Detecting 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. In this paper, we propose a monitoring approach to detect operator errors 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]. 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].

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 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 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. Fig. 1 gives an overview of the general monitoring approach.

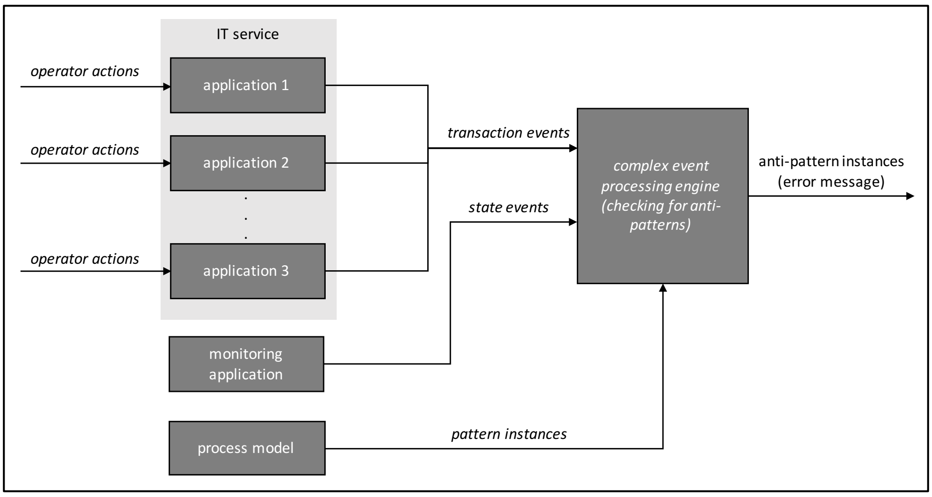

Fig. 1. General monitoring approach

The remainder of this paper is organized as follows: The next section gives a short overview of typical operator errors. Section three describes the fundamentals of TOSCA and XML nets, which are used to model the actual maintenance. Section four describes the concept of patterns and anti-patterns. Section five presents the proof of concept implementation. Afterwards related work is presented. Section seven concludes the paper.

2 OPERATOR ERRORS

Oppenheimer et al. [5] and many other authors like [4], [5], [11], [12] 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 I 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 (Topology and Orchestration Specification for Cloud Applications) 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.

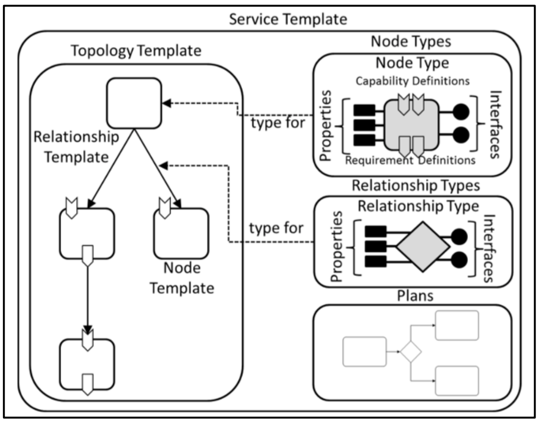

Fig. 2. TOSCA Service Template

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 *Operation*s, which are offered by the cloud application components and can be written in BPMN, BPEL or other languages. We use the notation of XML nets for the creation of *Plans,* which we name "maintenance plan" in the rest of the paper.

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) $\Psi =< D, FT, PR >$ 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) I_N is the function that assigns additionally a Node Type to each place.

(v) 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.

(v) 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.

(vi) 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.

(vii) M_0 is the initial marking. Markings are TOSCA Service Templates.

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

Fig. 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). *MyApplication* is hosted on *MyAppServer* and requires additionally the database *TestDatabase*. 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 selects *MyApplication* with the condition, that it is started. Before *MyApplication* can be configured it has to be stopped, which is represented in the first transition. 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, 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.

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 this definition 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, 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.

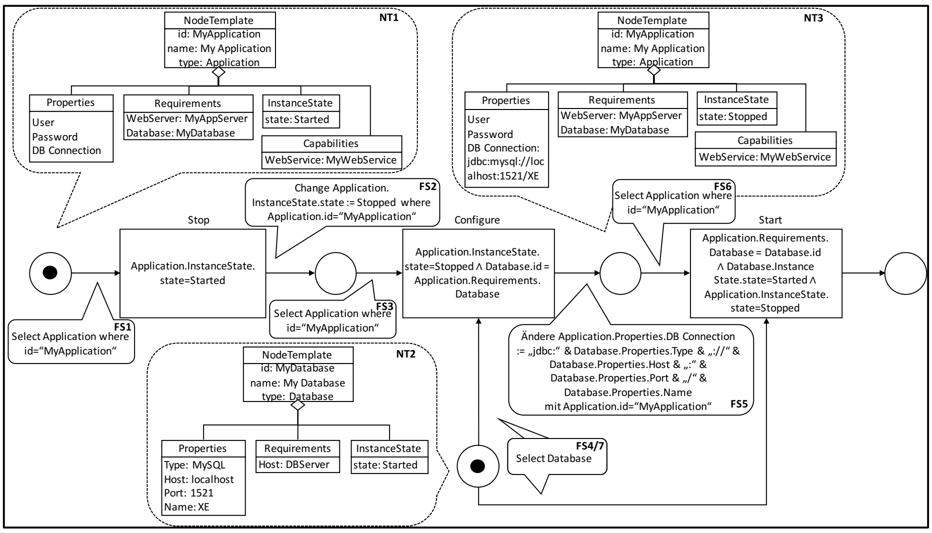

Fig. 3. Example of a TOSCA based XML net

4.1 Context

As described in chapter three, 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.

DEFINITON 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*.

DEFINITON 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.

DEFINITON 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 SC_i, $i \in SM$. The set of all transition inscriptions of a maintenance plan is defined as STI_i, $i \in SM$. The function $F: SC_i \rightarrow STI_i$ assigns a transition to each activity.

DEFINITON 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 SA_i, $i \in SM$. The set of all transitions of a maintenance plan is defined as ST_i, $i \in SM$. The function $F: SA_i \rightarrow ST_i$ 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.

DEFINITON 4.7: (TRANSACTION EVENT HISTORY). A *transaction event history* is a selection σ on the set of transaction events, which are in the time scope of the scheduled maintenance:

$$\text{TEH} \coloneqq \sigma_{timestamp \geq maintenance\_start\ \wedge timestamp \leq maintenance\_end}(TES)$$

DEFINITON 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*.

DEFINITON 4.9: (TIME). time is a function, which returns the current timestamp.

DEFINITON 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.

DEFINITON 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 STAE-CONDITION. The first eight patterns are highly influenced by the specification pattern of Dwyer at 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.
- **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.

Due to space limitations, we only present the three patterns and anti-patterns NEXT, STATE-CONDITION and VALUE in detail.

Pattern NEXT

- **Description**: This pattern describes pairs of activities, defining which activity has to occur after another. 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 $P1_i \coloneqq AM_i \times AM_i \times AM_i$ with the tuples $(a_{cur}, a_{nex}, a_{far})$ where,
  - the corresponding transitions of the activities t_{cur} and t_{nex} are connected through the same place,
  - t_{cur}, t_{nex} und t_{far} have to occur in the same path,
  - t_{far} always has to occur after t_{cur},
  - t_{far} and t_{cur} may not be connected through the same place.
- **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 $\mathrm{te}_{\mathrm{cur}}$ in the event stream ES occurs and none of the next events $\mathrm{te}_{\mathrm{nex}}$ conforms to the next activity $\mathrm{a}_{\mathrm{nex}}$. However, one of the next events conforms to an activity $\mathrm{a}_{\mathrm{far}}$:

$$\pi_{\mathrm{app,op,prop}}\mathrm{te}_{\mathrm{akt}} \in \pi_{\mathrm{a_{cur}.app,a_{cur}.op,a_{cur}.prop}}\mathrm{P1_i} \succ$$
$$\pi_{\mathrm{app,op,prop}}\mathrm{te}_{\mathrm{nex}} \notin$$
$$\pi_{\mathrm{app,op,prop}}\left(\pi_{\mathrm{a_{nex}}}\left(\sigma_{\mathrm{a_{cur}.app=te_{cur}.app \wedge a_{cur}.op=te_{cur}.pp \wedge a_{cur}.prop=te_{cur}.prop}}\mathrm{P1_i}\right)\right) \wedge$$
$$\pi_{\mathrm{app,op,prop}}\mathrm{te} \in$$
$$\pi_{\mathrm{app,op,prop}}\left(\pi_{\mathrm{a_{far}}}\left(\sigma_{\mathrm{a_{akt}.app=te_{cur}.app \wedge a_{cur}.op=te_{cur}.pp \wedge a_{cur}.prop=te_{cur}.prop}}\mathrm{P1_i}\right)\right)$$

- **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 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 SC_i for a maintenance plan i.
- **Anti-pattern**: This anti-pattern does actually not detect an error like described in chapter 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:
$$\pi_{app,op,prop} te \in \pi_{app,op,prop} SC_i \land$$
$$\pi_{zapp,state}(\sigma_{app=ve.app \land op=ve.op \land prop=ve.prop} \mathrm{SC}) \;/\; \pi_{app,state}(SEH) \neq \emptyset$$
- **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: $P3_i \coloneqq \pi_{app,op,prop,value}(\sigma_{prop \neq NULL} SA_i)$.
- **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:*
$$\pi_{app,op,prop} ve \in \pi_{app,op,prop} P3_i \land \pi_{app,op,prop,value} ve \notin P3_i$$
- **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 the operations.

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 and the functions described in chapter 4.1 all pattern instances of the maintenance plan can be derived automatically.

5 Implementation

The architecture of the proof of concept implementation consists of four main components. 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[1] 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[2]. 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.

The third component is an IT infrastructure monitoring tool like Nagios[3], or CloudWatch[4], which allows to check the state of an application in order to generate the state events.

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[5] is used. All anti-patterns are implemented as event queries in the event pattern language Siddhi[6] 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 == NACHFOLGER.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.ap-
pnex, 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 ti-
mestampfar 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)]
```

[1] www.horus.biz

[2] https://elastic.co

[3] https://nagios.org

[4] https://aws.amazon.com/en/cloudwatch/

[5] https://wso2.com/products/complex-event-processor/

[6] https://github.com/wso2/siddhi

```
select str:concat("The activity ",appnex, ", ", opnex, ",
", propnex, " was not performed after the activity ", ap-
pcur, " ,", opcur, ", ", propcur, ".") as message insert
into error_message;
```

Apart of the modelling component all components and Siddhi queries are already implemented in a prototype.

In the next months, an evaluation of the whole method will be performed. Therefore it is planned to replay common configuration settings like they are described for example in [9], [20]. Main evaluation criteria will be the time needed to detect operator errors, the ratio of detected/injected operator errors and the false positive rate.

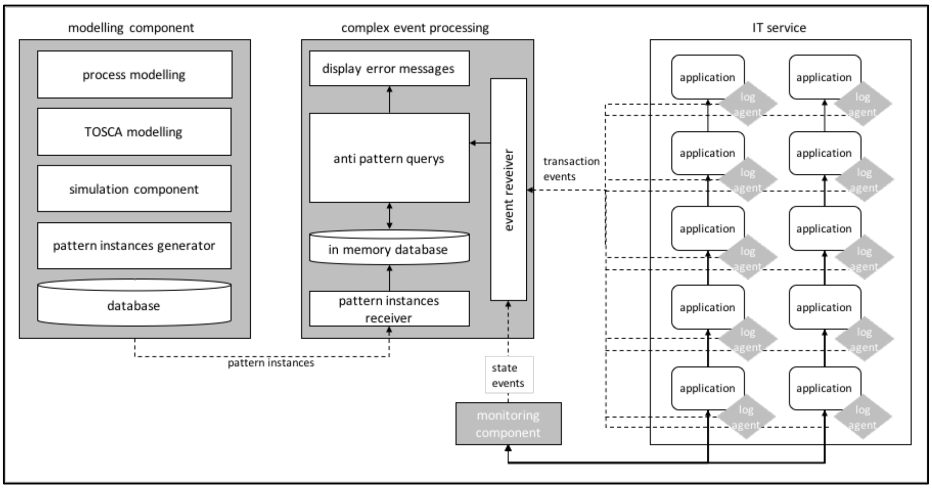

Fig. 4. Implementation architecture

6 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 pre-build model and an additional fault tree analyses 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.

7 Conclusion

In this paper, we describe an approach to detect operator errors 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 pre-defined 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 prototype to check the effectiveness of our approach is currently under construction. In order to evaluate the approach typical maintenance operations will be performed like the configuration of servers or a rolling upgrade. During these maintenance operations, typical errors will be injected on purpose. The prototype should be able to detect all those injected errors.

References

[1] H. S. Gunawi *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*, 2014, pp. 1–14.

[2] S. Hagen, M. Seibold, and A. Kemper, "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, 2012, pp. 368–376.

[3] T. Dumitraş and P. Narasimhan, "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*, 2009, p. 18.

[4] S. Pertet and P. Narasimhan, "Causes of failure in web applications," *Parallel Data Lab.*, p. 48, 2005.

[5] D. Oppenheimer, A. Ganapathi, and D. A. Patterson, "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 - Volume 4*, Berkeley, CA,

USA, 2003, pp. 1–1.

[6] D. Scott, “Making smart investments to reduce unplanned downtime,” *Tactical Guidel. Res. Note Note TG-07-4033 Gart. Group Stamford CT*, 1999.

[7] S. Elliot, “DevOps and the cost of downtime: Fortune 1000 best practice metrics quantified,” *Int. Data Corp. IDC*, 2014.

[8] A. Vetter, “Detecting Operator Errors in Cloud Maintenance Operations,” in *2016 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, 2016, pp. 639–644.

[9] K. Nagaraja, F. Oliveira, R. Bianchini, R. P. Martin, and T. D. Nguyen, “Understanding and Dealing with Operator Mistakes in Internet Services,” in *OSDI ’04: 6th Symposium on Operating Systems Design and Implementation*, 2004.

[10] Z. Yin, X. Ma, J. Zheng, Y. Zhou, L. N. Bairavasundaram, and S. Pasupathy, “An empirical study on configuration errors in commercial and open source systems,” in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, 2011, pp. 159–172.

[11] J. L. Peterson, “Petri net theory and the modeling of systems,” 1981.

[12] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: elements of reusable object-oriented software*. Pearson Education, 1994.

[13] W. M. van der Aalst, A. H. Ter Hofstede, B. Kiepuszewski, and A. P. Barros, “Workflow patterns,” *Distrib. Parallel Databases*, vol. 14, no. 1, pp. 5–51, 2003.

[14] N. Russell, A. H. Ter Hofstede, D. Edmond, and W. M. van der Aalst, “Workflow data patterns,” QUT Technical report, FIT-TR-2004-01, Queensland University of Technology, Brisbane, 2004.

[15] N. Russell, A. H. Ter Hofstede, D. Edmond, and W. M. van der Aalst, “Workflow resource patterns,” 2005.

[16] D. Riehle and H. Züllighoven, “Understanding and using patterns in software development,” *TAPOS*, vol. 2, no. 1, pp. 3–13, 1996.

[17] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett, “Property specification patterns for finite-state verification,” in *Proceedings of the second workshop on Formal methods in software practice*, 1998, pp. 7–15.

[18] W. Van Der Aalst, *Process mining: discovery, conformance and enhancement of business processes*. Springer Science & Business Media, 2011.

[19] M. Weidlich, J. Mendling, and M. Weske, “Computation of behavioural profiles of process models,” *Bus. Process Technol. Hasso Plattner Inst. IT-Syst. Eng. Potsdam*, 2009.

[20] X. Xu, L. Zhu, I. Weber, L. Bass, and others, “POD-diagnosis: Error diagnosis of sporadic operations on cloud applications,” in *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 2014, pp. 252–263.

[21] Chef, “About Handlers,” 08-Nov-2017. [Online]. Available: https://docs.chef.io/handlers.html.

[22] T. XU and Y. ZHOU, “Systems Approaches to Tackling Configuration Errors: A Survey,” 2014.

[23] M. Farshchi, J.-G. Schneider, I. Weber, and J. Grundy, “Metric selection and anomaly detection for cloud operations using log and metric correlation analysis,” *J. Syst. Softw.*, Mar. 2017.