Online Detection of Operator Errors in Cloud Computing Using Anti-patterns

Arthur Vetter(✉)

Horus software GmbH, Ettlingen, Germany
arthur.vetter@horus.biz

Abstract. IT services are subject of several maintenance operations like upgrades, reconfigurations or redeployments. Monitoring those changes is crucial to detect operator errors, which are a main source of service failures. Another challenge, which exacerbates operator errors is the increasing frequency of changes, e.g. because of continuous deployments like often performed in cloud computing. In this paper, we propose a monitoring approach to detect operator errors online in real-time by using complex event processing and anti-patterns. The basis of the monitoring approach is a novel business process modelling method, combining TOSCA and Petri nets. This model is used to derive pattern instances, which are input for a complex event processing engine in order to analyze them against the generated events of the monitored applications.

Keywords: Complex event processing · Anti-pattern · TOSCA · IT service management · Anomaly detection

1 Introduction

Operator errors have been one of the major reasons for IT service failures [1–6] and will probably continue to be regarding current trends like continuous delivery, DevOps and infrastructure-as-code [7], which are especially very common in cloud computing. In recent years, several studies and methods were developed to detect errors in very complex IT systems [8]. Those traditional methods are suited for detecting errors during "normal" operations, but not during change operations like reconfigurations or rolling upgrades, when one node after the other is upgraded [9]. The reason for the amount of operator errors is their human nature, because change operations are either performed or initiated by human operators.

This paper presents current research results of a novel monitoring approach for those change operations. The monitoring approach is based on a process model, combining TOSCA (Topology and Orchestration Specification for Cloud Applications) and high-level Petri nets [8], which explicitly models the maintenance operations of the IT service applications. This process model is used to derive pattern instances from it. Those pattern instances are checked through

P. Ceravolo et al. (Eds.): SIMPDA 2017, LNBIP 340, pp. 1–24, 2019.
https://doi.org/10.1007/978-3-030-11638-5_1

a complex event processing engine against state events and transaction events. State events describe the state of the application, whereas transaction events describe each single operation performed on the application. Therefore, the logs of the applications are filtered for meaningful transaction events and are sent to the complex event processing engine, allowing the detection of operator errors almost in real-time. The complex event processing engine compares the pattern instances with the generated events through anti-patterns and creates an error message, when an anti-pattern instance was detected. Figure 1 gives an overview of the general monitoring approach.

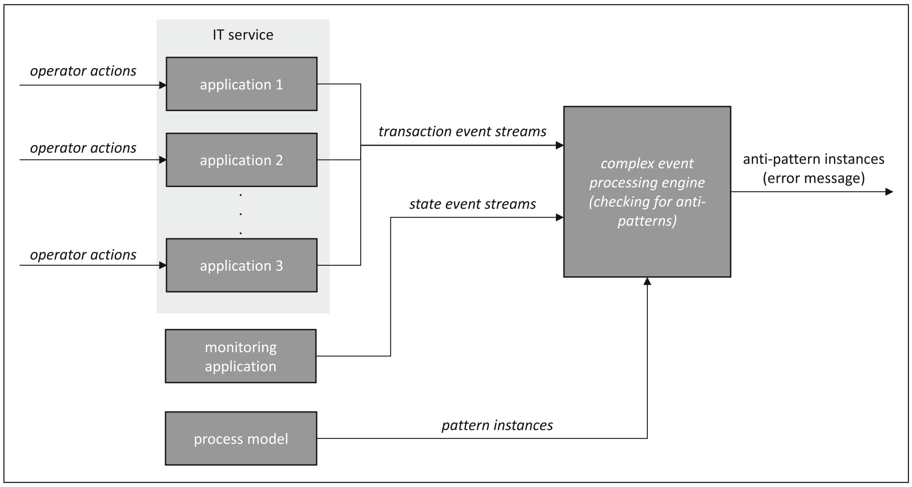

Fig. 1. General monitoring approach

The remainder of this paper is organized as follows: The next section gives a short overview of typical operator errors. Section 3 describes the fundamentals of TOSCA and XML nets, which are used to model the actual maintenance. Section 4 describes the concept of patterns and anti-patterns. Section 5 presents the proof of concept implementation. In the next section first experimental results are presented and discussed in the following section. Afterwards related work is presented. Section 9 concludes the paper.

2 Operator Errors

Oppenheimer et al. [5] and many other authors like [4,9] classify operator errors in process errors and configuration errors. Process errors can be further differentiated in following errors: forgotten activity, an unneeded activity was executed, a wrong activity was executed or actual correct activities were executed in the wrong order. Configuration errors can be separated in formatting errors and configuration value errors [13]. Formatting errors can be further separated in lexical errors, syntactical errors and typos. Configuration value errors can be further classified in local value inconsistencies and global environment inconsistencies. A monitoring approach to detect operator errors should be able to detect all those process and configuration error types. Table 1 gives an example for every type of operator error and a reference to a study with further information and examples.

Table 1. Operator error examples

Operator error	Example	Description	Reference
Forgotten activity	Forgot to restart a server		[4]
Unneeded activity	Unnecessary restart of a server		[9]
Wrongly executed activity	Restoration of a wrong backup		[4]
Wrong order	Bringing down two servers in parallel for configuration instead of sequentially maintaining the servers		[9]

Local Inconsistency	log_output = "Table"
	log = query.log
		According to the value "log", the user wanted to store logs in a file, but the value "log.output" controls to store data in a database table	[10]

Global Inconsistency	datadir = /some/old/path	"datadir" points to an old path, which does not exist anymore.	[10]
Lexical Errors	InitiatorName: iqn:DEV_domain	Only lowercase letters are allowed ("DEV")	[10]

Syntactical Errors	extension = mysql.so
	.....
	extension = recode.so
		"mysql.so" depends on "recode.so" and was configured in the wrong order	[10]

Typo	extension = recdoe.so
	extension = mysql.so
		The correct writing of "recdoe.so" is "recode.so"	[10]

3 Fundamentals

The process model is a combination of TOSCA and XML nets and was introduced in a former paper [8]. In this chapter, we describe the fundamentals of TOSCA and XML nets shortly and then describe how maintenance operations can be modelled with TOSCA and XML nets.

3.1 TOSCA

TOSCA is a standard, released by OASIS [14] to support the portability of cloud applications between different cloud providers and the automation of cloud application provisioning. Therefore, TOSCA provides a modelling language to describe cloud applications as Service Templates. A Service Template consists of a Topology Template and of optional Plans (see Fig. 2).

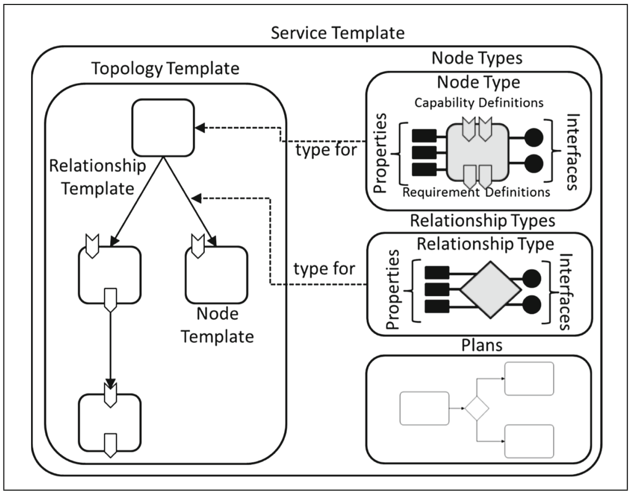

Fig. 2. TOSCA service template

A Topology Template describes the structure of a cloud application as a directed graph and consists of Node Templates and Relationship Templates. A Node Template represents a component of the cloud application, e.g. an application server and is described by a Node Type. A Node Type defines

– properties of the component (*Properties Definition*),
– available operations to manipulate the component (*Interfaces*),
– requirements of the component (*Requirement Definitions*),
– possible lifecycle states of the component (*Instance States*) and
– capabilities it offers to satisfy other components' requirements (*Capability Definitions*).

Plans are models to orchestrate the management Operations, which are offered by the cloud application components and can be written in BPMN, BPEL or other languages.

As TOSCA Service Templates are written as XML documents, we decided to use the notation of XML nets for the creation of Plans, which we name "maintenance plan" in the rest of the paper. Using XML nets has the advantage that no additional notation elements have to be defined like it is the case e.g. for BPMN [25]. Apart of that, XML nets allow to describe detailed manipulations of XML documents, which are used to model configuration operations in maintenance plans.

3.2 XML Nets

XML nets [15] are a high-level variant of Petri nets, in which places represent containers for XML documents. The XML documents must conform to the XML

Schema, which is assigned to a specific place. Edges are labeled with Filter Schemas, which are used to read or manipulate XML documents. Transitions can be inscribed by a logical expression, whose variables are contained in the adjacent edges. A transition in an XML net is enabled and can be fired for a given marking, when the following three conditions hold. First, every place in the pre-set of the transition holds at least one valid XML document, which conforms to the Filter Schema inscribing the edge to the transition. Second, every place in the post-set of a transition must contain one valid XML document, if the XML document has to be modified. If an XML document has to be created from scratch the place must not already contain this XML document. Third, for the given instantiation of the variables, the transition inscription has to be evaluated to true in order to enable the transition. If an enabled transition is fired, XML documents in the pre-set places are (partially) deleted or read for the given instantiation of variables, and new XML documents are created or existing XML documents are modified in the post-set places of the transition.

3.3 Modelling Maintenance Plans

This section describes the modelling of maintenance plans with TOSCA and XML nets, which allows to model applications and the orchestration of applications' management operations in one integrated model. Such a model can then be used to derive pattern instances. Therefore, we extend our former approach, introduced in [8]. The following adjustments are made to the general definition of TOSCA Node Templates:

- A Node Template represents exactly one instance of an application, that means the attributes *minIstances, maxInstances := 1.*
- Node Templates are extended with the complex element *InstanceState*, which stores the current state of the corresponding application.

The notation of XML nets is adjusted as follows:

- Places are containers for *Service Templates.* Every place is assigned to the general TOSCA XML schema and additionally to a single *Node Type*, which restricts the allowed filter schemas for corresponding *Node Templates.*
- Transitions represent operations, defined in *Interfaces* of the adjacent *Node Types.*
- Filter Schemas can either be used to select *Node Templates* or to modify *Properties*, or *Instance States* of a *Node Template.* Deleting whole *Node Templates* is in contrast to general XML nets not allowed. *Node Templates* can only change their status, e.g. to undeploy, but they cannot be deleted. The reason is, that for error detection purposes, even an undeployed Node has to be monitored to be sure it was really undeployed and e.g. has not been deployed by accident afterwards again. Deleting parts of a *Node Template* are allowed, e.g. deleting a property.
- Transitions hold the attributes *start* and *end*, which define when the operation has to be executed earliest and latest.

We define a maintenance plan as a tuple $\mathrm{MP} = <\mathrm{P}, \mathrm{T}, \mathrm{A}, \Psi, \mathrm{I_P}, \mathrm{I_N}, \mathrm{I_A}, \mathrm{I_T}, \mathrm{M_0}>$, where

(i) $<\mathrm{P}, \mathrm{T}, \mathrm{A}>$ is a Petri net with a set of places P, a set T of transitions, and a set A of edges connecting places and transitions (the definition and description of petri nets is excluded in this paper, but can be found, e.g., in [11]).

(ii) $\Psi = <\mathrm{D}, \mathrm{FT}, \mathrm{PR}>$ is a structure consisting of a finite and non-empty individual set D, a set of term and formula functions *FT* defined on *D*, and a set of predicates *PR* defined on *D*.

(iii) $\mathrm{I_P}$ is the function that assigns the TOSCA XML Schema to each place.

(iv) $\mathrm{I_N}$ is the function that assigns additionally a Node Type to each place.

(v) $\mathrm{I_A}$ is the function that assigns a Filter Schema to each edge. The Filter Schema must conform to the XML Schema and Node Type of the adjacent place.

(vi) $\mathrm{I_T}$ is the function that assigns a predicate logical expression as inscription to each transition. The inscription is built on a given structure Ψ and a set of variables. Only variables, which are contained in the Filter schemas of adjacent arcs, are allowed. The inscription must evaluate to true in order to enable the transition.

(vii) Each transition represents a value of the element *operation*, which is defined in the complex element *Interfaces* of the Node Type in the postset of the transition.

(viii) $\mathrm{M_0}$ is the initial marking. Markings are TOSCA Service Templates.

(ix) Each transition holds the attributes *start* and *end*.

Figure 3 shows an example of a maintenance plan to configure the database connection of the application *MyApplication* (Filter Schemas are written informally for readability reasons). It is assumed that the database and application are part of the Service Template *MyService*. *MyApplication* is hosted on *MyAppServer* and requires additionally the database *TestDatabase* (NT1). It is assumed, that when the change is performed, *MyApplication* is started. In the first place, which is linked to a Node Type *Application*, *MyApplication* is one possible representation. The first Filter Schema *FS1* selects *MyApplication*. Before *MyApplication* can be configured it has to be stopped, which is represented in the first transition. The condition in order to stop the application is, that *MyApplication* has to be started. Stopping is one possible operation, which is given by the Node Type *Application*. If at the beginning of executing the change, *MyApplication* is already stopped instead of started, it is a hint, that an incident or something unexpected happened, so the change execution should be interrupted. When *MyApplication* is stopped, the database connection can be set. Therefore, the Node Template *TestDatabase* is selected and the database connection is built up on the properties of *TestDatabase* and inserted in *MyApplication* through the Filter Schema *FS5*. Afterwards *MyApplication* can be started again, but only if *TestDatabase* is running (inscription assigned to transition *Start*).

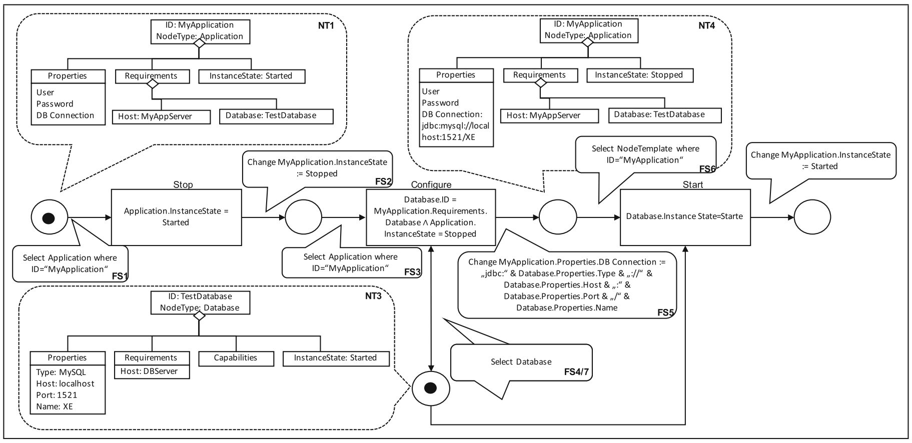
ID: MyApplication
NodeType: Application
NT1
Properties
User
Password
DB Connection
Requirements
InstanceState: Started
Host: MyAppServer
Database: TestDatabase
ID: MyApplication
NodeType: Application
NT4
Properties
User
Password
DB Connection:
jdbc:mysql://local
host:1521/XE
Requirements
InstanceState: Stopped
Host: MyAppServer
Database: TestDatabase
Select NodeTemplate where ID="MyApplication" FS6
Change MyApplication.InstanceState := Started
Change MyApplication.InstanceState := Stopped FS2
Stop
Application.InstanceState = Started
Configure
Database.ID = MyApplication.Requirements. Database ∧ Application. InstanceState = Stopped
Start
Database.Instance State=Starte
Select Application where ID="MyApplication" FS1
Select Application where ID="MyApplication" FS3
Change MyApplication.Properties.DB Connection := „jdbc:" & Database.Properties.Type & „://" & Database.Properties.Host & „:" & Database.Properties.Port & „/" & Database.Properties.Name FS5
ID: TestDatabase
NodeType: Database
NT3
Properties
Type: MySQL
Host: localhost
Port: 1521
Name: XE
Requirements
Host: DBServer
Capabilities
InstanceState: Started
Select Database FS4/7

Fig. 3. Example of a TOSCA based XML net

4 Pattern and Anti-pattern for Operator Error Detection

In computer science the term pattern is popular since the publication of the book about design patterns from Gamma et al. [12]. In this book, Gamma et al. describe patterns as solutions for recurring problems in a specific context. Aalst et al. [13] used the concept of patterns for business process modelling and described several patterns for the control flow perspective. Since then, many patterns were described for different perspectives of business process modelling, like for the data perspective [14,15]. Riehle and Züllighoven define a pattern more general as an abstraction of a recurring concrete form in a specific context [16]. A form is a finite number of distinguishable elements and their relationships [16]. A context restricts the possible usage of a form, because the form has to fit into this specific context. Based on the definition of Riehle and Züllighoven we define a pattern and anti-pattern as following:

Definition 4.1 (Pattern). A pattern is an abstraction of a welcomed, recurring, concrete form in a specific context.

Definition 4.2 (Anti-pattern). An anti-pattern is as an abstraction of an unwelcomed, concrete form in a specific context.

In our work, we use patterns to describe the planned to be control flow, application configurations and application states for the scheduled maintenance. So, patterns are used during the design phase. Anti-patterns are used to check during the actual execution of the maintenance (run-time), if a form of events exists, which does not fit to the planned forms. In the following we restrict and formalize the context of the used patterns and anti-patterns as well as the form of these patterns and anti-patterns.

4.1 Context

As described in Sect. 3, the monitoring approach is based on the comparison between produced events of monitored applications and pattern instances of the TOSCA management plan. Those parameters build the context of the patterns. We separate two kinds of events in our context: *state events* and *transaction events*. Definitions 4.3 and 4.4 formalize *state events* and *transaction events* in this paper.

Definition 4.3 (State Event). A state event is a tuple se = (timestamp, app, state), where:

– *timestamp* is the timestamp of the event creation.
– *app* is the *Node Template id* of the monitored application.
– *state* is the actual state of the application. Only values are allowed, which are defined in the Node Type of the application by the element *Instance States*.

The set of all state events is defined as *SES*.

Definition 4.4 (Transaction Event). A transaction event is a tuple *te = (timestamp, st, app, op, prop, value)*, where:

– *timestamp* is the timestamp of the event creation.
– *st* is the *Service Template id*, which identifies the service the application belongs to.
– *app* is the Node Template id of the monitored application.
– *op* describes the operation, which was conducted on the application. The value of *op* must correspond to one of the values, which are defined in the element *operation* of the *Node Type* of the application.
– *prop* describes the property, which was changed when the operation was executed. If no property was changed during the operation *prop* is null.
– *value* is the value of the property, which was changed. If *prop* is null, *value* also has to be null.

The set of all transaction events is defined as *TES*. State events and transaction events represent the actual events during a maintenance. The corresponding "to-be" events are conditions and activities, which can be derived from a TOSCA management plan. A condition represents a possible transition inscription, whereas activities represent firing sequences.

Definition 4.5 (Condition). A condition is a tuple *(app, op, prop, zapp, state)*, where:

– *app* is the id of the Node Template, on which the operation is performed.
– *op* is the operation, which is performed on the Node Template and is restricted in the Node Type of the Node Template.
– *prop* is the property of the Node Template, which is changed during the operation.

– *zapp* is the id of the Node Template, which has to be in a specific state in order to perform the operation.
– *state* describes in which state *zapp* has to be.

Let *SM* be the set of all maintenance plans. The set of all conditions of a maintenance plan is defined as $\mathrm{SC_i}$, i ∈ SM. The set of all transition inscriptions of a maintenance plan is defined as $\mathrm{STI_i}$ i ∈ SM. The function F: $\mathrm{SC_i} \rightarrow \mathrm{STI_i}$ assigns a transition to each condition.

Definition 4.6 (Activity). An activity is a tuple *a = (st, app, op, prop, value, start, end)*, where:

– *st* is the *Service Template id*, which identifies the service template in the TOSCA management plan.
– *app* is the id of the Node Template, on which the operation is performed.
– *op* is the operation, which is performed on the Node Template and is restricted in the Node Type of the Node Template.
– *prop* is the property of the Node Template, which is changed during the operation.
– *value* is the value of the property, which was changes. If prop is null, value also has to be null.
– *start* describes when the activity has to start earliest.
– *end* describes when the activity has to end latest.

Be *SM* the set of all maintenance plans. The set of all activities of a maintenance plan is defined as $\mathrm{SA_i}$, i ∈ SM. The set of all transitions of a maintenance plan is defined as $\mathrm{ST_i}$, ∈ SM. The function F: $\mathrm{SA_i} \rightarrow \mathrm{ST_i}$ assigns a transition to each activity.

Additionally, for some anti-patterns we need the history of transaction events and the latest state of an application called the state event history.

Definition 4.7 (Transaction Event History). A *transaction event history* is a selection on the set of transaction events, which are in the time scope of the scheduled maintenance:
$\mathrm{TEH} := \sigma_{\mathrm{timestamp} \geq \mathrm{maintenance\_start}\ \wedge\ \mathrm{timestamp} \leq \mathrm{maintenance\_end}} \mathrm{TES}$

Definition 4.8 (State Event History). The *state event history* SEH stores the latest state for each application in *SES*.

Furthermore, we define three functions, time, countTE and countA.

Definition 4.9 (Time). time is a function, which returns the current timestamp.

Definition 4.10 (CountTE). *countTE(te,TEH)* is a function, which counts the number of occurrences of the transaction event *te* in the transaction event history.

Definition 4.11 (CountA). *countA(a, S)* is a function, which counts the number of occurrences of an activity *a* in a set S.

After the description and definition of the context, the patterns and anti-patterns are described.

4.2 Pattern and Anti-pattern

All in all, we define ten patterns/anti-patterns in order to detect operation errors. These are NEXT, IMMEDIATELY NEXT, PRECEDENCE, IMMEDIATELY PRECEDENCE, OCCURRENCE, ALTERNATIVE OCCURRENCE, ABSENCE, ALTERNATIVE ABSENCE, VALUE and STATE-CONDITION. The first eight patterns are highly influenced by the specification pattern of Dwyer et al. [17] and are used to detect process errors. Whereas the VALUE anti-pattern is used to detect configuration errors. The STATE-CONDITION anti-pattern is used to check, if a resource is in the planned state in order to perform a task on it. To describe the patterns and anti-patterns following template is used:

- **Name:** The name of the pattern must be unique and should describe the purpose of the pattern.
- **Description:** Here the form of the pattern is described, which should occur in the maintenance.
- **Instances:** Here it is described, how instances of the pattern can be derived from the maintenance plan.
- **Example:** Here, examples of pattern instances are given.
- **Anti-pattern:** A description of the corresponding anti-pattern and which type of operator errors can be detected with the anti-pattern. Additionally, we formalize the conditions, which have to be violated in order to detect an operator error.
- **Similar pattern:** Here, similar patterns are referenced and differences are named.

Pattern NEXT
Description: This pattern describes pairs of activities, defining which activity has to occur after another (with possible activities inbetween). The pattern is used for controlling AND-joins, AND-splits and concurrent sequences in a maintenance plan.
Instances: To get all instances of this pattern for a TOSCA management plan i we create a relation $\mathrm{P1} := \mathrm{AM_i}$ x $\mathrm{AM_i}$ x $\mathrm{AM_i}$ with the tuples $(\mathrm{a_{cur}}, \mathrm{a_{nex}}, \mathrm{a_{far}})$ where,

- the corresponding transitions $\mathrm{t_{cur}}$ and $\mathrm{t_{nex}}$ of the activities $\mathrm{a_{cur}}$ and $\mathrm{a_{nex}}$ are connected through the same place,
- $\mathrm{t_{cur}}$, $\mathrm{t_{nex}}$ and $\mathrm{t_{far}}$ have to occur in the same path,
- $\mathrm{t_{far}}$ always has to occur after $\mathrm{t_{cur}}$,
- $\mathrm{t_{far}}$ and $\mathrm{t_{cur}}$ may not be connected through the same place.

Example: In Fig. 4 instances of the pattern NEXT are (a1, a2, a4), (a1, a2, a6), (a1, a3, a5), (a1, a3, a7), (a2, a4, a6) and (a3, a5, a7).

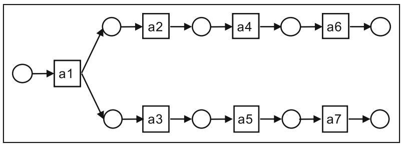

Fig. 4. Example pattern NEXT

Anti-pattern: The anti-pattern allows to detect operator errors of the type "wrong order". Besides, it is possible to detect operator errors of the type syntactical error, if a configuration parameter was changed in the wrong order. An error message is created, when a transaction event $\mathrm{te_{cur}}$ in the event stream occurs and none of the next events $\mathrm{te_{nex}}$ conforms to the next activity $\mathrm{a_{nex}}$. However, one of the next events conforms to an activity $\mathrm{a_{far}}$:

$$\pi_{\mathrm{app,op,prop}}\mathrm{te_{cur}} \in \pi_{\mathrm{a_{cur}.app,a_{cur}.op,a_{cur}.prop}}\mathrm{P1_i} \succ \pi_{\mathrm{app,op,prop}}\ \mathrm{te_{nex}} \in$$
$$\pi_{\mathrm{app,op,prop}}\ (\pi_{\mathrm{a_{far}}}(\sigma_{\mathrm{a_{cur}.app=te_{cur}.app} \wedge \mathrm{a_{cur}.op=te_{cur}.op} \wedge \mathrm{a_{cur}.prop={}_{cur}.prop}}\mathrm{P1_i})) \wedge$$
$$\pi_{\mathrm{app,op,prop}}(\pi_{\mathrm{a_{nex}}}\ ($$
$$\sigma_{\mathrm{a_{cur}.app=te_{cur}.app} \wedge \mathrm{cur.op=te_{cur}.op} \wedge \mathrm{a_{cur}.prop=te_{cur}.prop} \wedge}$$
$$_{\mathrm{a_{far}.app=te_{nex}.app} \wedge \mathrm{a_{far}.op=te_{nex}.op} \wedge \mathrm{a_{far}.prop=te_{nex}.prop}}\mathrm{P1_i})) \notin$$
$$\pi_{\mathrm{app,op,prop}}(\sigma_{\mathrm{te_{cur}.timestamp} > \mathrm{timestamp} \wedge \mathrm{te_{nex}.timestamp} < \mathrm{timestamp}}\mathrm{TEH})$$

Similar Pattern: The pattern IMMEDIATELY NEXT allows also to detect operator errors of the type "wrong order". However, the pattern IMMEDIATELY NEXT would create wrong error messages for concurrent sequences and can only be used for non-concurrent activities.

Pattern IMMEDIATELY NEXT
Description: This pattern describes pairs of activities, defining which activity has to be executed right after another activity (without any other activities occurring inbetween). The pattern is used for controlling XOR-joins, XOR-splits and non-concurrent sequences in a maintenance plan.
Instances: To get all instances of this pattern for a maintenance plan i we create a relation $\mathrm{P2_i} := \mathrm{AM_i}$ x $\mathrm{AM_i}$ with the tuples ($\mathrm{a_{cur}}$, $\mathrm{a_{nex}}$) where,

– the corresponding transitions of the activities $\mathrm{a_{cur}}$ and $\mathrm{ta_{nex}}$ are connected through the same place, and
– the corresponding transitions cannot be executed concurrent to other transitions.

Example: In Fig. 5 instances of the pattern IMMEDIATELY NEXT are (a1, a2), (a1, a3), (a2, a4), (a3, a5), (a4, a6) and (a5, a7).

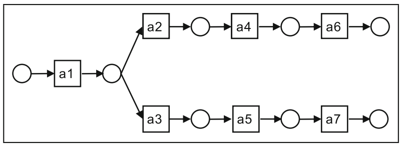

Fig. 5. Example pattern IMMEDIATELY NEXT

Anti-pattern: The anti-pattern allows to detect operator errors of the type "wrong order" for non-concurrent operations. An error message is created when a transaction event te_{cur} occurs in the event stream and the next following transaction event te_{cur+1} of the same service template does not correspond to the expected activity:

$$\pi_{app,op,prop} te_{cur} \in \pi_{a_{cur}.app,a_{cur}.op,a_{cur}.prop} P2_i \succ \pi_{app,op,prop}(\sigma_{st=te_{cur}.st} te_{cur+1})$$
$$\notin \pi_{app,op,prop}(\pi_{a_{nex}}(\sigma_{a_{cur}.app=te_{cur}.app \wedge a_{cur}.op=te_{cur}.op \wedge a_{cur}.prop=te_{cur}.prop} P2_i))$$

Similar Pattern: The pattern IMMEDIATELY NEXT is similar to the pattern NEXT. The difference is, that in the IMMEDIATELY NEXT pattern in contrast to the pattern NEXT no activities of the same service template are allowed between a pair of activities.

Pattern PRECEDENCE
Description: This pattern describes pairs of activities where one activity has to occur before another one. Like the pattern NEXT it is allowed that other activities occur inbetween the activities of such a pair of activities. The pattern is used for controlling AND-joins, AND-splits and concurrent sequences in a maintenance plan.
Instances: To get all instances of the pattern a relation $P3_i := AM_i \text{ x } AM_i$ with the tuples (a_{cur}, a_{pre}) is created where,

- the corresponding transitions of the activities a_{cur} and a_{pre} are connected through the same place, and
- the corresponding transitions can be executed concurrent to other transitions.

Example: In Fig. 4 instances of the pattern PRECEDENCE are (a2, a1), (a3, a1), (a4, a2), (a5, a3), (a6, a4) and (a7, a5).
Anti-pattern: With the anti-pattern it is possible to detect operator errors of the type "wrong order". An error message is created when a transaction event te occurs in the event stream which corresponds to an activity a_{cur}, but the corresponding transaction event for the activity a_{pre} does not exist in the transaction event history:

$$\pi_{app,op,prop} te \in \pi_{a_{cur}.app,a_{cur}.op,a_{cur}.prop} P3_i$$
$$\wedge\, \pi_{app,op,prop}(\pi_{a_{pre}}(\sigma_{a_{cur}.app=te.app \wedge a_{cur}.op=te.op \wedge a_{cur}.prop=te.prop} P3_i))$$
$$\notin \pi_{app,op,prop} TEH$$

Similar Pattern: The pattern IMMEDIATELY PRECEDENCE is also used to detect forgotten activities, which have to be executed before another activity. For the pattern IMMEDIATELY PRECEDENCE no activities are allowed between $\mathrm{a_{cur}}$ and $\mathrm{a_{pre}}$, whereas for the pattern PRECEDENCE additional activities in between are allowed. Besides, the pattern is similar to the pattern NEXT. The difference is, that the pattern NEXT checks for future activities, whereas the pattern PRECEDENCE checks for activities happened in the past of a maintenance execution.

Pattern IMMEDIATELY PRECEDENCE
Description: This pattern describes which activity has to be executed immediately before another one. It can be used for non-concurrent activities as well as for XOR-joins and XOR-splits.
Instances: To get all instances of this pattern for a maintenance plan i we create a relation $\mathrm{P4_i := AM_i \text{ x } AM_i}$ with the tuples $(\mathrm{a_{cur}}, \mathrm{a_{pre}})$, where

- the corresponding transitions of the activities $\mathrm{a_{cur}}$ and $\mathrm{a_{pre}}$ are connected through the same place, and
- the corresponding transitions cannot be executed concurrent to other transitions.

Example: In Fig. 5 instances of the pattern IMMEDIATELY PRECEDENCE are (a2, a1), (a3, a1), (a4, a2), (a5, a3), (a6, a4) and (a7, a5).
Anti-pattern: With the anti-pattern it is possible to detect operator errors of the type "wrong order". An error message is created when a transaction event te occurs in the event stream which corresponds to an activity $\mathrm{a_{cur}}$, but the latest transaction event of the same service template in the transaction event history does not correspond to $\mathrm{a_{pre}}$:

$$\begin{aligned}
&\pi_{\mathrm{app,op,prop}}\mathrm{te} \in \pi_{\mathrm{a_{cur}.app,a_{cur}.op,a_{cur}.prop}}\mathrm{P4_i}\\
\wedge\ &\pi_{\mathrm{app,op,prop}}(\pi_{\mathrm{a_{pre}}}(\sigma_{\mathrm{a_{cur}.app=te.app}\wedge\mathrm{a_{cur}.op=te.op}\wedge\mathrm{a_{cur}.prop=te.prop}}\mathrm{P4_i}))\\
&\notin \pi_{\mathrm{app,op,prop}}(\sigma_{\mathrm{max(timestamp)}}(\sigma_{\mathrm{timestamp<te.timestamp}\wedge\mathrm{st=te.st}}))\mathrm{TEH}
\end{aligned}$$

Similar Pattern: The pattern PRECEDENCE describes also activities which have to occur before another activity. In contrast to the pattern IMMEDIATELY PRECEDENCE the pattern PRECEDENCE allows other activities of the same service template to occur between a pair of activities.

Pattern STATE-CONDITION
Description: This pattern describes the state an application should have in order to be able to perform an operation on either the same or another application. Example: in order to shut down an application server, the database server must be in the state offline.
Instances: Instances of this pattern are all conditions $\mathrm{SC_i}$ for a maintenance plan i.
Example: In Fig. 6, which is a snippet of Fig. 3, the instance of the pattern STATE-CONDITION is (MyApplication, start, NULL, TestDatabase, started).

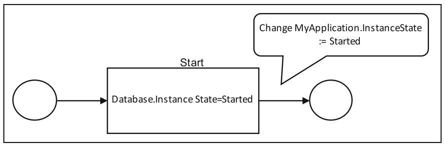
Change MyApplication.InstanceState := Started
Start
Database.Instance State=Started

Fig. 6. Example pattern STATE-CONDITION

Anti-pattern: This anti-pattern does actually not detect an error like described in Sect. 2. Instead, it detects malicious prerequisites, which would lead to an operation error. This is done by comparing the latest state of an application with the planned state:

$$\pi_{\mathrm{app,op,prop}}\mathrm{te} \in \pi_{\mathrm{app,op,prop}}\mathrm{SC_i}$$
$$\wedge\ (\pi_{\mathrm{zapp,state}}(\sigma_{\mathrm{app=te.app\wedge op=te.op\wedge prop=te.prop}}\mathrm{SC_i})/\pi_{\mathrm{app,state}}\mathrm{SEH}) \neq \emptyset$$

Similar Pattern: There are no similar patterns for the STATE-CONDITION pattern.

Pattern VALUE
Description: This pattern describes the value of a configuration parameter which has to be changed during the maintenance.
Instances: To get all instances of this pattern a selection on the set of all activities of the maintenance plan is performed in order to get only those activities which include a change of a property: $\mathrm{P5} := \pi_{\mathrm{app,op,prop,value}}\ (\sigma_{\mathrm{prop\neq NULL}}\mathrm{SA_i})$.
Example: In Fig. 3 the only instance of this pattern is (MyApplication, configure, DB Connection, jdbc:mysql://localhost:1521/XE).
Anti-pattern: This anti-pattern allows to detect operator errors of the types "wrongly executed activity", "lexical error", "local inconsistency","global inconsistency" and "typo" by checking the element value of a transaction event te:

$$\pi_{\mathrm{app,op,prop}}\mathrm{ve} \in \pi_{\mathrm{app,op,prop}}\mathrm{P5} \wedge\ \pi_{\mathrm{app,op,prop,value}}\mathrm{ve} \notin \mathrm{P5_i}$$

Similar Pattern: This pattern can be seen as a more detailed version of the OCCURRENCE pattern. However, the OCCURRENCE pattern just checks for executed operations and properties, but not for the actual values of modified properties.

Pattern OCCURRENCE
Description: This pattern includes all activities which have to be executed in a maintenance plan independent of the chosen path through the maintenance plan.
Instances: To get all instances of this pattern a set $\mathrm{P6_i}$ with all activities of the maintenance plan i is created, where

– every activity has to be executed independent of the chosen path in the maintenance plan.

Example: In Fig. 3 the instances of this pattern are (MyService, MyApplication, stop, NULL, NULL), (MyService, MyApplication, configure, DB Connection, jdbc:mysql://localhost:1521/XE) and (MyService, MyApplication, start, NULL, NULL).

Anti-pattern: The anti-pattern allows to detect errors of the kind "forgotten activity". An error message is created, when an activity was executed too seldom:

$$\exists\ a \in \sigma_{\text{time} > \text{a.end}} P6_i \wedge \text{countTE}(\pi_{\text{app,op,prop}} a, \\ \pi_{\text{app,op,prop}}(\pi_{\text{timestamp} \geq \text{a.start} \wedge \text{timestamp} \leq \text{a.ende}} \text{TEH})) \\ < \text{countA}(\pi_{\text{app,op,prop}} a, \pi_{\text{app,op,prop}}(\pi_{\text{start} \geq \text{a.start} \wedge \text{end} \leq \text{a.end}} P6_i))$$

Similar Pattern: With the anti-patterns NEXT, IMMEDIATELY NEXT, PRECEDENCE and IMMEDIATELY PRECEDENCE it is also possible to detect forgotten activities in a limited way. However it is only possible to detect a forgotten activity right before or after another activity. As an example lets assume we have a sequence (a, b, c, d, e). If the activity a and e occur, it is possible to detect the forgotten activities b and d with the similar patterns, but not the activity c. Only with the anti-pattern OCCURRENCE it is possible to detect the forgotten activity c.

Pattern ALTERNATIVE OCCURRENCE

Description: This pattern describes a pair of activities which cannot be executed together, like after XOR-Splits. However, one activity of such a pair of activities has to be performed during the maintenance.

Instances: To get all instances of this pattern a relation $P7_i := AM_i \times AM_i$ with the tuples (a_{cur}, a_{alt}), where

- the corresponding transitions of the activities a_{cur} and a_{alt} do not occur together in any path of the maintenance plan.

Example: In Fig. 5 instances of this pattern are (a2, a3), (a2, a5), (a2, a7), (a4, a3), (a4, a5), (a4, a7), (a6, a3), (a6, a5), (a6, a7), (a3, a2), (a3, a4), (a3, a6), (a5, a2), (a5, a4), (a5, a6), (a7, a2), (a7, a4) and (a7, a6).

Anti-pattern[1]**:** The anti-pattern allows to detect errors of the kind "forgotten activity". An error message is created, when activities of $P6_i$ were not executed. However, no error message is created for activities, if one alternative activity was already performed. The anti-pattern assumes, that the first executed alternative activity is the right one and therefore ignores all other activities, which may not be executed in conjunction with this first alternative activity.

Similar Pattern: The pattern OCCURRENCE does also detect forgotten activities, but it would create wrong error messages for exclusive activities, if already one of the exclusive activities was executed.

Pattern ABSENCE

Description: This pattern describes which activities may not occur during a maintenance.

[1] Due to space limitations we forgo the formal definition of the following anti-patterns.

Instances: The instances of this pattern are all possible activities, which could really occur during a maintenance, without all activities, which are also modelled in the maintenance plan. Note that in a maintenance plan only a subset of possible operations on service templates is modelled and therefore planned. All other operations should not occur during the maintenance. The number of instances of this pattern can get very high, because the number of possible operations, especially configurations can be huge. However, for the anti-pattern of ABSENCE the generation of pattern instances of the type ABSENCE is not needed as it is explained in the following.
Example: On the assumption that in Fig. 3 other possible operations of *MyApplication* would be "deploy" and "undeploy", some of the instances of the pattern ABSENCE would be (MyService, MyApplication, deploy, NULL, NULL) and (MyService, MyApplication, undeploy, NULL, NULL).
Anti-pattern: The anti-pattern detects errors of the kind "unneeded activity". An error message is created either when

– a transaction event does not correspond to one of the activities in $P6_i$ or $P7_i$, or
– a transaction event corresponds to one the of the activities in $P6_i$ or $P7_i$, but it did not occur during the planned maintenance window, or
– a transaction event corresponds to one the of the activities in $P6_i$ or $P7_i$ and it occurred during the planned maintenance plan, but it occurred too often during the maintenance window.

Similar Pattern: With the anti-patterns NEXT, IMMEDIATELY NEXT, PRECEDENCE and IMMEDIATELY PRECEDENCE it is also possible to detect unneeded activities, when the following or precedence activity was not the planned one. However, these anti-patterns do not know, if the unneeded activity is an activity which was just executed in the wrong order or if it is an activity which should not occur at all.

Pattern ALTERNATIVE ABSENCE
Description: This pattern describes activities which are not allowed to be executed depending on other activities. Such activities occur after XOR-Splits.
Instances: Instances of this pattern are the same like for the pattern ALTERNATIVE OCCURRENCE. Although the pattern instances are the same, the anti-pattern is different to the anti-pattern of ALTERNATIVE OCCURRENCE.
Example: For an example please see the examples of the pattern ALTERNATIVE OCCURRENCE.
Anti-pattern: An error message is created when one of the following conditions hold:

– A transaction event corresponds to an activity in $P7_i$. It is the first alternative activity and it occurred during the maintenance window, however it was performed too often.
– A transaction event corresponds to an activity in $P7_i$ and it occurred during the maintenance window, but an alternative activity was already performed before.

Similar Pattern: The pattern ABSENCE is similar, but the pattern does not check the absence of activities dependent of other activities.

4.3 Derivation of Pattern Instances

Pattern instances can be derived from the maintenance plan by simulating it. Therefore, the maintenance plan is marked with the Service Template of the to be maintained IT Service. The resulting simulation log is used to create log-based ordering relations and footprints like they are used in process mining and described in [18,19]. Based on these ordering relations two footprints are created. One footprint uses the basic ordering relations described in [18]. This footprint is used to derive the pattern instances IMMEDIATELY NEXT, PRECEDENCE, IMMEDIATELY PRECEDENCE and the activities a_{cur} and a_{nex} for the pattern instances of NEXT. In order to get a_{far} for the pattern instances of NEXT the second footprint is used, which is based on the extended ordering relations described in [19].

Instances of the pattern ALTERNATIVE OCCURRENCE and ALTERNATIVE ABSENCE are also derived from the second footprint. The pattern OCCURRENCE is instantiated with all simulated activities, which occur in every path of the simulation log. For the anti-pattern ABSENCE all possible activities are needed, which can be derived directly from the simulation log.

Instances of the pattern STATE-CONDITION can be derived from the activities in the simulation log and the corresponding function defined in Definition 4.5. The pattern VALUE can be instantiated by filtering all activities in the simulation log, whose attribute *prop* is not NULL.

5 Implementation

The architecture of the proof of concept implementation consists of four main components and is shown in Fig. 7. The first component is a modelling component, which allows to model maintenance plans and derive pattern instances of a maintenance plan. The modelling component is implemented in the software tool Horus[2] and already allows to model generic XML nets. The extension of the tool in order to model TOSCA service templates and link them to an XML net is currently under construction.

The second main component are the log agents. Log agents are used to get every new log entry of an application, transform the log entry into the format of a transaction event and send it to the complex event processing engine. In the proof of concept log agents are implemented with Beats and Logstash[3]. Both products are developed for fast log data extraction. Besides, Logstash contains a powerful regular expression engine, which supports the transformation of proprietary log entries into the generic format of transaction events.

[2] www.horus.biz.
[3] https://elastic.co.

The third component is an IT infrastructure monitoring tool like Nagios[4], CloudWatch[5], or Metricbeat[6] which allows to check the state of an application in order to generate the state events. In the proof of concept we use Metricbeat.

The fourth component is the complex event processing engine, which checks incoming state and transaction events against the pattern instances of the maintenance plan. In the proof of concept the complex event processing system of WSO2[7] is used. All anti-patterns are implemented as event queries in the event pattern language Siddhi[8] and have to be implemented only once. In order to check future maintenance plans, only the corresponding pattern instances have to be transferred to the complex event processing system. As an example, for an anti-pattern written in Siddhi, see the following anti-pattern NEXT, implemented as Siddhi query:

```
from te [(app == NEXT.appcur and op == NEXT.opcur
and prop == NEXT.propcur) in NEXT] insert into #temp;
from #temp as t join NEXT as n on t.app == n.appcur and
t.op == n.opcur and t.prop == n.propcur
select t.timestamp, n.appcur, n.opcur, n.propcur, n.appnex,
n.opnex, n.propnex, n.appfar, n.opfar, n.propfar
insert into #temp1;
from e1=#temp1 -> e2= incoming_te [e1.appfar == e2.app
and e1.opfar == e2.op and e1.propfar == e2.prop]
select e1.timestamp, e1.appcur, e1.opcur, e1.propcur,
e1.appnex, e1.opnex, e1.propnex, e2.timestamp as timestampfar
insert into #temp2;
from #temp2 [not((appnex == TEH.app and opnex == TEH.op
and propnex == TEH.prop in and timestamp < TEH.timestamp
and timestampentf >  TEH.timestamp) in TEH)]
select str:concat("The activity ",appnex, ", ", opnex, ", "
, propnex, " was not performed after the activity ", appcur,
" ,", opcur, ", ", propcur, ".") as message
insert into error_message;
```

Apart of the modelling component all components and Siddhi queries are implemented in a prototype, which is used to evaluate the approach. A first evaluation experiment was conducted, which is described in the following.

[4] https://nagios.org.
[5] https://aws.amazon.com/en/cloudwatch/.
[6] https://www.elastic.co/guide/en/beats/metricbeat/6.2/index.html.
[7] https://wso2.com/products/complex-event-processor/.
[8] https://github.com/wso2/siddhi.

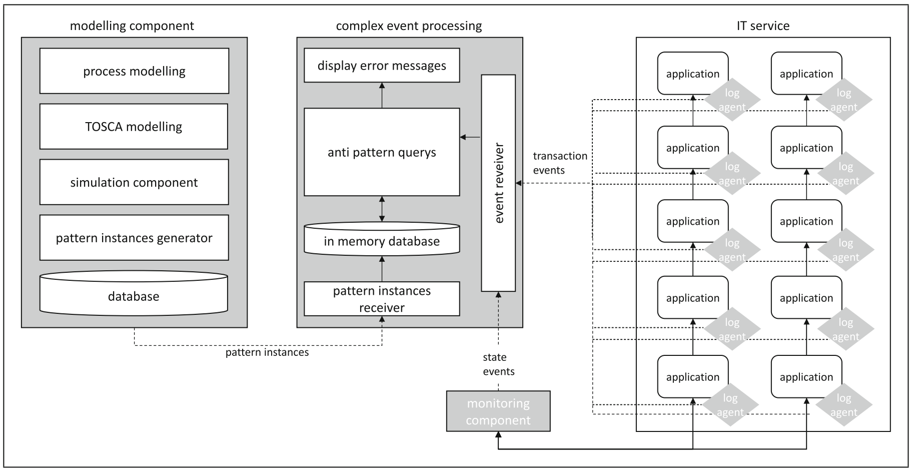

Fig. 7. Implementation architecture

6 Experimental Results

To evaluate the approach an exemplary IT service maintenance was performed. Therefore, we built an IT service environment using Amazon EC2[9] machines. The environment contains five EC2 machines. On two machines we installed an Apache webserver[10] hosting the open source application SugarCRM[11]. On two other machines Maria DB was installed. Both SugarCRM instances connect to the same Maria DB instance. In the experiment the configuration of both SugarCRM instances should be changed, so SugarCRM connects only to the second Maria DB instance anymore. The fifth EC2 machine is used to host the complex event processing engine and Logstash in order to transform log data and check the data for anti-patterns. Figure 8 gives an overview of the maintenance plan, which was used for evaluation and to derive the pattern instances for the experiment. The maintenance plan is described shortly in the following.

According to the maintenance plan first of all the database server named database2 needs to be started. Afterwards the loadbalancer is shut off in order to avoid connections to the webservers. When the loadbalancer is offline, the two webservers have to be stopped and reconfigured in order to connect to database2. After reconfiguring and stopping the webservers, they can be started again. If both webservers where stopped, the former database server can be shut down. Finally, the loadbalancer ha to be started again in order to redirect requests to the webservers. During the execution of this maintenance plan typical operator errors, like they are described in Sect. 2, were injected. Namely, those operator errors are:

[9] https://aws.amazon.com/ec2/.
[10] https://httpd.apache.org.
[11] https://www.sugarcrm.com.

1. Forgot to configure SugarCRM1
2. The loadbalancer was started before webserver2 was started
3. When configuring SugarCRM1 the IP address is changed and additionally without need the user is modified
4. Instead of stopping databaseserver1, databaseserver2 is stopped
5. The wrong IP address is entered, when configuring SugarCRM1
6. When changing the IP address a typo happens, so that the format of the IP address is xxx.xxx.xxxxxx instead of xxx.xxx.xxx.xxx
7. Webserver2 is started. However, the EC2 machine is stopped manually in order to simulate a software bug which hinders the webserver to start properly
8. A combination of error 1 and 2
9. A combination of error 3 and 4
10. A combination of error 3, 4 and 6.

In the first ten runs of the experiment one error was injected per run. Afterwards we repeated the experiment. However, in the second ten runs, errors were corrected immediately after their detection. By correcting them, the actual maintenance execution differs from the maintenance plan, because we did not model any procedural exception handlings.

In order to quantify the results, we used the metrics *Recall*, *Precision* and *F-Score* known from machine learning evaluations [24].

In the first ten runs all errors were identified and no false positives were reported, resulting in a Precision and Recall of 100%. The negative site of this is, that error messages were created multiple times for the same root cause. For example when the configuration of SugarCRM1 was forgotten (error 1) the anti-pattern NEXT as well as the anti-pattern PRECEDENCE created an error message, when the activity after the forgotten activity was executed. Additionally, the anti-pattern OCCURRENCE created an error message, because the occurrence of the activity could not be found in the transaction event history. In some runs this led to a ratio of up to four created error messages for one root cause.

In the second round of the experiment, when the errors were corrected after their identification, the precision decreased to 41%. The reason was the increasing number of false positives. For example when the forgotten configuration of SugarCRM1 was identified, the error was corrected by stopping webserver1 again, configuring SugarCRM1 and starting webserver1 again. When webserver1 was stopped respectively started the anti-pattern ABSENCE created two error messages that webserver1 was stopped respectively started too many times. Nonetheless, all injected errors were identified resulting in a Recall of 100%. Besides, the ratio of reported error messages to root causes improved, because of the immediate error handling. The F-Score over all runs is 73%. The F-Score of the first ten runs is 100%, whereas the F-Score only for the runs with immediate error handling is 58%.

In the next months we plan to conduct additional experiments to test the recall and precision of the method. Besides, we plan to perform performance

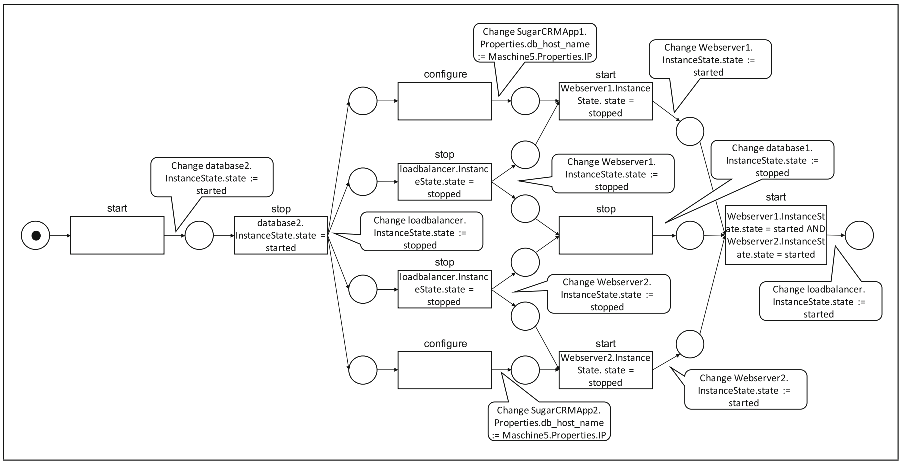
start
Change database2. InstanceState.state := started
stop
database2. InstanceState.state = started
Change loadbalancer. InstanceState.state := stopped
configure
stop
loadbalancer.InstanceState.state = stopped
stop
loadbalancer.InstanceState.state = stopped
configure
Change SugarCRMApp1. Properties.db_host_name := Maschine5.Properties.IP
Change SugarCRMApp2. Properties.db_host_name := Maschine5.Properties.IP
start
Webserver1.Instance State. state = stopped
Change Webserver1. InstanceState.state := stopped
stop
Change Webserver2. InstanceState.state := stopped
start
Webserver2.Instance State. state = stopped
Change Webserver1. InstanceState.state := started
Change database1. InstanceState.state := stopped
start
Webserver1.InstanceState.state = started AND Webserver2.InstanceState.state = started
Change loadbalancer. InstanceState.state := started
Change Webserver2. InstanceState.state := started

Fig. 8. Maintenance plan used for the experiment

tests as this is one of our main objectives, that the method reports errors within seconds. So, operators would have a realistic chance to correct their errors before they manifest in an IT service.

7 Discussion

The high false positive rate when error handling was performed can be interpreted as overfitting. On the contrary, the false positive rate would decrease, if error handling would be modelled explicitly in the maintenance plan. However, we think it is quite unlikely that an operator would model every possible exception, because this could lead to a very complex unreadable maintenance plan.

In general checking conformance of an event stream to a process model in an online setting leads to new challenges, that do not exist for traditional offline conformance checking methods. In offline conformance checking methods, analyzing event traces instead of event streams, the actual process execution can not be influenced anymore (apart of long running process, which are not finished when analyzing the log trace). In an online setting like in this work an operator would adjust the process execution spontaneous, because of the identification of errors or deviations from the process modell. This could lead to process executions differing a lot from the actual modelled process.

An error detection method or conformance checking method should be able to recognize, which deviations from the process modell are allowed ones, e.g. because of correcting an error and which one are real errors in order to reduce false positives. Therefore, we plan to extend our approach by adding a machine learning component, which analyzes which error messages are real errors and which were created because of correcting an already performed error.

8 Related Work

Related work can be separated in different areas of work. One area of work is the automation of typical operations like redeployments and integrated error exception handling, like it is provided by popular configuration management tools, e.g. Chef [21]. Those tools have the disadvantage, that they have just local information for error handling and no global view of the whole maintenance, which also could involve legacy systems [20].

Another area of work is the detection of configuration errors. Those approaches can be divided in rule based methods and online configuration validation [22]. Rule based methods try to avoid configuration errors a priori by correctness checks. These, help to detect wrong planned configuration errors. However, those approaches do not check if the configuration operation itself was executed as planned. So, forgotten configurations e.g. because a server was down or typos, when the configuration was done manually, cannot be detected.

The most related work to ours is the work of Xu et al. [20] and Farshchi et al. [23]. Both works describe an approach to monitor sporadic operations in cloud environments. Xu et al. developed a method called "POD-Diagnosis". They use a process model to detect operator errors through token replay by checking the conformance of observed logs with the prebuild model and an additional fault tree analysis in order to find the root cause of the error. In contrast to our work only the control flow of the process is modelled and can therefore be checked. Apart of that, in our approach no additional fault tree has to be build. Farshchi et al. build a regression-based model to find correlation and causalities between events described in logs and overserved metrics of resources. In their approach, assertions are derived from the regression-based model. However, they are also limited to control flow. Additionally, enough learning data is needed, which practically limits their approach to automated cloud environments. Our approach does not have to learn data and therefore can also be used to monitor manually executed steps or changes in legacy systems as long as those actions can be seen in the logs of the systems.

The field of business process compliance monitoring can also be seen as other related work, which uses patterns to check that process executions adhere to predefined compliance requirements [26]. Our work differs from this field in the way, that we do not use patterns to make sure, that specific compliance requirements like the "segregation of duty" pattern are covered during a process execution. The patterns used in this work are only used in order to be able to instantiate anti-patterns during process executions in order to identify situations, which must not occur. The notion of anti-pattern can be seen as the counterpart to compliance pattern [26]. The most related work to ours in this field is [27], who also use the notion of anti-patterns instead of compliance patterns. However, in our work no definition of patterns has to be performed by an user. Instances of the patterns are derived automatically from the created maintenance plan during design phase, which integrates the control flow and configuration modelling. The instantiation of patterns can even be influenced after modelling by the simulation of the maintenance plan. If paths of the maintenance plan will not be simulated, those will not be represented in the simulation log and therefore cannot be used for pattern instantiation. During the execution phase anti-patterns check only deviations from the derived pattern instances.

9 Conclusion

In this paper, we describe an approach to detect operator errors online during the execution of maintenance operations. Therefore, we define different anti-patterns, which are implemented as complex event processing queries and check in real time log entries and state metrics of observed resources against pattern instances of a predefined process model. The process model itself is realized as a TOSCA based XML net, combining the modelling of the control-flow and the resources. A first evaluation with a prototype implementation was performed, resulting in a very good error detection rate. On the contradictory site, the approach can result in a high false positive rate, when the process execution is adjusted spontaneously in order to correct the reported errors. Therefore, we plan to extend our approach in order to deal with this spontaneous flexibility during an IT service maintenance.

References

1. Gunawi, H.S., et al.: What bugs live in the cloud? A study of 3000+ issues in cloud systems. In: Proceedings of the ACM Symposium on Cloud Computing, pp. 1–14 (2014)
2. Hagen, S., Seibold, M., Kemper, A.: Efficient verification of IT change operations or: how we could have prevented Amazon's cloud outage. Presented at the Network Operations and Management Symposium (NOMS), 2012 IEEE, pp. 368–376 (2012)
3. Dumitra, T., Narasimhan, P.: Why do upgrades fail and what can we do about it? Toward dependable, online upgrades in enterprise system. In: Proceedings of the 10th ACM/IFIP/USENIX International Conference on Middleware, p. 18 (2009)
4. Pertet, S., Narasimhan, P.: Causes of failure in web applications. Parallel Data Laboratory, p. 48 (2005)
5. Oppenheimer, D., Ganapathi, A., Patterson, D.A.: Why do internet services fail, and what can be done about it? In: Proceedings of the 4th Conference on USENIX Symposium on Internet Technologies and Systems, vol. 4, Berkeley, p. 1 (2003)
6. Scott, D.: Making smart investments to reduce unplanned downtime. Tactical Guidelines Research Note TG-07-4033, Gartner Group, Stamford, CT (1999)
7. Elliot, S.: DevOps and the cost of downtime: fortune 1000 best practice metrics quantified. International Data Corporation, IDC (2014)
8. Vetter, A.: Detecting operator errors in cloud maintenance operations. In: 2016 IEEE International Conference on Cloud Computing Technology and Science (CloudCom), pp. 639–644 (2016)
9. Nagaraja, K., Oliveira, F., Bianchini, R., Martin, R.P., Nguyen, T.D.: Understanding and dealing with operator mistakes in internet services. In: OSDI 2004: 6th Symposium on Operating Systems Design and Implementation (2004)

10. Yin, Z., Ma, X., Zheng, J., Zhou, Y., Bairavasundaram, L.N., Pasupathy, S.: An empirical study on configuration errors in commercial and open source systems. In: Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles, pp. 159–172 (2011)
11. Peterson, J.L.: Petri Net Theory and the Modeling of Systems. Prentice Hall, Upper Saddle River (1981)
12. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Pearson Education, London (1994)
13. van der Aalst, W.M., Ter Hofstede, A.H., Kiepuszewski, B., Barros, A.P.: Workflow patterns. Distrib. Parallel Databases **14**(1), 5–51 (2003)
14. Russell, N., Ter Hofstede, A.H., Edmond, D., van der Aalst, W.M.: Workflow Data Patterns. QUT Technical report, FIT-TR-2004-01. Queensland University of Technology, Brisbane (2004)
15. Russell, N., van der Aalst, W.M.P., ter Hofstede, A.H.M., Edmond, D.: Workflow resource patterns: identification, representation and tool support. In: Pastor, O., Falcão e Cunha, J. (eds.) CAiSE 2005. LNCS, vol. 3520, pp. 216–232. Springer, Heidelberg (2005). https://doi.org/10.1007/11431855_16
16. Riehle, D., Züllighoven, H.: Understanding and using patterns in software development. TAPOS **2**(1), 3–13 (1996)
17. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Property specification patterns for finite-state verification. In: Proceedings of the Second Workshop on Formal Methods in Software Practice, pp. 7–15 (1998)
18. Van Der Aalst, W.: Process Mining: Discovery, Conformance and Enhancement of Business Processes. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-19345-3
19. Weidlich, M., Mendling, J., Weske, M.: Computation of behavioural profiles of process models. Business Process Technology, Hasso Plattner Institute for IT-Systems Engineering, Potsdam (2009)
20. Xu, X., Zhu, L., Weber, I., Bass, L., et al.: POD-diagnosis: error diagnosis of sporadic operations on cloud applications. In: 2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, pp. 252–263 (2014)
21. Chef: About Handlers, 08 November 2017. https://docs.chef.io/handlers.html
22. Xu, T., Zhou, Y.: Systems approaches to tackling configuration errors: a survey (2014)
23. Farshchi, M., Schneider, J.-G., Weber, I., Grundy, J.: Metric selection and anomaly detection for cloud operations using log and metric correlation analysis. J. Syst. Softw. **137**, 531–549 (2017)
24. Powers, D.: Evaluation: from precision, recall and F-measure to ROC, informedness, markedness and correlation. J. Mach. Learn. Technol. **2**(1), 37–63 (2011)
25. Kopp, O., Binz, T., Breitenbücher, U., Leymann, F.: BPMN4TOSCA: a domain-specific language to model management plans for composite applications. In: Mendling, J., Weidlich, M. (eds.) BPMN 2012. LNBIP, vol. 125, pp. 38–52. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-33155-8_4
26. Becker, M., Klingner, S.: A criteria catalogue for evaluating business process pattern approaches. In: Bider, I., et al. (eds.) BPMDS/EMMSAD-2014. LNBIP, vol. 175, pp. 257–271. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-43745-2_18
27. Awad, A., Barnawi, A., Elgammal, A., Elshawi, R., Almalaise, A., Sakr, S.: Runtime detection of business process compliance violations: an approach based on anti patterns. In: 12th Enterprise Engineering Track at ACM, SAC 2015 (2015)