BPMN model in Fig. 14), an activity that calls the `GetPerformerOfTask(A, I_id)`, where activity A is the operand of the 'PerformedBy' pattern; 'ReviewClearanceRecord' activity. This API call returns the user assigned to 'ReviewClearanceRecord' activity. This is followed by inserting the 'CheckAssignedUser' call activity as described in Sect. 3 is inserted just before the 'ReviewClearanceRecord' and after the 'GetPerformerOfTask' API call.

Then 'ReviewClearanceRecord' is added in a sub-process, which starts by an event-based gateway that catches either an escalation or 'OK' signal from the global 'CheckAssignedUser' process, indicating the escalation of a potential violation or the execution of the 'ReviewClearanceRecord' normally, respectively. If an escalation event is received a defined higher-level role will decide whether to resume the execution and override the 'PerformedBy' rule or to invoke a defined violation Action and terminate.

Compliance requirement $R4$ in Table 1 of the of the money laundering detection process introduced in Sect. 4.2 is an

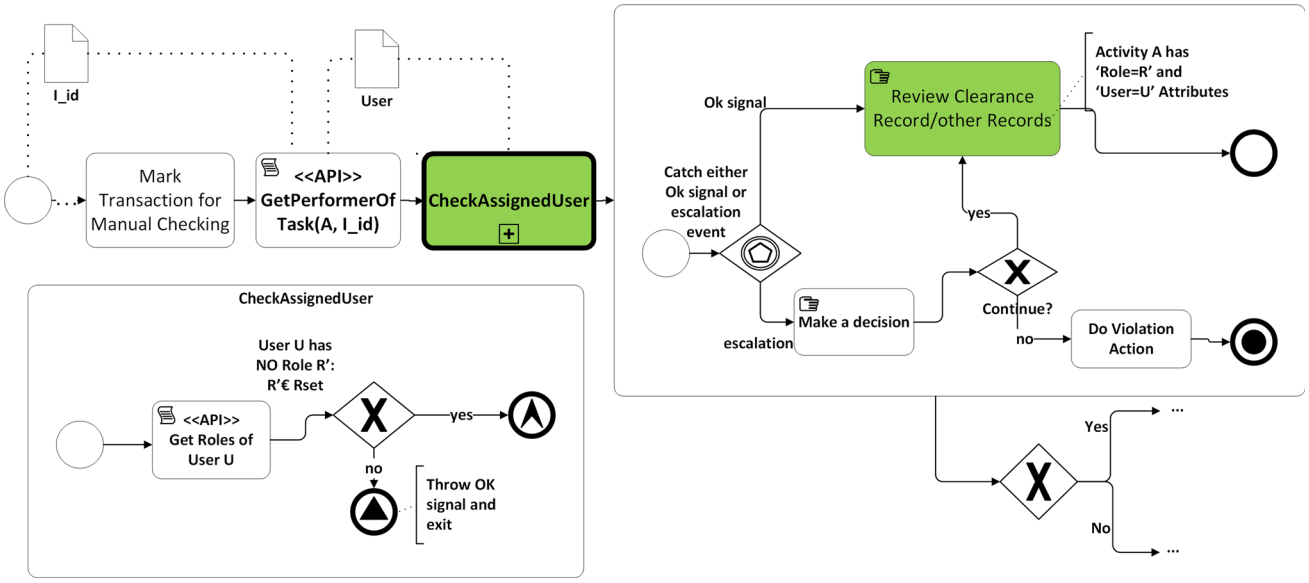


Fig. 17 Augmenting the AML detection process to detect, alert and prevent the violation of R3

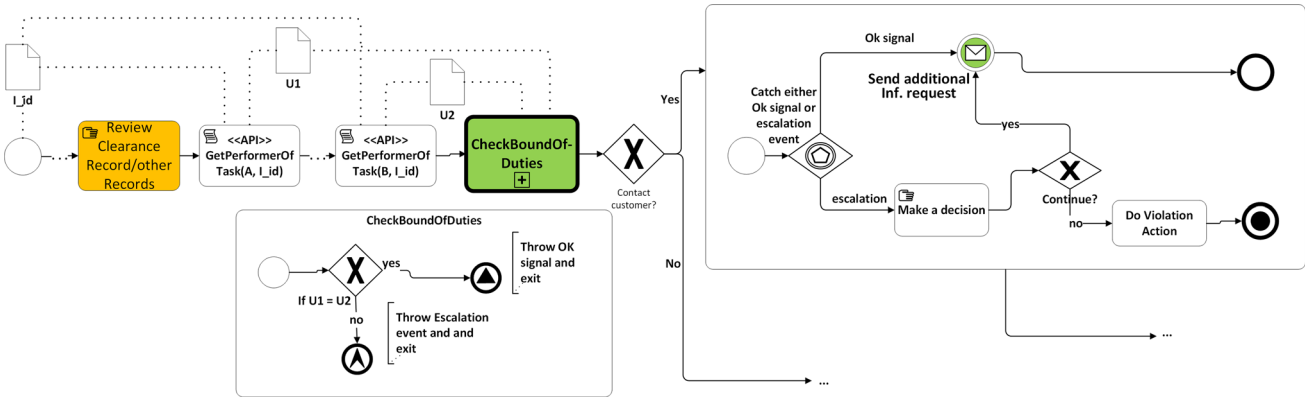


Fig. 18 Augmenting the AML detection process to detect, alert and prevent the violation of R4

example of *BoundedWith* compliance constraint. This can be represented using the *BoundedWith* pattern as:

$$R4: \; BindOfDuty(antecedent = Review\ Clearance\ Record, consequent = Send\ Add\ Inf\ Request, \ldots) \tag{4}$$

Similarly, Fig. 18 shows how the AML BPMN shown in Fig. 14 is automatically augmented following the proposed approach to enable the BPMN engine during runtime to detect, alert, prevent the violation of compliance requirement *R*4. Analogously, to avoid a cluttered diagram, we only show in Fig. 18 the affected parts of the AML BPMN model in Fig. 14 in the scope of the 'Bank' lane. As shown in Fig. 18, Just after the occurrence of the first operand of the BoundedWith pattern; 'ReviewClearanceRecord', an activity that calls the `GetPerformerOfTask(A, I_id)`, where activity *A* is the first operand of the 'BoundedWith' pattern; 'ReviewClearanceRecord'. This API call returns the user assigned to 'ReviewClearanceRecord' activity.

Then the execution flows until the second operand of the 'BoundedWith' pattern is reached; 'SendAddInfRequest'. Just before 'SendAddInfRequest', an activity that calls the `GetPerformerOfTask(B, I_id)` is inserted, where activity *B* is the second operand of the 'BoundedWith' pattern; 'SendAddInfRequest'. This API call returns the user assigned to 'SendAddInfRequest' activity. Then the 'CheckBoundedOfDuties' call activity as described in Sect. 3 is inserted just before the occurrence of the second operand of the 'BoundedWith' pattern, i.e., 'SendAddInfRequest'.

Then the second operand of the 'BoundedWith' pattern, i.e., 'SendAddInfRequest' is added in a sub-process, which starts by an event-based gateway that catches either an escalation event or an 'OK' signal from the global 'CheckBoundedOfDuties' process, indicating the escalation of a potential violation or the execution of the 'SendAddInfRe-

quest' normally, respectively. To offer some flexibility, if an escalation event is received a defined higher-level role will decide whether to resume the execution and override the 'BoundedWith' rule or to invoke a defined violation Action and terminate.

5 Related work

With the increase in attention paid to the role of compliance in organizations given the high-cost associated with non-compliance, including business failures, bankruptcy, significant fines and even criminal penalties, several work efforts have been produced in the area of compliance management attempting to address the current needs of organizations. The main focus of this paper is on runtime compliance monitoring, therefore, in the next discussion, prominent related-work efforts is summarized and appraised against the work proposed in this paper.

In the literature, runtime monitoring requires business process models to be reduced to some abstract representation, which are built up by collecting runtime information (e.g. exchanged messages sequences, performed activities). On the other hand, runtime monitoring also requires compliance requirements to be structurally/formally represented using a formal/structural language, e.g. LTL, CTL, ECA rules. In addition, various querying languages could also be utilized, such as *BP-Mon* [14] and XPath [59]. The actual compliance checking between abstract traces and formal rules/queries is performed by a runtime compliance checker (engine), which is usually an *external component* that is incorporated into the execution environment, but could also be an internal component. The checker can check the adherence to the requirements either after the execution is completed, or synchronous with the execution, following a more proactive approach. In the following, we classify related work into four categories; *graph-based* approaches, *formal-based* approaches, *XML querying* approaches and *complex-event* processing, which will be discussed in the following and appraised against the work presented in this paper.

Graph-based approaches mainly target the design-time phase of the business process lifecycle for (sub-)process models querying, substitution,and compliance checking; examples are: [29], [51], [17]. On the other hand, few studies [14,33] have addressed runtime compliance monitoring. *Business Process Monitoring* (*BP-Mon*) is a graphical query language proposed by Beeri et al. [14] to visually represent monitoring requirements against BPEL models, abstracted into event traces. Graph matching techniques (*homomorphism*) are then exploited to evaluate the compliance of *completed* running BPEL instances, focusing on control-flow and timing constraints. Similarly, the study in [33] adopts a graph-based compliance rule language to capture compliance requirements, supporting sequence, data and real-time constraints, where runtime compliance checking is done *synchronously* with the execution.

Influential *formal monitoring* approaches are reported in [12,13,25,27,36,37,40] by founding compliance requirements on a formal/mathematical language. The study in [37] uses *Event Calculus* (EC) as the formal basis of monitored constraints against BPEL models. EC is an expressive language; however it is excessively difficult to use. Monitoring is implemented using an integrity-checking technique on *completed* executions. EC is also used in [40], however to cope with the complexity of EC, *Declare* language [49] is utilized as a graphical intermediate representation. Logic programming reasoning is then used to dynamically reason about partial, evolving execution traces. These approaches [9,40] focus on control-flow and timing constraints.

Model-checking formal approaches is adopted in [9,27,35]. *LTL-FO+* is proposed in [27] as an extension to LTL that includes full first order quantification over data, focusing on control-flow and data requirements. In [35], *Declare* [49] is used, which is mapped into LTL, only supporting control-flow constraints, where monitoring is done synchronously with the execution. The same approach is applied in [36] using Declare and LTL to capture compliance requirements, while declarative process models are considered instead, mainly to detect conflicting compliance requirements.

Metric first-order temporal logic is used in [13], supporting past and bounded future operators. This approach [13] provides an optimized monitoring technique addressing control-flow and timing constraints, however the complexity of the adopted logic is not tackled. An extension is made in [12] to support data-constraints. The REALM model is proposed in [25] which constitutes (among others) a conceptual model and metadata. The conceptual model captures the concepts and relationships related to a certain domain (domain ontology), which are used to build compliance rules. To ensure the rigor of the framework, compliance rules are first represented formally using Past LTL, then mapped to proprietary notations. Compliance checking is also performed by a proprietary component (active correlation technology (ACT)) that correlate events to detect runtime violations. The approach supports control-flow and real-time constraints.

Prominent *XML querying* approaches are [26], [58], [52]. In [26] and [58], requirements in LTL are translated into equivalent XQuery expressions, and an XQuery engine is used to evaluate the compliance, focusing on sequence and data constraints. *BPath* [52] is proposed as an XPath extension with LTL modalities. BPath expressions are then mapped into XPath, and a native XML query engine is utilized, supporting sequence and timing constraints.

Influential rule-based proposals [9,11,15,43,44]. In [11] proposed that desired properties and constraints on BPEL systems be specified in Web service constraint language (WS-CoL), a special-purpose assertion specification language that borrows its roots from JML (Java Modeling Language) and extends it with constructs to gather data from external sources. WSCoL are interweaved into BPEL specification, and a dedicated monitoring manager evaluates the compliance by focusing on data constraints. In [15], compliance requirements are represented in Prolog and verified against a workflow language, supporting sequence and timing constraints using a rule engine.

In [43] a generic runtime compliance management framework is proposed, which is based on a set of Dwyer's property specification patterns [18], and provides a high-level conceptual model for compliance requirements refinements and the definition of recovery actions, as response to detected violations. The framework is realized by implementing it using BPMN models and event-condition-action (ECA) rules. This approach is closely related to the work presented in this paper, however, our work relies on a wider set of novel compliance patterns; moreover, our approach addresses the four structural facets of the BP lifecycle; and we define a novel evaluation approach based on anti-patterns.

Complex event processing (CEP) technology is utilized in [41,53,54,60]. Prominent efforts in this direction use event pattern languages (EPLs) to capture relevant requirements and constraints. In [41], a model-driven engineering approach is adopted, such that a high-level DSL language is introduced for the abstract specification of compliance constraints, with support for sequence and resource constraints.

The work in [60] only considers sequential requirements, where an approach is also introduced to filter and aggregate query results to provide compact feedback on deviations. Business processes are modelled in [53] as event flows where compliance requirements are structurally represented in a conceptual graphical rule model the authors also proposed; and then a CEP engine (SARI) [42] is utilized to check the compliance, with support for sequence and timing constraints. Major approaches in this category check compliance synchronously with the execution.

Several monitoring tools [1] have been developed for the cloud environments. However, these tools have mainly focused on monitoring the low level aspects of the computing infrastructure (e.g., memory, disk, CPU). To the best of our knowledge, this is the first approach that considers the monitoring and compliance of the high level and logical aspects of the business processes in the cloud environments without relying on any external compliance monitoring component.

6 Conclusion

This paper presented a pattern-based process rewriting approach to embed compliance monitoring logic within process definition. An advantage of this approach is to have a self-contained process definition that includes both the business and the compliance monitoring logic. Another advantage is to simplify the information system infrastructure required to enable compliance monitoring of running process instances. This is particularly useful in cloud computing environments where external monitoring is not applicable. The approach monitors runtime-related aspects such as timing and resource assignment constraints which would be practically impossible to assure their satisfaction at process design time. An assumption is that the process is compliant by design with the ordering or existence constraints within a compliance rule. If this is not the case for some rules, our approach is still able to find violations. For instance, for response and precedence rules if the consequent is missing in the process definition, monitoring logic is still able to alert for violations. However, it is recommended to run static compliance checking on the process definition before putting it into production as has been covered by a large body of research as shown in Sect. 5.

As a proof-of-concept, we have developed an integrated open-source tool suite as an instantiation artefact of the proposed self-alerting approach by considering BPMN (cf. Sect. 4.1). However, the approach is generic and can be applied to other business process modelling languages, where the development of some APIs that returns required runtime information, for example the 'Roles' assigned to a specific 'User', may be needed to realize our approach with other business process languages. Embedding compliance monitoring logic within process definition will help, in addition to monitoring for violations, simplify the infrastructure landscape required for process execution. This is achieved by eliminating the need for an external monitoring component. Especially for a cloud setting, this is of great value as host configuration options might be limited in some cases. Moreover, the message exchange and event streams will be much reduced thus making less footprint and thus less load on the hosting environment.

A limitation of our approach is that it is designed to be effective in managed process execution environments which requires process-aware information system and a centralalized process orchestration engine. This is a high-level maturity that organizations are working to reach. However, when this is in place, the required infrastructure will be much simpler as there will be no need to have external monitoring components and a lot of event streams between the execution component and the monitoring component will not be needed any more.

Acknowledgments This work was supported by King Abdulaziz City for Science and Technology (KACST) project 11-INF1991-03.

References

1. Alhamazani, K., Ranjan, R., Mitra, K., Rabhi, F.A., Jayaraman, P.P., Khan, S.U., Guabtni, A., Bhatnagar, V.: An overview of the commercial cloud monitoring tools: research dimensions, design issues, and state-of-the-art. Computing **97**(4), 357–377 (2015)
2. Awad, A., Barnawi, A., Elgammal, A., El Shawi, R., Almalaise, A., Sakr, S.: Runtime detection of business process compliance violations: an approach based on anti patterns. In: Wainwright, R.L., Corchado, J.M., Bechini, A., Hong, J. (eds.) Proceedings of the 30th Annual ACM Symposium on Applied Computing, Salamanca, Spain, April 13–17, 2015, pp. 1203–1210. ACM (2015)
3. Awad, A., Pascalau, E., Weske, M.: Towards instant monitoring of business process compliance. In: EMISA Forum, vol. 30 (2010)
4. Awad, A., Weidlich, M., Weske, M.: Specification. Verification and explanation of violation for data aware compliance rules, In: ICSOC/ServiceWave (2009)
5. Awad, A., Weske, M.: Visualization of compliance violation in business process models. In: BPM Workshops (2009)
6. Awad, A., Decker, G., Weske, M.: Efficient compliance checking using BPMN-Q and temporal logic. In: BPM (2008)
7. Baldwin, R., Cave, M., Lodge, M.: Understanding Regulation: Theory, Strategy, and Practice. Oxford University Press (2011)
8. Banescu, S., Petkovi, M.: Measuring privacy compliance using fitness metrics. In: BPM (2012)
9. Barbon, F., Traverso, P., Pistore, M., Trainotti, M.: Run-time monitoring of instances and classes of web service compositions. In: ICWS (2006)
10. Baresi, L., Guinea, S., Pasquale, L.: Self-healing BPEL processes with dynamo and the JBoss rule engine. In: ESSPE (2007)
11. Baresi, L., Guinea, S.: Towards dynamic monitoring of ws-bpel processes. In: Benatallah, B., Casati, F., Traverso, P. (eds.) Service-Oriented Computing—ICSOC 2005. Lecture Notes in Computer Science, vol. 3826, pp. 269–282. Springer, Berlin (2005)
12. Basin, D., Harvan, M., Klaedtke, F., Zalinescu, E.: Monpoly: monitoring usage control policies. In: Proceedings of the 2nd International Conference on Runtime Verification (RV 2011), pp. 360–364 (2012)
13. Basin, D., Klaedtke, F., Müller, S., Pfitzmann, B.: Runtime monitoring of metric first-order temporal properties. In: Hariharan, R., Mukund, M., Vinay, V. (eds.) IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, Leibniz International Proceedings in Informatics (LIPIcs), vol. 2, pp. 49–60. Dagstuhl, Germany (2008). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik
14. Beeri, C., Eyal, A., Milo, T., Pilberg, A.: Monitoring business processes with queries. In: VLDB (2007)
15. Chesani, F., Mello, P., Montali, M., Riguzzi, F., Sebastianis, M., Storari, S.: Checking compliance of execution traces to business rules. In: Ardagna, D., Mecella, M., Yang, J. (eds.) Business Process Management Workshops. Lecture Notes in Business Information Processing, vol. 17, pp. 134–145. Springer, Berlin (2009)
16. Decker, G., Overdick, H., Weske, M.: Oryx-sharing conceptual models on the Web. In: Conceptual Modeling—ER (2008)
17. Delfmann, P., Herwig, S., Lis, L., Stein, A., Tent, K., Becker, J.: Pattern specification and matching in conceptual models - a generic approach based on set operations. Enter. Model. Inf. Syst. Archit. **5**(3), 24–43 (2010)
18. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Patterns in property specifications for finite-state verification. In: ICSE (1999)
19. El Kharbili, M., de Medeiros, A.K.A., Stein, S., Van Der Aalst, W.M.P.: Business process compliance checking: current state and future challenges. In: MobIS (2008)
20. El Kharbili, M., Ma, Q., Kelsen, P., Pulvermueller, E.: Policy-based and model-driven regulatory compliance management. In: EDOC, CoReL (2011)
21. Elgammal, A., Butler, T.: Towards a framework for semantically-enabled compliance management in financial services. In: 1st International Workshop on Knowledge Aware Service Oriented Applications (KASA?15), co-located with ICSOC. Lecture Notes in Computer Science. Springer, Berlin (2014)
22. Elgammal, A., Turetken, O., Jan van den Heuvel, W., Papazoglou, M.: Formalizing and appling compliance patterns for business process compliance. In: Software and Systems Modeling, pp. 1–28 (2014)
23. Elgammal, A., Turetken, O., Jan van den Heuvel, W., Papazoglou, M.: Root-cause analysis of design-time compliance violations on the basis of property patterns. In: ICSOC, LNCS, vol. 6470. Springer (2010)
24. FATF-GAFI. Fatf 40 recommendations standard. Technical report (2003)
25. Giblin, C., Mueller, S., Pfitzmann, B.: Towards model-driven compliance automation, From regulatory policies to event monitoring rules (2006)
26. Hallé, S., Villemaire, R.: XML methods for validation of temporal properties on message traces with data. In: OTM (2008)
27. Hallé, S. Villemaire, R.: Runtime monitoring of message-based workflows with data. In: EDOC (2008)
28. Hartman, T.: The Cost of Being Public in the Era of Sarbanes-Oxley. Foley and Lardner LLP (2006)
29. Kühne, S., Kern, H., Gruhn, V., Laue, R.: Business process modeling with continuous validation. J. Softw. Evol. Process **22**(7), 547–566 (2010)
30. Luckham, D.: The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems. Addison-Wesley (2002)
31. Ly, L.T., Maggi, F.M., Montali, M., Rinderle-Ma, S., Van Der Aalst, W.M.P.: A framework for the systematic comparison and evaluation of compliance monitoring approaches. In: EDOC (2013)
32. Ly, L.T., Rinderle-Ma, S., Dadam, P.: Design and verification of instantiable compliance rule graphs in process-aware information systems. In: CAiSE (2010)
33. Ly, L.T., Rinderle-Ma, S., Knuplesch, D., Dadam, P.: Monitoring business process compliance using compliance rule graphs. In: OTM (2011)
34. Maggi, F.M., Di Francescomarino, C., Dumas, M., Ghidini, C.: Predictive monitoring of business processes. In: CAiSE (2014)
35. Maggi, F.M., Montali, M., Westergaard, M., Van Der Aalst, W.M.P.: An approach based on colored automata. In: BPM, Monitoring Business Constraints with Linear Temporal Logic (2011)
36. Maggi, F.M., Westergaard, M., Montali, M., van der Aalst, W.M.P.: Runtime verification of ltl-based declarative process models. In: Khurshid, S., Sen, K. (eds.) Runtime Verification. Lecture Notes in Computer Science, vol. 7186, pp. 131–146. Springer, Berlin (2012)
37. Mahbub, K., Spanoudakis, G.: A framework for requirements monitoring of service based systems. In: ICSOC (2004)
38. Mell, P., Grance, T.: Definition of cloud computing. Technical report. National Institute of Standard and Technology (NIST) (2009)
39. Mendling, J., Ploesser, K., Strembeck, M.: Specifying separation of duty constraints in BPEL4 people processes. In: BIS (2008)
40. Montali, M., Maggi, F.M., Chesani, F., Mello, P., van der Aalst, W.M.P.: Monitoring business constraints with the event calculus (2013)

41. Mulo, E., Zdun, U., Dustdar, S.: Domain-specific language for event-based compliance monitoring in process-driven SOAs. Serv. Orient. Comput. Appl. **7**(1) (2013)
42. Mulo, E., Zdun, U., Dustdar, S.: Monitoring web service event trails for business compliance. In: SOCA, pp. 1–8. IEEE (2009)
43. Namiri, K., Stojanovic, N.: Pattern-based design and validation of business process compliance. In: Proceedings of the 2007 OTM Confederated International Conference on On the Move to Meaningful Internet Systems: CoopIS, DOA, ODBASE, GADA, and IS—Volume Part I, OTM'07, pp. 59–76. Springer, Berlin (2007)
44. Narendra, N.C., Varshney, V.K., Nagar, S., Vasa, M., Bhamidipaty, A.: Optimal control point selection for continuous business process compliance monitoring. In: IEEE/SOLI 2008. IEEE International Conference on Service Operations and Logistics, and Informatics, 2008, vol. 2, pp. 2536–2541, Oct (2008)
45. OASIS. Web services - human task (ws-humantask) version 1.1. Technical report (2010)
46. OASIS. Web services business process execution language version 2.0. Technical report (2007)
47. OASIS. Ws-bpel extension for people (bpel4people) specification version 1.1. Technical report (2010)
48. Object Management Group. Business process model and notation specification 2.0.2. Technical report (2013)
49. Pesic, M., Schonenberg, H., van der Aalst, W.M.P.: DECLARE: full support for loosely-structured processes. In: 11th IEEE International Enterprise Distributed Object Computing Conference (EDOC 2007), 15–19 October 2007, Annapolis, Maryland, USA, pp. 287–300. IEEE Computer Society (2007)
50. Reuter, P., Truman, E.M.: Chasing dirty money: the fight against money laundering. Institute for International Economics (2005)
51. Sakr, S., Awad, A.: A framework for querying graph-based business process models. In: Proceedings of the 19th International Conference on World Wide Web, WWW '10, pp. 1297–1300. ACM, New York, NY, USA (2010)
52. Sebahi, S., Hacid, M.S. Business process monitoring with bpath—(short paper). In: OTM Conferences (1) (2010)
53. Thullner, R., Rozsnyai, S., Schiefer, J., Obweger, H., Suntinger, M.: Proactive business process compliance monitoring with event-based systems. In: EDOC Workshops (2011)
54. Thullner, R., Rozsnyai, S., Schiefer, J., Obweger, H., Suntinger, M.: Proactive business process compliance monitoring with event-based systems. In: Enterprise Distributed Object Computing Conference Workshops (EDOCW), 2011 15th IEEE International, pp. 429–437, Aug (2011)
55. Van Der Aalst, W.M.P., De Medeiros, A.K.A.: Process mining and security: detecting anomalous process executions. In: WISP (2004)
56. van der Aalst, W., van Hee, K., van der Werf, J.M., Kumar, A., Verdonk, M.: Conceptual model for online auditing. Decis. Support Syst. **50**(3) (2011)
57. Van Der Werf, J.M., Verbeek, E., Van Der Aalst, W.M.P.: Context-aware compliance checking. In: BPM (2012)
58. Venzke, M.: Specifications using xquery expressions on traces. Electron. Notes Theory Comput. Sci. **105**, 109–118 (2004)
59. W3C. Xml path language (xpath) 2.0 (second edition) (2011)
60. Weidlich, M., Ziekow, H., Mendling, J.: Event-based monitoring of process execution violations. In: BPM (2011)
61. Wolter, C., Miseldine, P., Meinel, C.: Verification of business process entailment constraints using spin. In: Massacci, F., Jr., Redwine, S.T., Zannone, N. (eds.) Engineering Secure Software and Systems. Lecture Notes in Computer Science, vol. 5429, pp. 1–15. Springer, Berlin (2009)
62. Xiangpeng, Z., Cerone, A., Krishnan, P.: Verifying bpel workflows under authorisation constraints. In: Dustdar, S., Fiadeiro, J., Sheth, A.P. (eds.) Business Process Management. Lecture Notes in Computer Science, vol. 4102, pp. 439–444. Springer, Berlin (2006)

Ahmed Barnawi is an Associate Professor at the Faculty of Computing and Information Technology, King Abdulaziz University (KAU), Saudi Arabia. He received his PhD in Communications Engineering from the University of Bradford, UK in 2006. He is currently the managing director of Cloud Computing Research group at KAU. He is a holder of multiple patents in wireless communications. His research interests include Business Process Management, Big Data systems and cloud computing.

Ahmed Awad is an Assistant Professor of information systems at the Faculty of Computers and Information, Cairo University, Egypt. He received his Ph.D. from Hasso-Plattner Institute, University of Potsdam, Germany in 2010. He received his B.Sc. and M.Sc. degrees in Information Systems from the Faculty of Computers and Information, Cairo University, Egypt, in 2000 and 2003 respectively. The research interests of Awad are business process management in general and study of compliance management of business processes in specific. He has published more than 10 journal, conference and journal papers in the topic of business process compliance.

Amal Elgammal is an Assistant Professor at Faculty of Computers and Information, Cairo University. She has obtained her Ph.D. in Information Systems from Tilburg University, the Netherlands in 2012. She received distinguished Bachelor and Masters degrees in Information Systems from Faculty of Computers and Information, Cairo University (2001 and 2007, respectively). She has been appointed with Trinity College Dublin, Future Cities research group (2014–2015), and Governance, Risk and Compliance Management Technology Centre (GRCTC) at University College Cork (2013–2014). Her research interests revolve around: Business process management; Ontology engineering; Process mining; Business Analytics and Intelligence; Governance, Risk Management and Compliance (GRC), Internet-of-Things; Smart Manufacturing; Smart Healthcare.

Radwa El Shawi is an Assistant Professor in college of Computer Science and Information Technology at Princess Nora Bint Abdul Rahman University, Riyadh, Saudi Arabia. She received her PhD degree from School of Information Technologies, University of Sydney, Australia in 2013. She obtained her B.Sc. and M.Sc. degrees in Computer Engineering from, Arab Academy for Science and Technology and Maritime Transport, Egypt in 2005 and 2008 respectively. Her research interests lie in the areas of graph theory, algorithms and data structures and geometric networks.

Abdullah Almalaise is Associate Professor in Faculty of Computing and Information Technology at King Abdulaziz University, Saudi Arabia. He received his Ph.D. in Computer Science from George Washington University, (2003) and M.SC in Management Information Systems from University of Illinois at Springfield (2001). He received B.Sc in Computer Science from University of Southern Mississippi (1990). His research interests include information systems, business process management and software engineering.

Sherif Sakr is an Associate Professor in the department of Health Informatics at King Saud bin Abdulaziz University for Health Sciences, Saudi Arabia. He is also an Associate Professor of Computer Science at University of New South Wales, Australia and a visiting Senior Researcher at National ICT Australia (NICTA). Previously, He had appointments with Macquarie University (Australia), Microsoft Research (USA), Alcatel Lucent Bell Labs and Cairo University (Egypt). He received his PhD degree in Computer and Information Science from Konstanz University, Germany in 2007. He received his B.Sc. and M.Sc. degrees in Computer Science from the Faculty of Computers and Information in Cairo University, Egypt, in 2000 and 2003 respectively. His research interest include the areas of graph data management, big data storage and processing in cloud computing environments. He is an IEEE Senior Member.