Enabling Compliance Monitoring for Process Execution Engines- Extended Version

Marwa Hussein Zaki, Ahmed Awad, and Osman Hegazy

Information Systems Department, Faculty of Computers and Information, Cairo University, Giza 12613, Egypt
m.hussein, a.gaafar, o.hegazy@fci-cu.edu.eg

Abstract. Business process management aims at controlling and managing business processes in order to achieve business goals within organizations. Most of organizations try to ensure that their processes are compliant with relevant regulations and laws. Runtime monitoring of process compliance is considered to be of crucial importance. Monitoring is achieved by listening to events generated from business process execution. In this regard, there are several frameworks that enable the monitoring based on variants of event processing technologies. Most of these frameworks presume a rich activity lifecycle model for the execution of tasks within business processes. However, most of process execution engines support simpler lifecycle models. Thus, these frameworks fall short in monitoring compliance for such engines due to missing needed input events.
The goal of this paper is to enable compliance monitoring for different process execution engines. First we propose a middleware layer that maps different execution engines' lifecycles states to a reference lifecycle model. Also, unmatched states will be derived from execution's engine states using complex event processing (CEP) techniques. Additionally, we implement compliance anti patterns to prove the feasibility of our approach. We apply the proposed approach to Activiti and jBPM open source process execution engines, the reference lifecycle model is the one supported by the BP-MaaS monitoring framework.

Keywords: Business Process Management; web services; compliance; runtime monitoring; lifecycle mapping

1 Introduction

In the recent years, most of the organizations try to ensure that their business processes are compliant with regulations and laws [5] that are imposed on them. Violations of these rules will cause legal and business problems to the business of those organizations [31]. The need for compliance has motivated researchers and practitioners to develop different approaches to check the compliance state of a business process [31], [5], [21]. These approaches address different aspects of business process compliance and can be employed at different phases of a business process lifecycle [21], e.g., at design time of a business process model [31], at the process execution (runtime) [3] or at the evaluation phase of completed business process instances [27].

Compliance monitoring at process execution time is considered of utmost importance. That is because not every possibility of violation can be checked and eliminated

at process design time. Several aspects of compliance lend themselves to runtime by nature. For example, compliance requirements that refer to timing constraints are best checked at runtime. Several approaches have been developed to enable the monitoring of running processes compliance status [22], [7], [3]. Most of the monitoring frameworks presume a rich activity lifecycle model, cf. [3], for the execution of tasks within business processes compared to the simple task lifecycle models supported by execution engines. So, many of the expected execution events by the monitoring frameworks will be missing. This introduces the threat of having violations go undetected. To make use of those frameworks, the gap between what is provided by execution engines and what is expected by monitoring frameworks has to be filled.

In this paper, we enable compliance monitoring for different process execution engines. We propose a mapping approach that fills the gap between the activity lifecycle models supported by different process execution engines and the reference lifecycle model supported by Russell et al. [29] work and the BP-MaaS [3] monitoring framework. In order to achieve the compliance monitoring, we implement compliance patterns presented in [3]. We have studied two different execution engines: Activiti [26] and jBPM [11]. For each, we have developed the supported task lifecycle, compared it to the reference lifecycle and identified the mapping. Some events are directly mapped, as they have same semantics but different naming. Other events cannot be directly mapped. We use CEP technology for the mapping process in Activiti and Drools rule engine to apply CEP features in the mapping process and compliance pattern implementation in jBPM.

The rest of the paper is organized as follows: Section 2 discusses some of the background techniques and concepts and presents a simple process model that will be taken as a running example. Section 3 presents our proposed model and reports about our results. Section 4 discusses related work and Section 5 concludes the paper with an outlook on future work.

2 Preliminaries

2.1 Reference Task Lifecycle

In Fig. 1 we use the activity lifecycle which contains a combination of the basic lifecycle of a work item and detour patterns from the work of Nick Russell et al. [29] which is also supported by BP-MaaS framework [3]. However, we exclude the offered state from our scope as we are not supporting any scheduling system that manages the offering mechanism and logic. The lifecycle illustrates the state transitions of tasks as follows: a work item (a task) is *created* by the system. It is either directly *started* or *allocated* to a resource. A task can be delegated to another resource either by the system (escalate), or the responsible resource (delegate). After assigning a resource, the task will be *started*. However, the resource can skip it to be *completed* directly. Moreover, a resource can suspend a task to be *suspended* and resume it again (resume) to be *started* and finally *completed* or (redo) it after completion. If the resource fails to execute the task *failed*, it can try to (restart) it again or (reallocate) to another resource .

The states according to the reference model are: *created*, *allocated*, *started*, *suspended*, *failed*, *completed*. These states are the main ones which we need to apply the

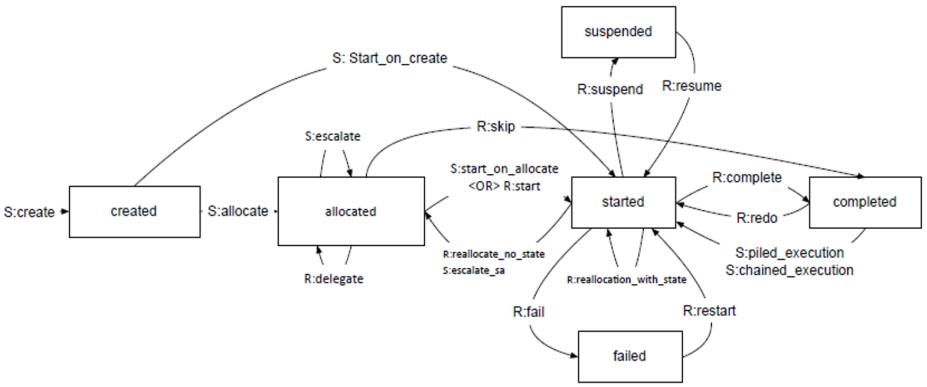

Fig. 1: Reference task lifecycle model adapted from [29]

mapping from any execution engine to them. The main actions which led to the previous states are: *create*, *allocate*, *delegate*, *escalate*, *skip*, *reallocate*, *suspend*, *resume*, *fail*, *restart*, *complete*, *redo*, *start_on_create*, *start_on_allocate*. One of the main contributions in this research is to derive any missing states or actions that do not exist or are not supported by the execution engines to supply all information to the monitoring system. For example, if the execution engine does not support the reallocate action, we can derive that a reallocation took place when we detect that a started event occurred followed by an allocated event with a different performer. Also we can derive that a start_on_create action happened when we detect that a created event occurred followed by a started event without any allocated event in between for the same task.

2.2 Complex Event Processing (CEP)

We use Complex Event Processing (CEP) technology to monitor raw events generated from business process execution engines and infer the missing reference events to be used in the monitoring process. CEP is mainly about near real time execution and implementation. It is a well-researched and well founded technology that provides continuous, real-time and actionable insights into events that are flowing through business networks. Complex event processing begins with the arrival of simple events containing raw business data and temporal information. Typically, the events are produced from live data sources such as sensors or from any instrumentation in a business process. The streams of simple events are analyzed in near real time fashion to identify and extract complex events. A complex event is a specific combination of simple events that represents a condition, a trend, or a change that is meaningful to any organization [10], [20]. Esper [1] is a an open source component and engine for the implementation of CEP. It uses a stream oriented language called EPL, it is similar to SQL in syntax but it has been extended with different features [10].

[1] `http://www.espertech.com/esper/`

2.3 Compliance Patterns and Anti-patterns

Based on [3], a set of *compliance patterns* are supported in order to define compliance constraints to be monitored over runtime environments and to detect any violations. Also monitoring is achieved using the *anti-pattern* approach which is considered as a dynamic set of rules that represent the negation to the defined pattern. For example, if we need to detect the absence of task (A) within the scope of the process instance, it is easier to search for the anti-pattern violation of this rule which is the occurrence of task (A) instead of scan the whole process execution to prove that task (A) does not occur. These anti-patterns are independent from any technology and is used in the compliance monitoring process through the different phases of the task lifecycle. We will discuss the implementation of these patterns using *jBPM* and *drools* engines later in the paper. We refer the reader to [3] for more details about the formalization of compliance anti-patterns.

2.4 Running Example

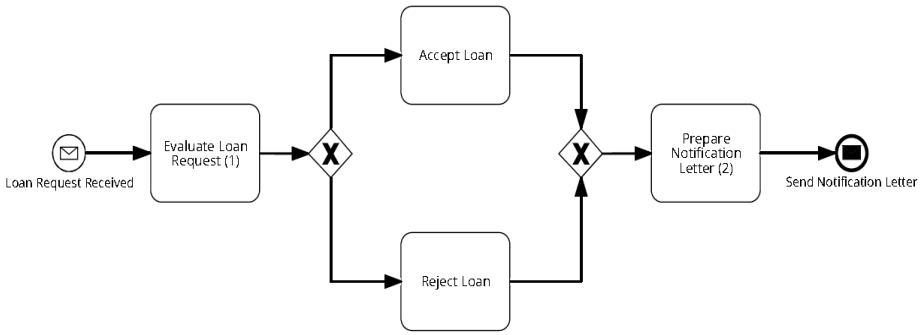

Fig. 2: Simple loan evaluation process

Fig. 2 describes a simple loan evaluation process. The process starts when the user requests a loan from the bank, This loan will be evaluated by one performer. Based on the decision he takes the loan will be accepted or rejected. After that, a notification letter will be prepared and sent to the user with the decision. There are two compliance requirements to be checked in order to achieve the process compliance. First the *Evaluate Loan Request* task should not be delegated to more than one person, also the performer of *Evaluate Loan Request* task and *Prepare Notification Letter* task should not be the same. To check the first compliance requirement, we will use the *Exist* pattern which checks that a certain task occurred specific number of times under some conditions. To check the second compliance requirement, we will use the *separation of duty* pattern which checks that the performer of the tasks is not the same.

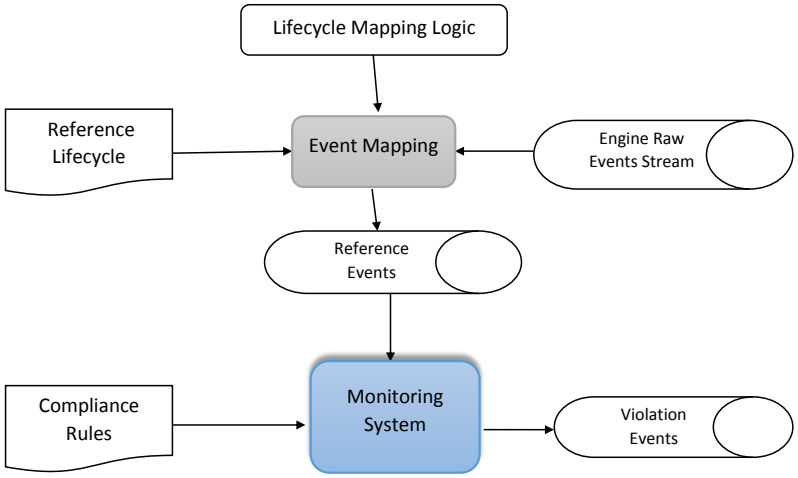

Fig. 3: Proposed monitoring system framework

3 Enabling Monitoring

This section discusses the compliance monitoring enabling middleware which consists of: a) An engine-specific mapping to the reference task lifecycle b) Implementation of the compliance monitoring anti patterns presented in [3]. Fig. 3 depicts the architecture of the middleware. The *Lifecycle Mapping Logic* is a set of logical rules used to perform the mapping from *Engine Raw Events* to the *Reference Events* based on the *Reference Lifecycle*. Currently, these rules are manually derived and programmed into an executable language. For the case of jBPM, the mapping rules are encoded as Drools rules. For the case of Activiti, these are encoded as extensions to the engine and written in EPL syntax, more details are given later in this section. The mapping logic is done offline and once per process execution engine. It only needs to be revisited in case either the engine lifecycle or the reference lifecycle is changed.

At runtime, the *Monitoring System* detects and throws *Violation Events*, if any, based on the input *Compliance Rules* and the events coming from the reference events stream. The following Section discusses into details how the lifecycle mapping works.

3.1 Lifecycle Mapping

This section introduces an overview of the proposed mapping approach. Our approach has two inputs: (1) *Engine Raw Events Stream* which contains the events generated

from the execution of a process model with the engine's lifecycle and (2) *Lifecycle Mapping Logic* which contains a list of logical rules used to obtain the corresponding reference event, cf. Section 2.1. The output of this part is the *Reference Event Stream* which contains the mapped reference events and the derived missing events that the engine does not support to be used again in the monitoring process.

We begin the mapping process by detecting the events generated from the execution engines.

There are four different possibilities for mappings an engine-specific event/state to the reference state. The first possibility is that there is a *direct correspondence*. That means that both the engine activity state and the reference state carry the same name and meaning. The second possibility is that there is a *naming mismatch* but a *conceptual match*. For instance, the engine defines acivity state *assigned* which is matched with the reference state *allocated*. The third possibility is that the reference state can be derived via observing a pattern over one or more of the engine's states. As an example, the reference state transition *resumed* can be derived by detecting a sequence of *suspended* and *started* events of the same task instance and the same performer. The last possibility is that there is *no mapping possible*, this means that the engine doesn't support this state through its service API.

We investigated different execution engines' lifecycles for different languages such as BPMN, BEPL and workflow systems (YAWL) Table 1 presents a small survey about these engines.

We report implementation on *jBPM* [11] and *Activiti* [26] as *Camunda* [2] and *Activiti* almost have the same API functions. We investigated the YAWL workflow engine's lifecycle without a specific implementation as this workflow system is more scientific and complex for regular user rather than BPMN execution engines which are easier to understand and use.. Table 2 summarizes the mapping between the different states of these engines's lifecycle to the reference lifecycle.

The direct mapping and the naming mismatch are straightforward. In the latter case, when the engine generates an event with the mis-named state, our approach simply generates another event copying all the data of the original event except for the state where the conceptually equivalent state is substituted. In the third case, the approach can derive the reference state by observing the occurrence of one or more of the engine's events. This process happens by adding engine extension to throw a new event when the engine performs an operation on the task that is not supported by its lifecycle.

The following rules are specific for each engine. These rules explain how the mapping and the reference states can be derived based on one or more of the engine's states. For *Activiti* engine:

- **Rule 1:** (*Start_on_Create*) Start_on_Create action can be inferred when task t is created with performer r_i and the system directly start this task. If the start event is not supported by the engine, we considered that this task is immediately started after creation and we throw a new event of type *start_on_create* which will be mapped to *started* to represent the missing start event.

[2] `https://camunda.com/`

Table 1: List of BPMN execution engines

Execution Engine	**Type**	**open/closed source**	**available doc**	**task/activity lifecycle states**
Tim	BPMN	closed	no	-
Sydle Seed	BPMN	free trial for modeler,no access to the engine	yes only for modeler	active,completed, completed with errors, aborted
orchestra	BPMN	open	yes	for process: active,retired,undeployed
Bonita BPM	BPMN	open	yes	ready,completed,failed
Omni workflow	BPMN	open	not for the engine	-
Activiti	BPMN	open	yes	create,assignment, complete
Camunda	BPMN	open	yes	new,unassigned, assigned,delegated, completed
jBPM	BPMN	open	yes	created,reserved, inprogress,suspended, failed,completed,error, exited,obsolete
Bosch	BPMN	closed	no	-
imixs	BPMN	open	yes but not complete	no explicit lifecycle
ProcessMakerWorkflow	BPMN	closed	no	-
ActiveVOS	BPEL	need license	yes	ready,reserved, inprogress,suspended, failed,completed, error,obsolete
ApacheODE	BPEL	open	yes	based on WS-BPEL standards
ExpressBPEL	BPEL	closed	no	-
iBolt Server	BPEL	closed	no	-
Parasoft BPEL Maestro	BPEL	closed	no	-
YAWL workflow engine	workflow	open	yes	Enabled, Fired, Executing, Complete, Is parent, Deadlocked, Cancelled, Withdrawn, ForcedComplete,Failed, Suspended, CancelledByCase, Discarded

Table 2: Mapping the lifecycle of different engines to the reference lifecycle

Reference Lifecycle states	**Engine Lifecycle States**			
	Activiti	**jBM**	**Camunda**	**YAWL**
Created	create	created	new	Enabled
Allocated	assignment	reserved	assigned	Enabled with resource information
Started	Rule 1	in progress	Rule 1	Fired
Suspended	Rule 2	suspended	Rule 2	Suspended
Failed	no mapping possible	failed	no mapping possible	Failed
Completed	complete	completed	completed	Complete

– **Rule 2:** (*Suspended*) Suspended event can be inferred when the whole process instance is suspended as the engine doesn't support the suspended state on the task level. when the engine report a wait state for the process instance, we can detect that a suspended event occurred and a new event of type *suspended* is thrown to the stream.

After the investigation of the Activiti engine, we found that the failed state is not supported into it and there is no way of mapping this state.

3.2 Compliance Monitoring Implementation

This subsection provides the implementation details of the two components of the proposed compliance monitoring framework using *Activiti* and *jBPM* engines. Complex Event Processing with the help of *Esper* engine, which is responsible for event analysis in CEP, is used to implement the mapping approach on *Activiti* process engine. The mapping approach on *Activiti* consists of mainly 3 parts: (1) *Business Process Model*; that represents the actual process model that will be presented in the form of an XML document. This document contains the exact description of every part and notation of the process's BPMN diagram, and this is where the task listeners will be defined for the first time to detect any change in the task's state that will happen through the execution of the model, (2) *Task Listeners*; It is responsible for detecting the change in the task's state, and send the information about it to a predefined event that will be thrown to the Esper engine to be analyzed later, (3) *Esper Engine*; it is the main part of the Esper which contains all the components related to it. It is responsible for generating stream and populate it with events that is coming from different sources, initiates listeners to detect events from the stream and throw new events in case of any changes after processing, communicate between our proposed approach and the stream to throw and receive mapped events. (4) *Stream*; which contains all kinds of events either row, complex or mapped events thrown from our approach. All the business logic including detecting queries is implemented using Event Processing Language (EPL).

The following lists present a sample query of the implementation of the mapping cases. It introduces sample of the EPL statements used in the naming mismatch case in *Activiti* for the assignment state.

Listing 1.1: EPL statements for naming mismatch for the allocated state

```
insert into referenceeventsstream(taskInstanceId, receiveTime,
eventType, performer ) select taskInstanceId,receiveTime,
'allocated',performer from engineraweventsstream where
eventType = ? "
```

The mapping lifecycle with *jBPM* is a bit different. jBPM is an open source business process management suite allows user to execute business processes using latest BPMN notations. The main lifecycle states extracted from Russell et al.[28] and mapped by our mapping approach are: *created*, *reserved*, *inprogress*, *suspended*, *failed*, and *completed*. *jBPM* supports the CEP technology through the integration with Drools which is a Business Rules Management System (BRMS) solution [2]. This Drools engine supports CEP streams and events concepts by applying rule engine semantics. The mapping approach on *jBPM* starts by defining an active session to the engine to register the streams which contains the events generated from the execution of the process. After that the drools rules match the incoming events with the defined mapping logic and fire when a mapping match is found. These rules acts like EPL statements in Esper engine. Whenever a change happens in the task's state, a new event is thrown to the entry point (acts as the stream in Esper) and the corresponding rule fires when there is a match between the rule content and the events sent through the entry point.

The following list introduces sample of the drools rule used in the naming mismatch case in *jBPM* for the inprogress state.

Listing 1.2: inprogress detection drools rule

```
rule "inprogress␣Rule"
when
$mevent : events( tasktype == "InProgress") from entry-point engineStream
then
$mevent.seteventType("started");
update($mevent);
```

The second part of our approach is the implementation of the compliance rules using the anti patterns technique presented in [3]. We implemented most of the patterns, e.g. *exist*,*absence*, *response*, *one to one response*, *next*, *sequence* and *separation of duty* using the rule semantics logic of the drools engine [2]. It is combined with jBPM to model the combination of business process with business rules and complex event processing technology. The framework uses the reference events stream and the compliance patterns entered as input and match each pattern with its corresponding anti-pattern rule to detect any possible violation as depicted in Fig. 3.

We implemented this part using the drools fusion component which contains complex event processing features. The anti patterns queries is written using the drools rule language (DRL) syntax as drools rules and stored in the production memory of the drools engine. The compliance rules which present one of the inputs for the compliance monitoring part is stored in the working memory of the engine as "facts" to be compared later with the stored drools rules. The stream containing events in drools engine is called entry points and hold raw and mapped events from the mapping part. Based on

the entered compliance rules and the events from the entry point, the drools rules fire and throw a violation alert when a violation to the compliance rules occurs.

The following list introduces sample of the rules used to implement the compliance rules anti-patterns. These anti patterns are written in drools rule language (DRL) [2].

Listing 1.3: Exist anti pattern rule

```
1
2 Exist anti pattern
3 when
4 Event ( $task : task, $ts : tistamp, $type : type,
5 $pi : processinstance) from entry-point Stream
6 $comprules : Comprules ( pattern == "exist", scopestart == $type,
7 $mul: multiplicity )
8 $total: Number( intValue > $mul )
9 from accumulate ( Event ( task == $comprules.antecedent,
10 tistamp > $ts, type == $comprules.scopeend, processinstance == $pi,
11 $ant: task ) from entry-point Stream,count($ant))
12 then
13 System.out.println("Exist␣pattern␣VIOLATION␣-->␣the␣antecedent␣count
14 is␣GREATER␣THAN␣multiplicity,␣Evaluate␣Loan␣request␣delegation
15 happened␣more␣than␣once␣");
```

The full set of implemented rules and mapping part implementation on *Activiti* and *jBPM* engines could be found here: https://github.com/MarwaHusseinZaki/Lifecycle-Mapping.git

3.3 Results

```
>>> Throwing event: task create successfully: LoanRequestReceivedEvent{ taskInstanceId= 8 ,receiveTime= Sun Feb 26 16:23:49
EET 2017 ,eventtype= create ,performer= Galal }

event mapped successfully: {receiveTime=Sun Feb 26 16:23:49 EET 2017, performer=Galal, taskInstanceId=8, eventType=created}

task startoncreate successfully: LoanRequestReceivedEvent{ taskInstanceId= 8 ,receiveTime= Sun Feb 26 16:23:49 EET 2017
,eventtype= startoncreate ,performer= Galal }

event mapped successfully: { taskInstanceId= 8 ,receiveTime= Sun Feb 26 16:23:49 EET 2017,eventtype= started ,performer= Galal }

>>> Throwing event: task assignment successfully: LoanRequestReceivedEvent{ taskInstanceId= 8 ,receiveTime= Sun Feb 26
16:23:49 EET 2017,eventtype= assignment ,performer= Marwa }

event mapped successfully: {receiveTime=Sun Feb 26 16:23:49 EET 2017, performer=Marwa, taskInstanceId=8,
eventType=allocated}

>>> Throwing event: task assignment successfully: LoanRequestReceivedEvent{ taskInstanceId= 8 ,receiveTime= Sun Feb 26
16:23:49 EET 2017 ,eventtype= assignment ,performer= Alaa }

event mapped successfully: {receiveTime=Sun Feb 26 16:23:49 EET 2017, performer=Alaa, taskInstanceId=8, eventType=allocated}

>>> Throwing event: task complete successfully: LoanRequestReceivedEvent{ taskInstanceId= 8 ,receiveTime= Sun Feb 26 16:23:49
EET 2017 ,eventtype= complete ,performer= Alaa }

event mapped successfully: {receiveTime=Sun Feb 26 16:23:49 EET 2017, performer=Alaa, taskInstanceId=8,
eventType=completed}

Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 5.92 sec

Results :

Tests run: 1, Failures: 0, Errors: 0, Skipped: 0
```

Fig. 4: Activiti engine sample output

In our example mentioned in section 2.4, the loan evaluation task will first be allocated and started with a performer *Galal*, suspended for a while and will be delegated twice to two different performers *Marwa* and *Alaa* and the task will be completed with the third performer *Alaa*. Due to some internal circumstances, the same person *Alaa* performed the loan evaluation task and prepare notification letter task. We executed the process model on *Activiti* and *jBPM* process engines. We found that based on the implemented logic rules queries in the mapping part on Activiti engine, the naming mismatch queries mapped the Activiti row event to the reference events based on the reference lifecycle. For example, every *assignment* event generated from the engine when the task is first assigned to a performer or delegated to different performer; is mapped to *allocated* event based on the reference lifecycle notations. Every delegate,complete,create or assignment actions in the Activiti engine is detected and a new mapped event is thrown to the stream. Also based on the study of the engine events we found that Activiti engine doesn't support the *start* and *suspended* state, only *create*, *assignment*, *complete* states is supported, so we threw a new events of type *start_on_create* which is mapped to *started* to replace the missing start event in the engine and and of type *suspended* to replace the missing suspend event. The same result is achieved on jBPM engine, the engine's events *created, reserved, inprogress, suspended, completed* are mapped to the reference lifecycle events *created, allocated, started, suspended, completed*.

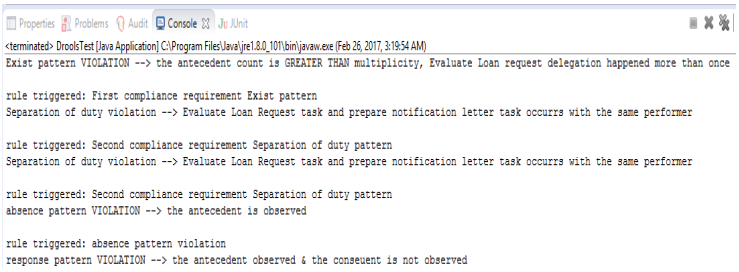

Fig. 5: Compliance Rules anti patterns sample output

Regarding the compliance monitoring violation part implemented on *jBPM* engine; and based on the compliance requirements enforced on the process, we detected that a violation happened to the first compliance requirement which states that "the first task can not be delegated more than once"; as actually this task delegated twice to two different performer *Marwa* and *Alaa*; so a violation alert is raised. As the performer *Marwa* completed the first and last task, another violation event raised to the second requirement which states that "the performer of first and last task should not be the same"

Fig. 4 indicates sample output to the events that is supported by Activiti and the corresponding mapped events in our reference lifecycle while Fig. 5 is a sample output of the compliance anti patterns implementation using jBPM and drools engines.

4 Related Work

Recently, a lot of business process runtime monitoring frameworks were introduced. [1] combines BPM with CEP for monitoring business process execution. They used the activity lifecycle from [33], the refined process structure tree (RPST) [25] to analyze process models, CEP queries and process event monitoring points (PEMPs) to monitor business process execution based on events in semi-automated environments. Similarly in [4], the authors present an extension to Business Process Modeling and Notations (BPMN) which implements a connection between process models and events, along with CEP and PEMPs technique. Also, [7] presents an approach that enables monitoring of business processes with execution data. Their approach combines complex event processing, business process management, and PEMPs techniques. [14] introduces a framework that addresses the gap between events occurring during process execution and the correlation of these events to the corresponding process using PEMPs.

In [8] the authors present a framework for monitoring the progress of task execution and predicting potential problems in its behaviour during runtime, using Support Vector Machines (SVMs) machine learning. The authors in [3] present a runtime business process compliance monitoring framework, BP-MaaS. Their work is based on compliance patterns for the abstract specification of runtime constraints and anti-pattern queries notation to detect runtime compliance violations. It is implemented using CEP technology. The authors in [16] studied the limitations of current CEP technology in supporting anomaly management and proposed a dynamic CEP infrastructure that is able to capture the normal behavior of monitored objects and to create all necessary queries for detecting violations automatically. n [19],[18] authors presented a framework for monitoring business process compliance by introducing the extended Compliance Rule Graph (eCRG) for monitoring compliance rules visually with respect to all relevant process perspective. This framework supports the activity lifecycle by capturing the activity states and implements a correlation mechanism between events. [17] presented a generic framework to monitor process instances from different process perspectives. They are using the lifecycle from [29] and defines the events as a points to be tracked by their monitoring system which carry information from different perspectives. Our work could complement their work as they are not focusing on the resource perspective in their implementation. [9] focused on realizing a monitoring component for the YAWL system using sensors, which monitors conditions that can be achieved through cases.

All the previous work focused on monitoring business process over runtime environment based on events with different task or activity lifecycles. Our work aims at trying to help them by developing a mapping approach that will unify the states of events produced from process execution with their supported lifecycle to remove any inconsistency.

With respect to task lifecycle used in monitoring systems, [13,12] discuss increasing the number of observed events by capturing data state transition events in non automated environment. Their main concern is about the object lifecycle states generated from data objects. Similarly in [15] the authors use object state transitions as additional monitoring information for capturing process progress. [23] addresses the problem of modeling processes with complex data dependencies and their enactment from process models. They focused on the process data level and data objects. Meanwhile, [32]

presents an approach that captures the whole process life cycle and all kinds of changes in an integrated way. They are focusing mainly on control-flow changes. The changes of the resource perspective or data perspective are out of the scope of that paper.

Regarding the business process monitoring with jBPM and drools, [30] describes the design and implementation of an independent, third party contract monitoring service so-called Contract Compliance Checker. It is provided with the specification of the contract to enforce, in order to determine whether the actions of the business partners are consistent with the contract or not. The implementation of their checker is written in the EROP language and translated later to Drools rule engine language. While [24] introduces a layer above a business process, in which the controls are modeled and evaluated against existing process models and instances. It describes an approach for the automation of Internal Controls in an enterprise. They implement this layer using JBoss jBPM and JBoss Rule Engine (Drools). [6] developed a monitoring infrastructure called Glimpse to support runtime performance analysis. They adopted the Drools Fusion rule language in the implementation.

As discussed above, most of the monitoring frameworks use different task lifecycle models. Also, most of researches which address lifecycle models focus on either the data perspective or the lifecycle states extracted from the non automated environments, not generated from execution engine. For those approches that support automated process execution, we position our work as complementary to their work when it comes to actual implementations on available open source process execution engines.

5 Conclusion and Future work

In this paper, we proposed an approach that maps different execution engines' lifecycles states to a reference lifecycle model. Also, the approach infers missing events that the execution engine does not support. The proposed approach serves as a middleware between process enactment and third party monitoring systems. To prove the validity of our middleware, we implemented most of the compliance patterns presented by the BPMaas framework to enable compliance monitoring over jBPM engine. We implemented our approach using Activiti and jBPM open source execution engine and we use CEP technology with Esper, Java and EPL language and CEP with Drools Rule Language (DRL). Our approach focus on the compliance monitoring as one of the many cases of business process monitoring over runtime, but the mapping part can be used by any framework to support any kind of process monitoring. As a future work, we intend to apply our mapping approach on more execution engines to prove its feasibility. Also we will try to expand our monitoring framework by building a dashboard to enable users enter the compliance patterns and determine the events stream to let the framework detect possible violations.

References

1. Michael Backmann, Anne Baumgrass, Nico Herzberg, Andreas Meyer, and Mathias Weske. Model-driven event query generation for business process monitoring. In *ICSOC Workshops - CCSA, CSB, PASCEB, SWESE, WESOA, and PhD Symposium, Berlin, Germany, December 2-5, 2013. Revised Selected Papers*, volume 8377 of *LNCS*, pages 406–418. Springer, 2013.

2. Michal Bali. *Drools JBoss Rules 5.0 Developer's Guide*. Packt Publishing, 2009.
3. Ahmed Barnawi, Ahmed Awad, Amal Elgammal, Radwa Elshawi, Abduallah Almalaise, and Sherif Sakr. An anti-pattern-based runtime business process compliance monitoring framework. *INTERNATIONAL JOURNAL OF ADVANCED COMPUTER SCIENCE AND APPLICATIONS*, 7(2), 2016.
4. Anne Baumgrass, Nico Herzberg, Andreas Meyer, and Mathias Weske. BPMN Extension for Business Process Monitoring. In *Enterprise modelling and information systems architectures - EMISA 2014, Luxembourg, September 25-26, 2014*, volume 318275, pages 85–98. GI, 2014.
5. Jörg Becker, Christoph Ahrendt, André Coners, Burkhard Weiß, and Axel Winkelmann. Modeling and Analysis of Business Process Compliance. In *Governance and Sustainability in Information Systems. Managing the Transfer and Diffusion of IT - IFIP WG 8.6 International Working Conference, Hamburg, Germany, September 22-24, 2011. Proceedings*, volume 366, pages 259–269. Springer, 2011.
6. Antonia Bertolino, Antonello Calabrò, Francesca Lonetti, and Antonino Sabetta. Glimpse: a generic and flexible monitoring infrastructure. In *Proceedings of the 13th European Workshop on Dependable Computing*, pages 73–78. ACM, 2011.
7. Susanne Bülow, Michael Backmann, Nico Herzberg, Thomas Hille, Andreas Meyer, Benjamin Ulm, Tsun Yin Wong, and Mathias Weske. Monitoring of business processes with complex event processing. In *Business Process Management Workshops - BPM 2013 International Workshops, Beijing, China, August 26, 2013, Revised Papers*, volume 171 of *LNBIP*, pages 277–290. Springer, 2013.
8. Cristina Cabanillas, Claudio Di Ciccio, Jan Mendling, and Anne Baumgrass. Predictive task monitoring for business processes. In *Business Process Management - 12th International Conference, BPM 2014, Haifa, Israel, September 7-11, 2014. Proceedings*, volume 8659 of *Lecture Notes in Computer Science*, pages 424–432. Springer, 2014.
9. Raffaele Conforti, Marcello La Rosa, and Giancarlo Fortino. Process monitoring using sensors in YAWL. In *Proceedings of the First YAWL Symposium*, volume 982 of *CEUR Workshop Proceedings*, pages 49–55. CEUR-WS.org, 2013.
10. Opher Etzion and Peter Niblett. *Event Processing in Action*. Manning Publications Company, July 2010.
11. Simone Fiorini and Arun V Gopalakrishnan. *Mastering jBPM6*. Packt Publishing, 2015.
12. Nico Herzberg, Matthias Kunze, and Andreas Rogge-Solti. Towards process evaluation in non-automated process execution environments. In *Proceedings of the 4th Central-European Workshop on Services and their Composition, ZEUS-2012, Bamberg, Germany, February 23-24, 2012*, volume 847, pages 97–103. CEUR-WS.org, 2012.
13. Nico Herzberg and Andreas Meyer. Improving process monitoring and progress prediction with data state transition events. volume 1029, pages 20–23, 2013.
14. Nico Herzberg, Andreas Meyer, and Mathias Weske. An Event Processing Platform for Business Process Management. In *Proceedings of the 17th IEEE International Enterprise Distributed Object Computing Conference*, pages 107–116. IEEE Computer Society, 2013.
15. Nico Herzberg, Andreas Meyer, and Mathias Weske. Improving business process intelligence by observing object state transitions. *Data and Knowledge Engineering*, 98:144–164, 2015.
16. Bastian Hoßbach and Bernhard Seeger. Anomaly management using complex event processing: extending data base technology paper. In *Joint 2013 EDBT/ICDT Conferences, EDBT '13 Proceedings, Genoa, Italy, March 18-22, 2013*, pages 149–154. ACM, 2013.
17. Amin Jalali and Paul Johannesson. Multi-perspective business process monitoring. In *BPMDS*, volume 147 of *LNBIP*, pages 199–213. Springer, 2013.
18. David Knuplesch, Manfred Reichert, and Akhil Kumar. Visually monitoring multiple perspectives of business process compliance. In *BPM*, volume 9253 of *LNCS*, pages 263–279. Springer, 2015.

19. David Knuplesch, Manfred Reichert, and Akhil Kumar. A framework for visually monitoring business process compliance. *Inf. Syst.*, 64:381–409, 2017.
20. Editors David Luckham and Roy Schulte. *Event Processing Glossary - Version 1 . 1*. Number July. 2008.
21. Linh Thao Ly, Stefanie Rinderle-Ma, Kevin Göser, and Peter Dadam. On enabling integrated process compliance with semantic constraints in process management systems - requirements, challenges, solutions. *Information Systems Frontiers*, 14(2):195–219, 2012.
22. Fabrizio Maria Maggi, Marco Montali, Michael Westergaard, and Wil M. P. van der Aalst. Monitoring business constraints with linear temporal logic: An approach based on colored automata. In *Business Process Management - 9th International Conference, BPM 2011, Clermont-Ferrand, France, August 30 - September 2, 2011. Proceedings*, volume 6896, pages 132–147. Springer, 2011.
23. Andreas Meyer, Luise Pufahl, Dirk Fahland, and Mathias Weske. Modeling and enacting complex data dependencies in business processes. In *BPM*, volume 8094 of *LNCS*, pages 171–186. Springer, 2013.
24. Kioumars Namiri and Nenad Stojanovic. A formal approach for internal controls compliance in business processes. In *8th Workshop on business process modeling, development, and support*, pages 1–9, 2007.
25. Artem Polyvyanyy, Jussi Vanhatalo, and Hagen Völzer. Simplified computation and generalization of the refined process structure tree. In *Web Services and Formal Methods - 7th International Workshop, WS-FM 2010, Hoboken, NJ, USA, September 16-17, 2010. Revised Selected Papers*, volume 6551 of *Lecture Notes in Computer Science*, pages 25–41. Springer, 2010.
26. Tijs Rademakers. *Activiti in Action*. Manning Publications, 2012.
27. Elham Ramezani, Dirk Fahland, and Wil M. P. van der Aalst. Where did I misbehave? diagnostic information in compliance checking. In *Business Process Management - 10th International Conference, BPM 2012, Tallinn, Estonia, September 3-6, 2012. Proceedings*, volume 7481 of *LNCS*, pages 262–278. Springer, 2012.
28. Nick Russell and Wil M. P. van der Aalst. Work distribution and resource management in bpel4people: Capabilities and opportunities. In *Advanced Information Systems Engineering, 20th International Conference, CAiSE 2008, Montpellier, France, June 16-20, 2008, Proceedings*, volume 5074 of *Lecture Notes in Computer Science*, pages 94–108. Springer, 2008.
29. Nick Russell, Wil M. P. van der Aalst, Arthur H. M. ter Hofstede, and David Edmond. Workflow resource patterns: Identification, representation and tool support. In *Advanced Information Systems Engineering, 17th International Conference, CAiSE 2005, Porto, Portugal, June 13-17, 2005, Proceedings*, volume 3520 of *Lecture Notes in Computer Science*, pages 216–232. Springer, 2005.
30. Massimo Strano, Carlos Molina-Jiménez, and Santosh K. Shrivastava. Implementing a rule-based contract compliance checker. In *Software Services for e-Business and e-Society, 9th IFIP WG 6.1 Conference on e-Business, e-Services and e-Society, I3E 2009, Nancy, France, September 23-25, 2009. Proceedings*, volume 305 of *IFIP*, pages 96–111. Springer, 2009.
31. Oktay Türetken, Amal Elgammal, Willem-Jan van den Heuvel, and Mike P. Papazoglou. Enforcing compliance on business processes through the use of patterns. In *19th European Conference on Information Systems, ECIS 2011, Helsinki, Finland, June 9-11, 2011*, page 5, 2011.
32. Barbara Weber, Manfred Reichert, Stefanie Rinderle-Ma, and Werner Wild. Providing Integrated Life Cycle Support in Process-Aware Information Systems. *International Journal of Cooperative Information Systems*, 18(01):115–165, 2009.
33. Mathias Weske. *Business Process Management: Concepts, Languages, Architectures*. Springer, 2007.