Enabling Compliance Monitoring for Process Execution Engines

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. Most of organizations try to ensure that their business processes are compliant with regulations and laws, so runtime monitoring of process compliance is considered to be of crucial importance. 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. 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. Additionally, we implement compliance anti patterns to prove the feasibility of our approach.

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

1 Introduction

Most of the organizations try to ensure the compliance of their business processes against regulations and laws to avoid non-compliance penalties [21]. As a result, different approaches were developed to check the compliance of a business processes through different phases of a business process lifecycle [13,21] e.g., at design time [21], at the process execution (runtime) [3] or at the evaluation phase [18].

Compliance monitoring at process execution time is considered of utmost importance because not every possibility of violation can be checked at design time. Several approaches have been developed to enable the monitoring of running processes compliance, e.g. [14,3]. Most of the monitoring frameworks presume a rich activity lifecycle model, cf. [3,10] compared to the simple task lifecycle models supported by business process 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 propose a mapping approach that fills the gap between the activity lifecycle models supported by different process execution engines and the reference

lifecycle model proposed by Russell et al. [19] and supported by the BP-MaaS monitoring framework [3], as an example monitoring framework, and two process execution engines: Activiti [17] and jBPM [6]. For each, we have studied the supported task lifecycle models of the engines, compared them to the reference lifecycle and identified the mapping. To achieve the compliance monitoring, we implemented the compliance patterns presented in [3].

The rest of the paper is organized as follows: Section 2 presents our proposed model, Section 3 discusses related work and Section 4 concludes the paper with an outlook on future work. The needed preliminaries, some of the background techniques and a simple running example are provided in an extended version of this paper [9].

2 Enabling Monitoring

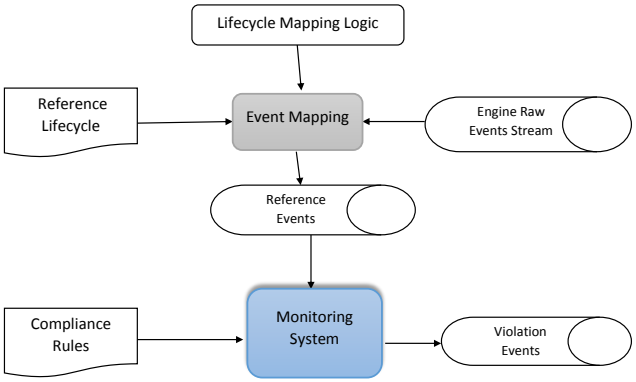

Fig. 1: Proposed monitoring system framework

Our contribution can be seen as a middleware which consists of: a) An engine-specific mapping to the reference task lifecycle b) Implementation of the compliance monitoring anti patterns in [3]. Fig. 1 depicts the architecture of the middleware. The *Lifecycle Mapping Logic* is a set of rules used to perform the mapping from *Engine's raw events* to the *reference events*. 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 Activiti, rules 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.

2.1 Reference Task Lifecycle

The lifecycle in Fig. 2 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 . More details about this lifecycle could be found in the extended version of this paper [9].

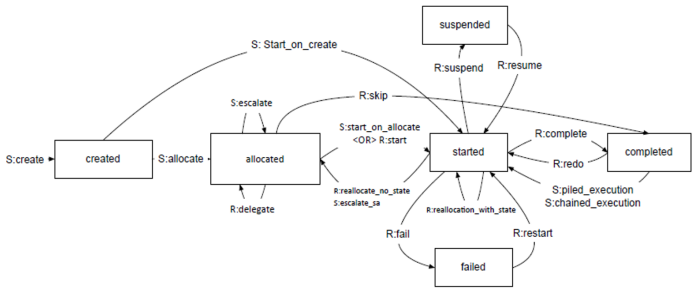

Fig. 2: Reference task lifecycle model adapted from [19]

2.2 Lifecycle Mapping

The lifecycle mapping part has two inputs: (1) *Engine Raw Events Stream* and (2) *Lifecycle Mapping Logic* which contains a list of rules used to obtain the corresponding reference event. The output of this part is the *Reference Event Stream* which contains the mapped and the derived missing reference events.

There are four different possibilities for mapping an engine-specific event/state to the reference state. The first possibility is that there is a *direct correspondence*. 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 activity 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 does not support this state.

We investigated different execution engines' lifecycles for different languages such as BPMN, BEPL and workflow systems (YAWL). A survey about these engines and their lifecycles could be found on the extended version of this paper [9] We report implementation on *jBPM* [6] and *Activiti* [17] as *Camunda* [1] and *Activiti* almost have the same API functions. We investigated the YAWL workflow engine's lifecycle [20] 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 1 summarizes the mapping between the different states of these engines's lifecycle to the reference lifecycle.

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

Table 1: Mapping the lifecycle of different engines to the reference lifecycle

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

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 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.

For *Activiti* and *Camunda* engines:

- **Rule 1:** (*Start_on_Create*) action can be inferred when task t is created with performer r_i and the system directly starts this task. If the start event is not supported by the engine, we consider that the task is immediately started after creation and a new *start_on_create* event is thrown and mapped to *started* reference event.
- **Rule 2:** (*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 reports a wait state for the process instance, we can detect that a *suspended* event occurred and is thrown to the stream.

After the investigation of the Activiti and Camunda engines, we found that the *failed* state is not supported and there is no way of mapping this state.

2.3 Compliance Monitoring Implementation

This subsection provides the implementation details of compliance monitoring within *Activiti* and *jBPM* engines. We use complex event processing (CEP) with the help of *Esper* within *Activiti* which consists of four parts: (1) *Business Process Model* where we define task listeners in the XML file defining the process, (2) *Task Listeners* that detect any change in the task's state through the process execution and generate a predefined event to be analyzed later, (3) *Esper Engine* that is responsible for generating streams and populating them with events coming from different sources, initiating listeners to detect events from the engine, throwing new events in case of any changes after processing and communicating with our middleware (4) *Stream*; that contains all kinds of

[2] Enabled state with more information about the resource

events either raw, mapped or complex events thrown from our middleware. All the logic for mapping events or implementation of compliance anti patterns is implemented using Event Processing Language (EPL) and Drools Rule Language (DRL).

The following lists present a sample EPL query of the implementation of the mapping cases.

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 = ? "
```

jBPM supports the CEP technology through the integration with Drools [2]. The mapping approach on *jBPM* starts by defining an active session to register the streams which contains the raw events generated from the process instances execution. After that Drools rules match the incoming events with the defined mapping logic and fires when a mapping match is found.

The following list introduces a 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]. The framework uses the reference events stream and the compliance patterns as input and matches each pattern with its corresponding anti-pattern rule to detect any possible violation as depicted in Fig. 1.

We implement this part using the Drools fusion component which contains complex event processing features. The anti patterns queries are written using the Drools rule language (DRL) and are 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 streams containing events in Drools engine are called entry points. Based on the compliance rules and the events from the entry points, Drools rules fire and throw a violation event whenever a violation occurs. The following list introduces sample of the rules used to implement the compliance rules anti-patterns.

Listing 1.3: Exist anti pattern rule

```

Exist anti pattern
when Event ( $task : task, $ts : tistamp, $type : type,
```

```
$pi : processinstance) from entry-point Stream
$comprules : Comprules ( pattern == "exist", scopestart == $type,
$mul: multiplicity )
$total: Number( intValue > $mul )
from accumulate ( Event ( task == $comprules.antecedent,
tistamp > $ts, type == $comprules.scopeend, processinstance == $pi,
$ant: task ) from entry-point Stream,count($ant))
then System.out.println("Exist␣pattern␣VIOLATION␣-->
the␣antecedent␣count
is␣GREATER␣THAN␣multiplicity,␣Evaluate␣Loan␣request␣delegation
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. Also more details about the evaluation and results of our approach could be found in this extended version of the paper [9].

3 Related Work

Regarding runtime monitoring frameworks, [1] combines BPM with CEP for monitoring business process execution. They used the activity lifecycle from [23], the refined process structure tree (RPST) [16] to analyze process models, CEP queries and process event monitoring points (PEMPs) to monitor business process execution based on events in semi-automated environments. [8] 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 [4] the authors present a framework for monitoring the progress of task execution and predicting problems 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 specification of runtime constraints and anti-pattern queries notation to detect runtime compliance violations using CEP.

In [11,12], 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. [10] presented a generic framework to monitor process instances from different process perspectives. They are using the lifecycle from [19] 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. [5] 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, [7] discuss increasing the number of observed events by capturing data state transition events in non automated environment. They focus on the object lifecycle states generated from data objects. [15] addresses the problem of modeling processes with complex data dependencies and their enactment from process models. They focus on the process data level and data objects. Meanwhile, [22] 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.

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.

4 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*, 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. Cristina Cabanillas, Claudio Di Ciccio, Jan Mendling, and Anne Baumgrass. Predictive task monitoring for business processes. In *BPM*, volume 8659 of *LNCS*, pages 424–432. Springer, 2014.

5. 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.
6. Simone Fiorini and Arun V Gopalakrishnan. *Mastering jBPM6*. Packt Publishing, 2015.
7. Nico Herzberg and Andreas Meyer. Improving process monitoring and progress prediction with data state transition events. volume 1029, pages 20–23, 2013.
8. Nico Herzberg, Andreas Meyer, and Mathias Weske. An Event Processing Platform for Business Process Management. In *EDOC*, pages 107–116. IEEE Computer Society, 2013.
9. Marwa Hussein, Ahmed Awad, and Osman Hegazy. Enabling compliance monitoring for process execution engines. technical report, Faculty of Computers and Information, Cairo University, 2017. `http://scholar.cu.edu.eg/?q=marwa_hussein/files/paper.pdf`.
10. Amin Jalali and Paul Johannesson. Multi-perspective business process monitoring. In *BPMDS*, volume 147 of *LNBIP*, pages 199–213. Springer, 2013.
11. 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.
12. David Knuplesch, Manfred Reichert, and Akhil Kumar. A framework for visually monitoring business process compliance. *Inf. Syst.*, 64:381–409, 2017.
13. 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.
14. 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 *BPM*, volume 6896 of *LNCS*, pages 132–147. Springer, 2011.
15. 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.
16. Artem Polyvyanyy, Jussi Vanhatalo, and Hagen Völzer. Simplified computation and generalization of the refined process structure tree. In *WS-FM Workshop*, volume 6551 of *LNCS*, pages 25–41. Springer, 2010.
17. Tijs Rademakers. *Activiti in Action*. Manning Publications, 2012.
18. Elham Ramezani, Dirk Fahland, and Wil M. P. van der Aalst. Where did I misbehave? diagnostic information in compliance checking. In *BPM*, volume 7481 of *LNCS*, pages 262–278. Springer, 2012.
19. 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*, volume 3520 of *LNCS*, pages 216–232. Springer, 2005.
20. Arthur H. M. ter Hofstede, Wil M. P. van der Aalst, Michael Adams, and Nick Russell, editors. *Modern Business Process Automation - YAWL and its Support Environment*. Springer, 2010.
21. Oktay Türetken, Amal Elgammal, Willem-Jan van den Heuvel, and Mike P. Papazoglou. Enforcing compliance on business processes through the use of patterns. In *ECIS*, page 5, 2011.
22. 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.
23. Mathias Weske. *Business Process Management: Concepts, Languages, Architectures*. Springer, 2007.