UNIVERSITY OF TARTU
Institute of Computer Science
Software Engineering Curriculum

Ilia Aphtsiauri

Declarative Process Mining on the Cloud

Master's Thesis (30 ECTS)

Supervisor: Fabrizio Maggi, PHD

Tartu 2017

Acknowledgements

I would like to express my sincere gratitude to my supervisor Dr. Fabrizio Maggi who provided guidance and colossal support, also big thanks to the academic staff of the Computer Science faculty.

I wish to thank my friends and great colleagues at Veriff ÖU for support and flexible working hours.

Last but not least I would like to thank my family and friends for being with me on every step of the way.

Declarative Process Mining on the Cloud

Abstract:

This thesis provides an overview of the Declare language and declarative process mining algorithms, followed by the description of currently available tools for a declarative process mining. This thesis provides the availability of all the discussed tools on a cloud platform and introduces two new tools. One provides the event monitoring capabilities and and the other one generates a verbal representation of a Declare model. All the described process mining tools are implemented as bundles of the cloud platform RuM. Afterwards, the new user interface and functionalities of the tools are described. The evaluation part of the thesis presents, the mining tools on the cloud and the capabilities of the live event monitoring tool.

Keywords: process mining, declarative process modelling, cloud applications

CERCS: P170

Deklaratiivne protsessikaeve pilveteenuses

Lühikokkuvõte:

Antud magistritöö annab ülevaate deklaratiivse keele ja deklaratiivse protsessikaeve algoritmide kohta. Sellele järgneb deklaratiivse protsessikaeve tarvis kasutatavate vahendite kirjeldus. Töö tagab eelnevalt käsitletud vahendite kättesaadavust pilvplatvormil ning tutvustab kaks uut vahendit, mis pakuvad sündmuse seirevõimekust ja deklaratiivse mudeli suulise esitluse genereerimist. Kõik kirjeldatud protsessikaeve vahendid on rakendatud kimpudena pilvplatvormil RuM. Samuti on kirjeldatud uus kasutajaliides ja vahendite funktsioonid. Töö hindamisosas olid esitatud pilvel olevad kaevevahendid ja otsesündmuste seirevahendi võimed.

Võtmesõnad:

protsessikaeve, deklaratiivne protsessi modelleerimine, pilvrakendused

CERCS: P170

Table of Contents

1 Introduction ... 6
1.1 The aim of the thesis ... 7
1.1.1 Process Discovery ... 7
1.1.2 Log Generation ... 7
1.1.3 Runtime Monitoring ... 8
1.2 Structure of the thesis ... 8
2 Background ... 9
2.1 Process mining ... 9
2.2 Linear Temporal Logic ... 10
2.3 Declare: LTL-Based Constraint Language ... 11
2.3.1 Existence Templates ... 12
2.3.2 Relation Templates ... 13
2.3.3 Negation Templates ... 14
2.4 Event Log Specification ... 15
2.4.1 Mining eXtensible Markup Language ... 16
2.4.2 eXtensible Event Stream ... 17
3 Contribution ... 21
3.1 Declarative Process Mining Tools ... 21
3.1.1 MINERful ... 21
3.1.2 Declare Miner ... 21
3.1.3 Deviance Miner ... 22
3.1.4 MINERful Simplification ... 22
3.1.5 Log Generation ... 22
3.1.6 FLLOAT ... 23
3.1.7 MobuconLTL, MobuconLDL, Online Analyzer ... 23
3.1.8 Open Services Gateway Initiative (OSGi) ... 24
3.2 Mining Bundle Development ... 24
3.2.1 Bundle Manifest ... 24
3.2.2 Bundle Input Output ... 25
3.2.3 Bundle Registration into RuM ... 27
3.2.4 Bundle lifecycle ... 28
3.3 Monitoring Bundle Development ... 29
3.3.1 Monitoring Bundle Implementation ... 31
3.3.1.1 Runtime Verification Bundle ... 32

3.3.1.2 Conformance Visualizer Bundle Implementation ... 32
4 Evaluation ... 35
4.1 MINERful ... 35
4.2 MINERful Simplification ... 35
4.3 MINERful Log Generation ... 36
4.4 Declare Model Verbalization ... 38
4.5 FLLOAT ... 38
4.6 MoBuConLTL, MoBuConLDL, Online Analyser ... 40
5 Conclusion ... 44
6 References ... 45
Appendix ... 47
I. License ... 47

1 Introduction

Business Process Management (BPM) is the science of developing, analyzing, and managing all the processes performed by an organization. BPM identifies the processes carried out in the organization, measures their performance and productivity, trying to improve the results over the time by optimizing them. Optimization may include, cost, error or execution time reduction [1].

Nowadays, most of the systems store process execution information, in the form of event logs, which are the entry point of any business process mining technique. The goal of process mining is to identify or provide meaningful information, which can help analyze and improve the current process. Properly constructed event log should consist of a case (an instance of the process) and activity (an individual step in the process). Additional information for example timestamp, resources involved in the event is also used in process mining techniques [2].

Figure 1 shows the three types of process mining operations that can be executed on the event log. Process discovery - the primary objective is to discover a business process model from the event log, without any prior information about the process. Conformance checking - takes as an input an event log and a process model and it checks if the log is compliant with the model. Enhancement - the primary objective of this operation is to improve or extend an existing process model; as an input this procedure takes a process model and an event log, the result is a new improved or extended process model [2].

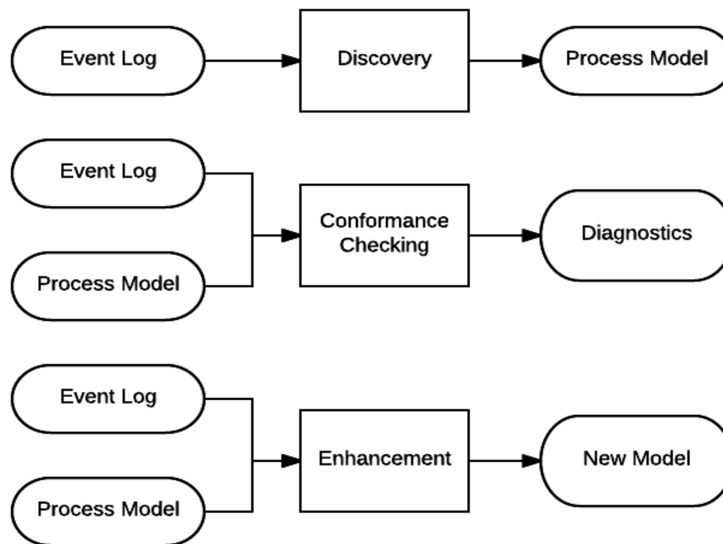


Figure 1. Three types of process mining operations (Adopted from [2])

Process discovery is one of the key components of business process management, the goal of which is to deliver the business process model. Taking into consideration the complexity of the process discovery step, multiple techniques were introduced, to extract a complete process model from the event logs. The majority of this methods produce a procedural model. A procedural model explicitly specifies all the possible behaviors and choices in the process (not specified behaviors are forbidden), as a result, procedural process models obtained from process discovery techniques are often spaghetti-like and very hard to interpret.

This led to the development of a different process mining approach – the declarative approach. Declarative process models specify behavioral constraints meaning that if the restriction does not prohibit something, it is allowed. As a result, this produces more compact and flexible models [3].

1.1 The aim of the thesis

The aim of the thesis is to implement a repository of declarative process mining tools as a part of a cloud platform. Thus allowing the users to benefit from running resource extensive tasks, easily accessibility and platform independence. The tools are divided into the following groups: Process Discovery, Log Generation and Runtime Monitoring

1.1.1 Process Discovery

MINERful – Considering the adjustability of artful processes, a procedural representation can lead to highly entangled models. As a result, the models are extremely complex to understand and person analysing them is susceptible to errors. Therefore, it is possible to describe the process models in a declarative way. MINERful was designed to discover control-flow of artful processes and produce declarative process models [8].

MINERful Simplification – Declarative process discovery algorithms use event logs to validate discovered constraints which may result in disregarding constraint interaction. Hence, discovered process models may contain counter excluding constraints, therefore there is no possibility to have traces compliant with all of them together. Additionally, the discovered models can contain redundant constraints. This algorithm addresses this issues and automatically trims the discovered models to exclude redundancies and resolves interaction collisions. The algorithm uses the automata-product monoid concept to assure model consistency and removal of redundant constraints [13] [14].

Declare Miner– Declare Miner is a two-phase algorithm for declare discovery, in the first phase an Apriori algorithm is used to identify frequent activities in the event log. The second phase operates on discovered frequent activity sets and produces candidate constraints. Finally, the candidate constraints are trimmed to the ones that are satisfied in the log using *Sequence Analysis* [9].

Declare Model Verbalization – Provides a verbal representation of the discovered declarative models.

Declare Deviance Miner – Business process deviance mining is a branch of business process mining. The primary objective of deviance mining is to provide reasons why an event log deviated from a defined process model. The deviations can be either positive or negative. Positive deviations result in higher performance metrics than expected, like achieving flawless conclusive results in a smaller amount of time, with smaller expenses or with lower resource utilization. Negative deviance, on the other hand, is when process outcome is unfavourable [11].

1.1.2 Log Generation

MINERful Model To Log – Simulated generation of event logs from a process model is crucial for testing and verifying the correctness of process discovery algorithms. Therefore, a tool providing this capability for declarative process discovery techniques is important. This thesis provides an implementation of a simulated event log generation algorithm from Declare process models. The Declare models can be provided either as a JSON file or as a

standard Declare XML file. The algorithm translates the constraints presented in the Declare model into regular expressions and afterwards employs Finite State Automata to simulate the process. The user additionally can define as an input number of traces and length [15].

1.1.3 Runtime Monitoring

FLLOAT – Runtime monitoring should provide the means to check whether the running processes satisfy the defined constraints and rules and is considered as crucial task to supply proper operational decision support. Providing precise runtime tracking capabilities is usually delegated to the verification branch. Thus plugin provides several verifications techniques for finite state automata. The goal of verification is to inspect the concerning system properties and confirm if they meet the defined standards [18].

MoBuConLTL, MobuConFLLOAT– Verifying all the details of process beforehand is impossible on multiple occasions. It will be incorrect to consider that participant behavior can be known. For this reason, runtime verification capabilities are provided. These plugins provide runtime monitoring functionalities and verify model compliance at runtime translating rules into automata [12] [16].

Online Analyser – Is a tool for runtime verification of multi-perspective declarative models. Multi-perspective monitoring means that the processes are not only evaluated in terms of the sequences of events but also by aspects of data and time [17].

Runtime Monitor Visualizer – Provides easily understandable graphical user interface, which displays the data provided by the runtime monitoring tools.

1.2 Structure of the thesis

This thesis has the following structure. Section 2 provides a background information about process mining. Section 3 introduces the tools which are part of Declarative process mining repository and in detail discusses the implementation approaches. Section 4 provides the evaluation of the tools which were implemented and explains their functional capabilities.

2 Background

2.1 Process mining

Process mining's fundamental objective is to discover, monitor and provide improvement ways of the processes at hand. The entry point of process mining is an event log. Each separate data entry is referred as an event. The event should contain information about the executed step in the process (activity), each activity should be part of an instance of the process (case). Additionally, an event can hold information regarding the actors executing the activity, timestamp and information about the data which was needed to execute the activity. Event data can be stored in different data storages like databases, mail archives etc. The efficiency of process mining tools highly depends on the event log quality, therefore, for the systems that plan to support the process analyses, it is crucial to treat log as first-class citizen artifacts [2].

The following criteria measure the quality of a log. Trustworthiness recorded events are part of a given case and the information they hold is correct. Completeness the scope should not be missing any event. Semantics of the stored events, should be clearly described. Security actors should be aware of stored event types and what is the purpose of saving them. Log quality ranges from excellent (★★★★★) to poor (★). Table 1 provides detailed explanations of the quality levels [2].

Table 1 Log quality levels (Adopted from [2])

Level	Characterization
★★★★★	The log is trustworthy and complete. Events are distinctly outlined. Stored events follow the precise semantics of defined ontology. Events are stored automatically in a systematic and reliable way. Security concerns are taken into consideration.
★★★★	The log is trustworthy and complete. Events are stored automatically in a systematic and reliable way. Activities and cases are mentioned distinctly.
★★★	Events are automatically but not regularly stored. At least some portion of stored events match reality, meaning that the log is not complete, but the information stored in the events is correct.
★★	Events are automatically but not regularly stored. No conventional approach is designed to define which events to store. Additionally, it is possible to complete the process without storing the events which result in missing events.
★	Events are missing, and recording events do not hold real and complete data. Handwritten event logs usually have such a tendency.

Three types of process mining operations can be executed on an event log. Figure 1 provides the input and output overview of the operations. Process discovery - the primary objective is to discover the model from the event log, without any prior information about it. Created models are represented using for example UML activity diagram, BPMN or Petri net. Conformance checking takes as an input an event log and a process model. The operation can

be applied on various types of models: procedural, declarative, organizational, etc. Conformance checking verifies if the given model and event log are compliant. This operation yields the data which displays the discrepancies between the log and the model. Enhancement - the primary objective of this activity is to improve or extend an existing process model, as an input this operation takes a process model and an event log, the result is a new improved or extended process model [2].

Producing clear and understandable process models is one of the key goals of process discovery. Procedural approaches are dependent on describing the control-flow by providing all the possible options on process execution, and this results in extremely complex models especially for processes characterized by high variability. Alternative to procedural approach is declarative approach. Declarative process models specify behavioral constraints meaning that if the constraint does not prohibit something, it is allowed. As a result, this produces more compact and flexible models. One of the languages used in the declarative approaches is Declare. Declare is based on Linear Temporal Logic, which is introduced in Section 2.2, Section 2.3 afterwards will describe Declare language [3].

2.2 Linear Temporal Logic

Linear Temporal Logic (LTL) is the language that depicts series of state progressions in a reactive system. The progression is a conversion through states. A state of the system is described as propositional formulae. Because of this declarative characteristics, LTL's semantic is used for a constraint specification in a declarative process model. The process model is an automaton which can administer if the state constraints are satisfied during the execution. The specifics of LTL formula syntax are adopted for declarative process modeling and is illustrated in Table 2 [4].

Table 2 LTL formula operators, σ represents trace of events, p LTL formula, $\models$ denotes that p satisfies σ (Adopted from [4])

Operator	**Definition**
not(!)	$\sigma \models !p$ holds true if $\sigma \models p$ is not satisfied
and($\wedge$)	$\sigma \models p \wedge q$ holds true if $\sigma \models p$ and $\sigma \models q$
or($\vee$)	$\sigma \models p \vee q$ holds true if $\sigma \models p$ or $\sigma \models q$
next($\bigcirc$)	$\sigma \models \bigcirc p$ holds true in the next occurrence
until (U)	$\sigma \models p\mathrm{U}q$ holds true if p holds true until q holds. q is in the current or transitioned to future state
eventually ($\Diamond$)	Is an abbreviation of $\Diamond p = (p \vee !p)\mathrm{U}p$, eventually - indicates that the constraint holds true before (including) last occurrence, in time
always ($\Box$)	Is an abbreviation of $\Box p = !\Diamond !p$, always - indicates that from current until the last occurrence of the constraint it holds true
weak until (W)	Is an abbreviation of $pWq = (pUq) \vee (\Box p)$, weak until – indicates that the constraint holds true until (U) some period of time or always

2.3 Declare: LTL-Based Constraint Language

Traditional workflow management system languages such as BPMN, EPC, UML etc. specify the process model executions as a set of step-by-step instructions. This specification is referred as an imperative modelling approach. Thus a highly structured process is produced. As a result, execution decisions are planned during the modelling phase and users do not have the flexibility to modify the process model at run-time [5] [4].

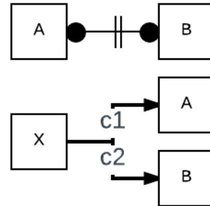


Figure 2. Imperative vs. declarative approach (Adopted from [5])

Declarative process modelling approach defines the process models by specifying a list of constraints, which should not be violated during the execution. As a result, the control flow of the process is inherently defined. Figure 2 shows the difference between the two approaches, given the situation that if A is executed, execution of B is prohibited and vice versa. This is easily presented with the help of not co-existence template, but in imperative languages, there are no such constructs. Therefore, the lower level construct, such as decision “X” is used. Also, conditions “c1” and “c2” should be specified as mutually exclusive [5].

Declare represents a constraint-based modelling system. Furthermore, Declare is not limited to the declarative process modelling features, it also supports: model development, model verification, automated model execution, changing models at run-time, executed processes analysis, and large process decomposition features which are traditional for other workflow management systems [5].

The Declare language was introduced to simplify constraint semantics for people who are not familiar with LTL, constraints in Declare are represented graphically. Declare offers rich verity of templates. Templates consist of a name, an LTL formula and a graphical representation, as shown in Figure 3. Here, the constraint defines that event (A, completed), should be followed by (B, completed) at least once. LTL representation of this constraint is $\Box((A,\ completed) \Rightarrow \Diamond(B,\ completed))$. Instead of having to specify the formulas, the graphical representation of the response template can be used. Figure 3 shows the graphical representation of the constraint. Declare counts about twenty templates which are grouped into three thematic groups. *Existence* templates specify the number of activity executions. *Relation* templates specify connection among multiple activities. *Negation* templates specify negative connections [4].

Declare Template

name	response
LTL	$\Box((\boldsymbol{A},\ completed) \Rightarrow \Diamond(\boldsymbol{B},\ completed))$

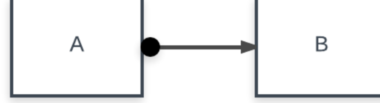
graphical	

Figure 3. Declare constraint template representation (Adopted from [4])

2.3.1 Existence Templates

Existence templates require a single event and define its position or cardinality in a trace**.** Figure 4 shows the graphical representations of the *existence* templates**.** The group is divided into four subgroups. *Existence* subgroup specifies least amount of times the event should be executed. Absence subgroup designates the maximum number of times the activity can be performed. *Exactly*, provides the exact number of times the activity should be executed. *Init*, specifies that the trace case should start with the specified event [4].

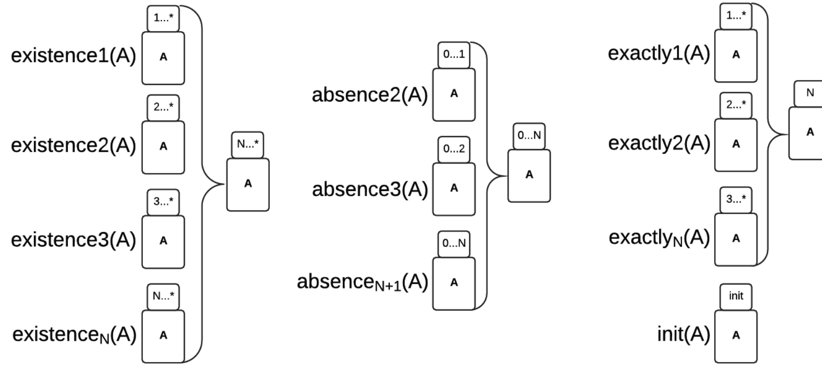


Figure 4. Notation for the existence templates (Adopted from [4])

The graphical representation of the existence templates as the corresponding LTL formulas shown in Table 3. *existence*(A) defines that the event A should be eventually be fulfilled during the case execution. *existence2*(A) recursively specifies the lower bound of the number of executions. *Absence* template negates the existence template, which results in setting the upper bound of the execution number. *Exactly*, template functions are represented as the combination of the $existence_N$ and $absence_{N+1}$ functions. *init*(A) defines that A should hold as a starting event of the given case [4].

Table 3. LTL formulas for existence templates (Adopted from [4])

Template name	**LTL formula**
existence(A)	$\Diamond$(A)
existence2(A)	$\Diamond$((A) $\wedge$ $\bigcirc$(existence(A)))
existence3(A)	$\Diamond$((A) $\wedge$ $\bigcirc$(existence2(A)))
…	…
$existence_N$(A)	$\Diamond$((A) $\wedge$ $\bigcirc$($existence_{N-1}$(A)))
absence2(A)	*!existence2*(A)
absence3(A)	*!existence3*(A)

… $absence_N$(A)	… $!existence_N$(A)
exactly1(A) *exactly2*(A) … $exactly_N$(A)	*existence*(A) $\wedge$ absence2(A) *existence2*(A) $\wedge$ absence3(A) … $existence_N$(A) $\wedge$ $absence_{N+1}$(A)
init(A)	*A*

2.3.2 Relation Templates

Relation templates describe relationships of multiple activities. For simplicity purposes in the examples provided here we have only two activities A and B as parameters. The line that connects these two activities represents a unique LTL formula which defines their relationship. Figure 5 shows the graphical representation of the relation templates.

Templates *responded existence* and *co-existence* are not concerned with activity execution order. The *responded existence* template defines that activity B should execute prior or after the activity A is executed. The *co-existence* template states that if either A or B activity is executed the remaining one should also be executed.

For templates *response, precedence* and *succession* activity execution order is important, but it does not state that executions should follow straight after each other meaning that other activities can be executed in between. In order to successfully execute *response*, completion of A should be followed by execution of B. *Precedence*, specifies that execution of B should be preceded by execution of A. *Succession*, is a bi-directional blend of *response* and *precedence* and both should be satisfied. *Alternate response*, *alternate precedence* and *alternate succession* specify that the execution of events A and B should alternate, meaning that activity A cannot be executed twice before executing B. *Chain response*, *chain precedence* and *chain succession* strictly state that execution of activities A and B should be directly after each other. LTL formulas of the relation templates are shown in Table 4 [4].

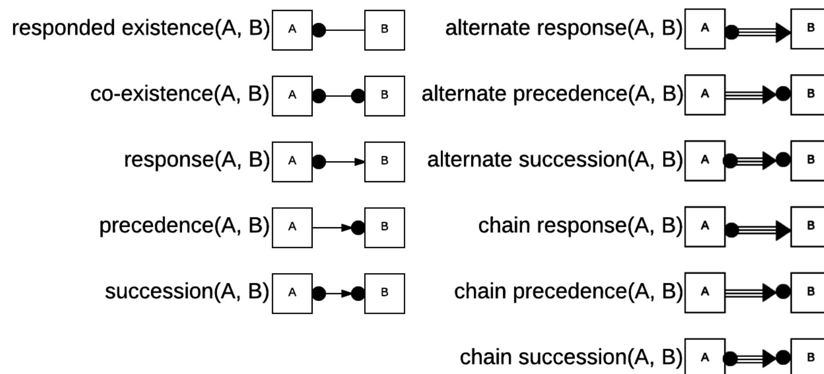


Figure 5. Notations for the relation templates (Adopted from [4])

2.3.3 Negation Templates

Negation templates can be described as negated relation templates. The *not responded existence* stipulates that activity B should never be executed (not before nor after), if activity A is executed. The *not co-existence* specifies that A and B cannot be executed together.

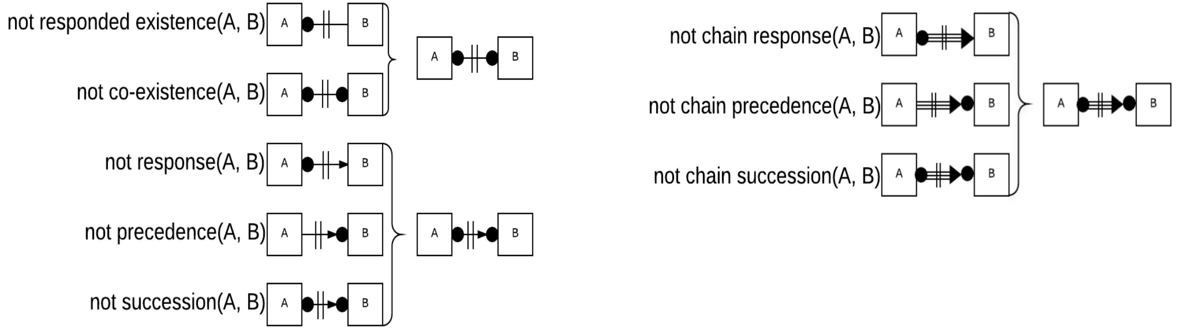


Figure 6. Notations for the negations templates (Adopted from [4])

Additionally, if event A is never executed *responded existence*(A, B) and *not responded existence*(A, B) hold, thus 'negation' is not a 'logical implication'. The *not response* template defines, that B cannot be executed after A is executed. The *not precedence* template specifies that activity A should not precede event B if B is executed. The *not succession* template, is a consolidation of templates *not response* and *not precedence*. The *not chain response* implies, that B cannot be executed right after execution of activity A. According to the template, *not chain precedence*, execution of activity A should never precede, execution of activity B. Following the same logic as above the *not chain succession* template is established by the combination of *not chain response* and *not chain precedence*. The of graphical representation of the negation templates is shown in Figure 6 [4].

Table 4. LTL formulas for relation templates (Adopted from [4])

Template name	LTL formula
responded existence(A, B) *co-existence*(A, B)	$\Diamond(A) \Rightarrow \Diamond(B)$ $\Diamond(A) \Leftrightarrow \Diamond(B)$
response(A, B) *precedence*(A, B) *succession*(A, B)	$\Box(A \Rightarrow \Diamond(B))$ $!(B)\ W\ (A)$ *response*(A, B) $\wedge$ *precedence*(A, B)
alternate response(A, B) *alternate precedence*(A, B) *alternate succession*(A, B)	*response*(A, B) $\wedge\ \Box(A \Rightarrow \bigcirc($*precedence*(B, A)$))$ *precedence*(A, B) $\wedge\ \Box(B \Rightarrow \bigcirc($*precedence*(A, B)$))$ *alternate response*(A, B) $\wedge$ *alternate precedence*(A, B)
chain response(A, B) *chain precedence*(A, B) *chain succession*(A, B)	*response*(A, B) $\wedge\ \Box(A \Rightarrow \bigcirc(B))$ *precedence*(A, B) $\wedge\ \Box(\bigcirc(B) \Rightarrow A))$

	chain response(A, B) $\wedge$ *chain precedence*(A, B)

Figure 6 shows that the templates are grouped into three equivalence classes. Thus, the eight formulas can be reduced to the three equivalent ones. This reduction is made based on the LTL formulas presented in the Table 5. The *not responded existence*(A, B) formula states that if activity A occurred, activity B cannot be executed in the rest execution of the case. As the ordering of the events is not mandatory for the *responded existence* and *co-existence* templates, thus *not responded existence*(A, B) = *not co-existence*(A, B), as a result this two templates can be satisfied by one formula. Therefore, the templates *not response*(A, B) and *not precedence*(A, B) state that if event A is executed, in the following execution of trace event B should not be executed, thus *not response*(A, B) = *not precedence*(A, B). The *not succession*(A, B) is a combination of *not response* and *not precedence*, therefore the formula *not response*(A, B) = *not precedence*(A, B) = *not succession*(A, B), as a result this three templates are equivalent to *not succession*(A, B) formula representation. As the templates *not chain response*, *not chain precedence* and *not chain succession* extend the base formulas the equality *not chain response*(A, B) = *not chain precedence*(A, B) = *not chain succession*(A, B) hold, and this three templates can be represented as a single *not chain succession*(A, B) LTL formula [4].

Table 5. LTL formulas for negation templates (Adopted from [4])

Template name	LTL formula
not responded existence(A, B) *not co-existence*(A, B)	$\Diamond(A) \Rightarrow !(\Diamond(B))$ *not responded existence*(A, B) $\wedge$ *not responded existence*(B, A)
not response(A, B) *not precedence*(A, B) *not succession*(A, B)	$\Box(A \Rightarrow !(\Diamond(B)))$ $\Box((\Diamond(B)) \Rightarrow !A)$ *not response*(A, B) $\wedge$ *not precedence*(A, B)
not chain response(A, B) *not chain precedence*(A, B) *not chain succession*(A, B)	$\Box(A \Rightarrow !(\Diamond(B)))$ $\Box((\Diamond(B)) \Rightarrow !A)$ *not chain response*(A, B) $\wedge$ *not chain precedence*(A, B)

2.4 Event Log Specification

Event logs, in real life solutions, appear in multiple forms and instantiations, and one of the most important tasks is to standardise them. Below, two different standards are described, which are supported by process mining tools, both the standards are XML based [6][7].

Figure 7 shows the general overview of the event log structure**.** An event log can contain multiple process instances without any concern of their order. The instance of an activity which occurred during the process instance is called event. Events are stored in a sequential order of their occurrence. The occurrence time of the event is stored in the timestamp element. Frequently in the event, the resource that was responsible for the execution of the given event is also specified. The resource is not limited to a person who uses the system; it also can be the system itself or some third party. Additionally, multiple other attributes can be stored in the event [6][7].

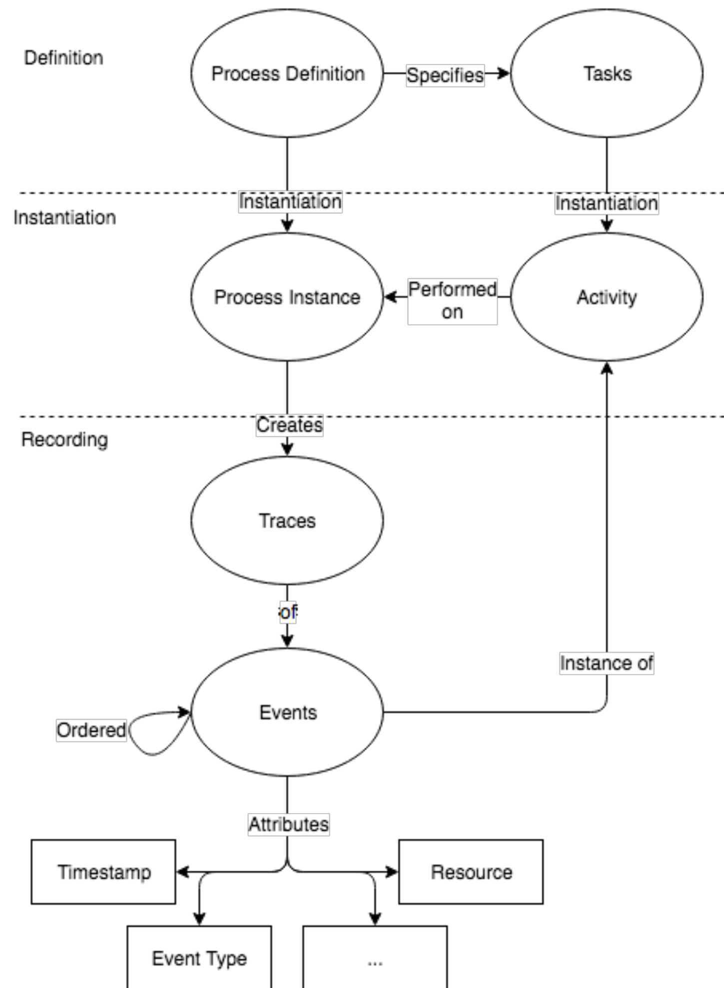


Figure 7. Event log structure (Adopted from [6])

2.4.1 Mining eXtensible Markup Language

Mining eXtensible Markup Language (MXML) was introduced in 2003. Its main goal was to standardise the log storage format and to use it as input for process mining tools [7].

Figure 8 represents the meta model of MXML log format. Table 6 depicts a partial representation of an MXML log. The *WorkflowLog* element is a root element of the log. *Source* holds the information about the organization that generated the log. MXML log can document multiple processes, represented in the *Process* element tag. Each process can hold the information regarding the multiple process execution instances, represented in the *ProcessInstance* element tag [6].

Recorded events and corresponding attributes are stored in the *AuditTrailEntry* element. The *WorkflowModelElement* represents the name of the activity. Events are atomic recordings which do not possess the sense of duration, but activity on the other hand do. Therefore, *EventType* element specifies in what state the event execution is at a given time. Example values of the event type are *start* and *complete*, specifying the beginning and the finish times of the execution. The *Timestamp* element holds the event execution date and time. The *Originator* element stores the identifier of the resource performing the activity. The *Data* element can hold some additional data attributes to provide more detailed information about the event [6].

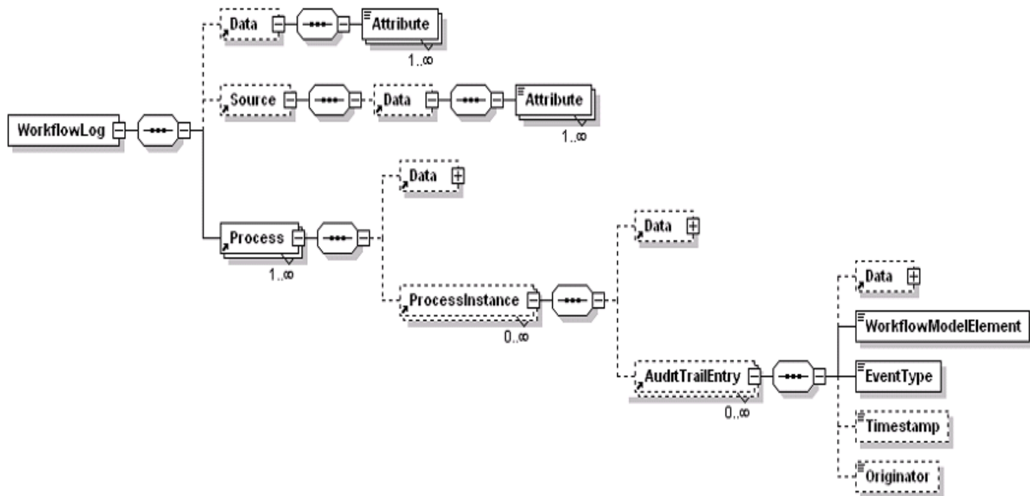


Figure 8. MXML meta model[1]

MXML has some limitations. The most severe problem is related to the semantics of MXML handling the additional information stored in the *Data* element of the event. The data type of the value is extremely hard to determine as all the values are treated as String. Additionally, initially MXML was designed with expectation to describe strictly structured processes only. Based on the experience the new eXtensible Event Stream format was introduced which is described in the next section [6].

Table 6. Partial MXML log

```
1  <?xml version="1.0" encoding="UTF-8" ?>
2  <!-- This file has been generated with the OpenXES library. It conforms -->
3  <!-- to the legacy MXML standard for log storage and management. -->
4  <!-- OpenXES library version: 1.0RC7 -->
5  <!-- OpenXES is available from http://code.deckfour.org/xes/ -->
6  <WorkflowLog>
7      <Source program="XES MXML serialization openxes.version="1.0RC7"/>
8      <Process id="Selling process">
9          <ProcessInstance id="1">
10             <AuditTrailEntry>
11                 <WorkflowModelElement>Receive Payment</WorkflowModelElement>
12                 <EventType>complete</EventType>
13                 <Timestamp>2016-11-29T11:02:00.000+01:00</Timestamp>
14                 <Originator>Hele</Originator>
15                 <Data>
16                     <Attribute name="Activity">Receive Payment</Attribute>
17                     <Attribute name="requestedBy">Mark</Attribute>
18                     <Attribute name="Costs">50</Attribute>
19                 </Data>
20             </AuditTrailEntry>
21             ...
22         </ProcessInstance>
23     </Process>
24 </WorkflowLog>
```

2.4.2 eXtensible Event Stream

eXtensible Event Stream (XES)[2] is the second attempt towards standardization of the event log format, and was designed to overcome all the shortcomings of its predecessor. The new standard should have four main characteristics. *Simplicity*, logs should be human readable, easy to produce and parse. *Flexibility*, logs from wide range of application domains should

[1] http://www.processmining.org/_media/presentations/miningmetamodelimoa2005.ppt
[2] http://www.xes-standard.org/

be apprehended. *Extensibility*, standard should be scalable for the future changes. *Expressivity*, the generalization of the log should not be for the cost of information loss, the maximum range of the information should be acquired [6][7].

Figure 9 shows the complete meta model of the XES format. Table 7 depicts a partial representation of an XES log. In XES a single occurrence of the process instance is represented as a *trace* element, the number of the *trace* elements is not limited. *trace* on its hand can hold arbitrary number of the *event* elements. The event is a portrayal of atomic activities state like in MXML [6][7].

Table 7. Partial XES log

```
1   <?xml version="1.0" encoding="UTF-8" ?>
2   <!-- This file has been generated with the OpenXES library. It conforms -->
3   <!-- to the XML serialization of the XES standard for log storage and -->
4   <!-- management. -->
5   <!-- XES standard version: 1.0 -->
6   <!-- OpenXES library version: 1.0RC7 -->
7   <!-- OpenXES is available from http://www.openxes.org/ -->
8   <log xes.version="1.0" openxes.version="1.0RC7">
9            ...
10      <trace>
11          <string key="description" value="instance with id 1"/>
12          <string key="concept:name" value="1"/>
13          <event>
14              <date key="time:timestamp" value="2016-11-29T11:02:00.000+01:00"/>
15              <string key="concept:name" value="Receive Payment"/>
16              <string key="lifecycle:transition" value="complete"/>
17              <float key="cost:total" value="50"/>
18              <string key="org:resource" value="Hele"/>
19          </event>
20                  ...
21      </trace>
22  </log>
```

The information describing either of the elements is stored in the children elements called attributes. Attribute elements are limited to types; string, integer, float, boolean, date, id, list and container. List attribute type may consist of multiple child attribute elements, can be empty as well, the child elements should have a unique key property and they are ordered. Container attribute element also consists of multiple child attribute elements and might be empty as well but on the other hand, the child attributes are not ordered. Attribute elements should have a property key which defines to what extension (if any) attribute belong. Table 8 shows the standard XES extension. Attributes can be nested in attributes as well to provide some additional information [6][7].

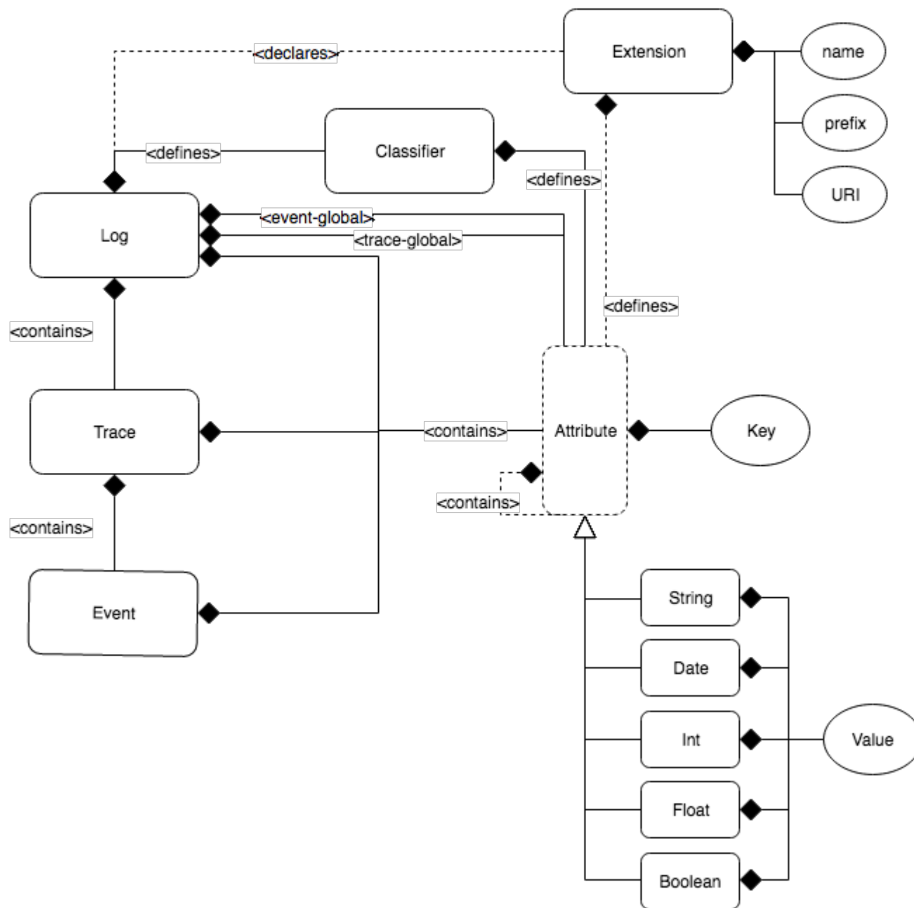


Figure 9. XES meta model (Adopted from [7])

OpenXES Java library[3] represents an implementation of the XES standard. Additionally, OpenXES can perform I/O operations on the XES format event logs and (de)serialize the MXML event log format [7].

Table 8. XES standard extensions (Adopted from [7])

Attribute Level	Key	Type	Description
Concept Extension			
log, trace, event	name	string	Stores a generally understood name for any type hierarchy element. For logs, the name attribute may store the name of the process having been executed. For traces, name attribute usually stores the case ID. For events, the name attribute represents the name of the event, e.g. the name of the executed activity represented by the event.
event	instance	string	The instance attribute is defined for events. It represents an identifier of the activity instance whose execution has generated the event.

[3] http://code.deckfour.org/xes/

Lifecycle Extension			
log	model	string	This attribute refers to the lifecycle transactional model used for all events in the log. If this attribute has a value of “standard”, the standard lifecycle transactional model of this extension is assumed.
event	transition	string	The transition attribute is defined for events, and specifies the lifecycle transition represented by each event.
Time Extension			
log, trace, event, meta	timestamp	date	The date and time, at which the event has occurred.
ID Extension			
log, trace, event, meta	id	id	Unique identifier (UUID) for an element.

3 Contribution

This chapter in Section 3.1 provides an overview of the algorithms and the tools which are part of the declarative process mining family and are adopted to be used on the cloud platform RuM[4]. RuM (from Rule Mining) is a web-application with the focus on supporting process mining tools.

Afterwards Section 3.2 describes the development process of the mining bundles. Mining bundles are the ones that produce an output file, this file can be a process model, and event log, a Microsoft Word document verbalizing the business process, etc. Mining bundles include following solutions:

- MINERful
- MINERful Simplification
- MINERful Log generation
- Declare Miner
- Declare Deviance Miner
- Declare Model Verbalization
- FLOATT

Section 3.3 discusses the approach and the implementation of the monitoring bundles. This bundles provide capabilities to monitor the running process, checks if currently executed cases satisfy or violate the business constraints. Monitor bundles include the following solutions:

- Log replayer
- MoBuConLTL
- MoBuConLDL
- OnlineAnalyzer
- Visualizer

3.1 Declarative Process Mining Tools

This section provides an overview of the algorithms and the tools which are part of the declarative process mining family and are adopted to be used on the cloud platform RuM.

3.1.1 MINERful

The MINERful algorithm is two phase algorithm for the discovery of Declare models, from event logs. The first phase prepares a knowledge base which stores extracted statistical information from the event log and is established on *MINERfulKB* concept. The second phase creates the process model by discovering the constraints through queries on the results produced by the *MINERfulKB* in the first phase. The efficiency of the algorithm is highly dependent on the quality of the input log [8].

3.1.2 Declare Miner

Declare Miner is based on a two-phase algorithm for the discovery of Declare models, from event logs. In the first phase an Apriori algorithm is used to identify frequent activities in the event log. The second phase operates on discovered frequent activity sets and produces

[4] At the moment of writing this thesis the RuM platform is in implementation phase and is not publicly accessible. The source code and general information can be found at https://github.com/FableBlaze/RuM

the candidate constraints. Finally, the candidate constraints are trimmed using *Sequence Analysis* [9] [10].

The goal is to verify if constraint holds over the traces in the log, this is achieved by investigated the positioning of activities in the trace [9].

Vacuity detection is an additional input parameter option, enabling this option the output list of constraints will include the constraints which are activated and hold frequently for given traces, if the detection is not activated trivially satisfied constraints will be included as well [9].

3.1.3 Deviance Miner

Business process deviance mining is a branch of business process mining. The primary objective of deviance mining is to provide reasons why some traces in an event log lead to a normal execution and others deviate from the standard behaviour. The deviations can be either positive or negative. Positive deviations result in higher performance metrics than expected, like achieving flawless conclusive results in a smaller amount of time, with smaller expenses or with lower resource utilization. Negative deviance, on the other hand, is when process outcome is unfavourable [11].

The deviance mining algorithm accepts as an input a log where each trace is marked as "normal" or "deviant". The output of the deviance mining algorithm includes two Declare process models, one process model includes constraints characterizing "normal" traces i.e. satisfied in "normal" traces and violated in "deviant" traces. The other one characterizing "deviant" traces. These models give feedback about the reason of the deviations in terms of Declare patterns [11].

3.1.4 MINERful Simplification

Declarative process discovery algorithms do not take into consideration constraint interaction. Hence, discovered process models may contain counter excluding constraints. Additionally, redundant constraints may be the reason for the verbose models. Declare templates are hierarchical; if the child template is satisfied the parent is satisfied as well**.** For example, *responded existence* (a, b) can be considered a parent of the *response* (a, b) constraint and therefore, it can be inferred that *response* ⊑ *responded existence*. Because of this nature the redundancy may occur in discovered declarative process model. The MINERful Simplification algorithm addresses the issues and automatically trims the discovered model to exclude repetitions and resolves interaction collisions. The algorithm uses the automata-product monoid concept to assure model consistency and removal of redundant constraints [13] [14].

3.1.5 Log Generation

The generation of event logs from a process model is crucial for testing and verifying the correctness of process discovery algorithms. Therefore**,** a tool providing this capability for declarative process discovery techniques is important. This capability consists in the event log generation algorithms from Declare process models. The algorithm interprets the constraints presented in Declare model into regular expressions and afterwards employs Finite State Automata to simulate the process. The user additionally can define as an input number of traces and length [15].

3.1.6 FLLOAT

Linear-time Temporal Logic (LTL) is appropriate for representing declarative process models. Nonetheless, the semantics of LTL is defined with infinite traces in mind. BPM system traces in most of the cases are finite, and because of that, the finite trace assumption is made. This resulted in alteration of the LTL to support a finite trace. The modified version is denoted as LTL_f (LTL on finite traces) [18]. LTL_f additionally represents one of the backbones of declarative process mining system DECLARE. To verify if the relevant trace prefixes include present execution of the LTL_f formula φ, φ is converted to a state machine [18].

Finite state machine (FSA) is a state machine which defines possible alphabet of events which can occur in the log, the states list, in which the FSA can be transitioned after an occurrence of event in the alphabet, the initial state of the FSA and the transition function [18].

The FLLOAT plugin supports satisfiability, validity and logical implication operations on a FSA. FSA is satisfiable if the list of accessible states includes a final state. FSA is valid if the negated list of accessible states does not in include a final state. Two FSA's are logical implicit if the list of accessible states of negated union of negated first automaton, and second automaton contains a final state [18].

3.1.7 MobuconLTL, MobuconLDL, Online Analyzer

These tools provide functionalities for runtime monitoring of business rules. The output should be easy to analyze thus providing the relevant information to make changes to the process model if needed timely. Currently three monitoring frameworks are supported: MobuconLTL monitors LTL – based rules. MobuconLDL monitors LDL based rules. Linear Dynamic Logic (LDL) is an extension of LTL which introduces more expressive constraints. Online Declare Analyzer to monitor multi-perspective rules [12] [16] [17].

These plugins have the following capabilities to ensure conformance checking at runtime:

- Intuitive diagnostics, i.e., detailed overview of the violated constraints and the reasons of violation.
- Continuous support, i.e., diagnostic information is provided event after a violation has occurred.
- Recovery capabilities, i.e., techniques for recovering the monitor after a violation.

The output of the plugins depicts if the currently executed trace is satisfying every constraint. The constraint can be in four states: satisfied, possibly satisfied, possibly violated and permanently violated. Satisfied, the trace is conformant with the process model. Possible satisfied, the trace is currently conformant, but the state may change if a certain set of activities are executed. Possibly violated, the trace is currently violated, but the state may change if a certain set of activities are executed. Violated, the trace violates the constraint, and it is beyond the bounds of possibility that trace will become satisfactory. The violation may be due to two reasons: First, the execution of prohibited event occurred. Second, multiple constraints are conflicting. If the reason of violation is conflict, it is not feasible in the future that all constraints become satisfied. Additionally, if trace execution is terminated possibly violated constraints turn into permanently violated. The reason of this transformation is that there are no future activity sequences executed which may satisfy the constraint [12] [16] [17].

3.1.8 Open Services Gateway Initiative (OSGi)

The Open Services Gateway Initiative (OSGi) framework is a general-purpose application development environment which provides developers with capabilities to deliver extensible and modular software. Modules are referred to as *bundles*. Figure 10, provides the general overview of the framework architecture [19].

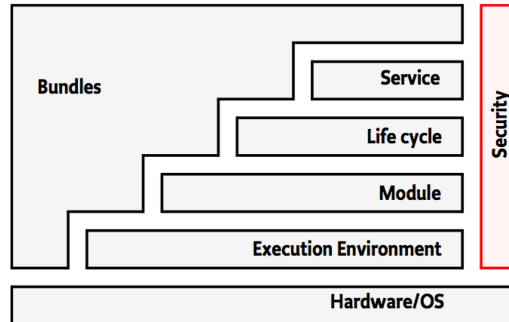


Figure 10. OSGi framework layers (Taken from [19])

The security layer is the improvement of the Java 2 security, to provide proper level of abstraction between the bundle interactions during the runtime. The module layer provides the rules for package sharing and hiding between the bundles. The Life Cycle layer defines an API which manages the bundles in the module layer. Additionally, it defines how to start and stop the bundle. The service layer simplifies the service bundle development by decoupling the specification from implementation. Thus, the bundle developer can subscribe to the service and specify the unique needs of the bundle at the run-time [19].

3.2 Mining Bundle Development

This section describes an approach which is used in order to port the solutions described in the Section 3.1 to the cloud based system RuM which has been implemented in OSGi framework to support modularity. RuM allows user to run resource extensive processes also the tasks are started on a “fire and forget” principle, meaning that the task will be carried out even if the user logs out of RuM.

As the list of the mining plugins is long and the development process of each of them is similar, this chapter will describe a specific example which covers the MINERful bundle development in detail.

3.2.1 Bundle Manifest

In order to provide the meta information to the OSGi framework a bundle should define a manifest file which is located at META-INF/MANIFEST.MF. The meta information is presented as a key value pair. Keys are the headers which provide OSGi framework with the information and the requirements needed for the the bundle to be installed into the OSGi environment. Table 10 shows the partial (The list of MINERful class path libraries is not complete) manifest file of the MINERful.

Table 9. MINERful bundle manifest file

```
1.  Manifest-Version: 1.0
2.  Bundle-Description: MINERful, an algorithm for the automated discovery of de-
    clarative process constraints
3.  Bundle-SymbolicName: MINERful
4.  Bundle-Version: 0.3.6
5.  Bundle-Name: MINERful
```

```
6.  Bundle-Vendor: University of Tartu
7.  Bundle-ManifestVersion: 1
8.  Bundle-Activator: ee.ut.cs.rum.minerful.plugin.v1.Activator
9.  Service-Component: OSGI-INF/MINERful.xml
10. Import-Package: ee.ut.cs.rum.plugins.development.interfaces,
11.  ee.ut.cs.rum.plugins.development.interfaces.factory,
12.  org.osgi.framework;version="[1.7.0,1.7.0]",
13.  org.slf4j;version="[1.7.2,1.7.2]",
14. Import-Bundle: com.google.gson;version="[2.6.2,2.6.2]"
15. Bundle-ClassPath: lib/automaton.jar,
16.  lib/combinatoricslib-2.0.jar,
17.  lib/commons-cli-1.2.jar,
18.  lib/commons-lang3-3.1.jar,
19.  lib/commons-math3-3.1.1.jar,
20.  lib/DeclareVisualizer.jar,
21.  lib/dom4j-1.6.1.jar
22.  ...
```

Manifest headers provide the following information:

- Bundle-Description: Human readable textual description of the bundle functionality.
- Bundle-SymbolicName: Unique identifier of the bundle.
- Bundle-Version: Bundle version, it is possible to have a multiple versions of the same bundle in the same environment
- Bundle-Name: Human readable bundle name
- Bundle-Activator: The class which is notified during the bundle state changes.
- Service-Component: Defines the consumed service functionality metadata
- Import-Package: Imports external dependencies required by the bundle. Version ranges can be provided as well.
- Import-Bundle: Imports the external bundle. Versions can be provided as a range
- Bundle-Classpath: Provides information about the library specific dependencies, which are not available on the platform.

Additionally, the RuM Platform requires uniqueness of the Bundle-Name and Bundle-Version combination.

3.2.2 Bundle Input Output

To install the bundle into the RuM platform the bundle should provide a JSON format description, which has a specific set of elements. The descriptions hold the information about the input parameters and output files, if any. The file is located at *resources/plugininfo.json*. Table 11 shows the general structure of the *plugininfo.json* file. All the root level elements are required.

Table 10. Structure of the *plugininfo.json*

```
1.  {
2.   "name": "Plugin name",
3.   "description": "Plugin description which will be shown to the user",
4.   "parameters": [
5.      List of the input parameters
6.    ],
7.    "outputs": [
8.      List of the output files
9.    ]
10. }
```

The Parameters element describes the list of the input parameters of the bundle. The information is used to provide a user of the system with understandable user interface, and the

system access to the values entered by the user. All the parameter objects are required to have following elements:

- internalName – The name by which the parameter will be accessible in the development environment.
- displayName – The name which will be displayed to the user.
- description – Additional information which will be displayed to the user as an additional information.
- required – Marks parameter as a mandatory or optional.
- parameterType –RuM parameter type.

At the moment of writing of this thesis RuM supported parameter types are following:

- STRING – Provides a user with a text input field, accepts string values. Additionally, string type requires two customization fields:
  - maxInputLength – Maximum allowed number of characters' user can input.
  - defaultValue – If the field is left empty, a beforehand provided default value will be used.
- INTEGER – Provides a user with a number input field, accepts numeric (without decimal places) values. Additionally, integer type requires three customization fields:
  - minValue – Minimum number a user can input.
  - maxValue – Maximum number a user can input.
  - defaultValue – If the field is left empty, a beforehand provided default value will be used.
- DOUBLE – Provides a user with a number input field, accepts numeric (with decimal places) values. Additionally, double type requires four customization fields:
  - decimalPlaces – Number of allowed decimal places
  - minValue – Minimum number a user can input.
  - maxValue – Maximum number a user can input.
  - defaultValue – If the field is left empty, a beforehand provided default value will be used.
- FILE – Provides a user with a file select dialog, the user can select from publicly available files on the server or from user's personal machine. Additionally, file type requires one customization fields:
  - inputTypes – File types which are allowed for the input parameter.
- SELECTION – Provides a user with an option input field, from which the user can select value(s) from a predefined list of values. Additionally, selection type requires three customization fields:
  - selection – Marks if the user is allowed to select more than one item at a time.
  - defaultValue – If the field is left empty, a beforehand provided default value will be used.
  - selectionItems – Contains a list of possible value items. Each item should define the following list of elements:
    - internalName – The name by which the parameter will be accessible in the development environment.
    - displayName – The name which will be displayed to the user.

- description – Additional information which will be displayed to the user as an additional information.

Table 12 shows the snippet from the MINERful *plugininfo.json* file, which presents the *parameters* element partial set.

Table 11. MINERful *plugininfo.json* parameter partial

```
1.  "parameters": [{
2.      "inputTypes": ["text", "xes"],
3.      "internalName": "logFile",
4.      "displayName": "Log File",
5.      "description": "File to be mined",
6.      "required": true,
7.  "parameterType": "FILE"
8.    },
9.    {
10.     "minValue": 0,
11.     "maxValue": 1.0,
12.     "defaultValue": 1.0,
13.     "internalName": "supportThreshold",
14.     "displayName": "Support Threshold",
15.     "description": "Minimum number of events that have to be in-
    cluded in the generated traces.",
16.     "required": false,
17.     "parameterType": "DOUBLE"
18.   },
19. ...
20. ]
```

The Outputs element describes the bundle output information. The Outputs element contains a set, where each element represents an output file. Each bundle is required to have at least one output file. Bundles can create temporary files that are not considered as an output of the bundle therefore there is no need in describing them.

Each output object consists of two elements:

- fileName – The name of the output file.
- fileTypes – The list of possible output file types. The output types are not case sensitive.

Table 12 shows the snippet from the MINERful *plugininfo.json* file, which presents the *outputs* element set.

Table 12. MINERful *plugininfo.json* describing the bundle output

```
1.  "outputs": [{
2.      "fileName": "process_model",
3.      "fileTypes": ["xml", "csv", "json"]
4.  }]
```

3.2.3 Bundle Registration into RuM

In order to register the bundle as a RuM mining bundle, a bundle should provide the implementation of the RuM *RumPluginFactory* interface, which is discussed in detail in the next section. The *RumPluginFactory* uses an OSGi declarative service (DS) functionality.

DS functionality provides capabilities to use metadata (XML) to consume or define the service. An XML description should contain the information which provides the name of the

service interface and the implementation component instance. The file is usually stored in the OSGI-INF folder and it is referenced in the manifest file with *Service-Component* header. Table 14 shows the MINERful service component implementation.

Table 13. MINERful service component

```
1. <component name="MINERfulPluginFactory">
2.   <implementation class="ee.ut.cs.rum.minerful.plugin.v1.RumPluginFacto-
   ryImpl" />
3.   <service>
4.   <provide interface="ee.ut.cs.rum.plugins.development.interfaces.RumPlugin-
   Factory" />
5.   </service>
6. </component>
```

3.2.4 Bundle lifecycle

Figure 11 shows the bundle life cycle in the OSGi framework. The entry point for the bundle is the installation. Install state specifies that the bundle has been loaded to the OSGi container but not all of the bundles' dependencies are resolved. In order to be RuM compliant the bundle should provide the implementation for the *RumPluginFactory* interface shown in the Table 14.

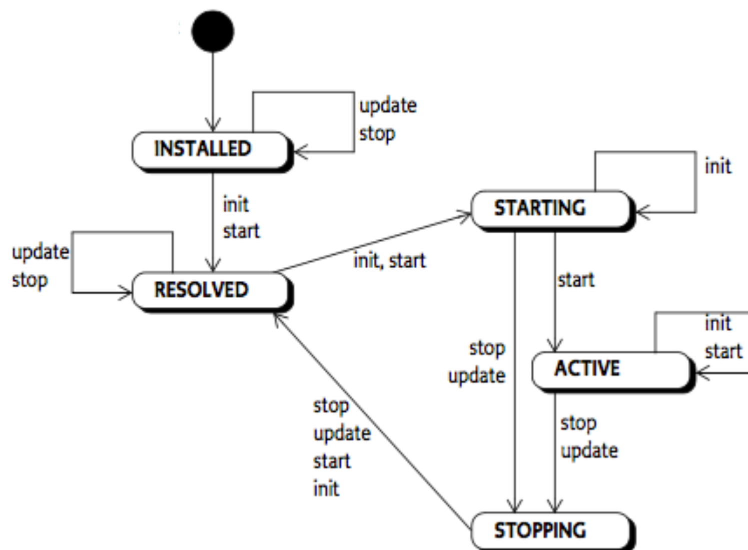


Figure 11. OSGi bundle life cycle state diagram (Taken from [19])

The method *getPluginInfoJSON* is invoked by RuM during the plugin installation. The method implementation should return the JSON string which is bundle input/output parameter specification provided in the *plugininfo.json* file.

Table 14. RumPluginFactory interface

```
1. public interface RumPluginFactory {
2.     public String getPluginInfoJSON();
3.     public RumPluginWorker createRumPluginWorker();
4. }
```

After the checks are finished and the dependencies are resolved at a class level, the bundle enters the resolve state, and is waiting to be started.

Table 16 shows the *BundleActivator* interface. The bundle should provide the implementation of the interface, so that OSGI framework can control the bundle life cycle. The *start* method is invoked in order to start a bundle. The *stop* method is called to stop an execution of the bundle and return it to the resolve state.

Table 15. OSGi BundleActivator interface[5]

```
1. public interface BundleActivator {
2.     public void start(BundleContext context);
3.     public void stop(BundleContext context);
4. }
```

As soon as the bundle is started RuM invokes a *createRumPluginWorker* method. The implementation of the method should return a new instance of the class which implements the *RumPluginWorker* interface shown in Table 17. Every time the bundle is started the new instance of the class is returned therefore making each bundle invocation unique.

Table 16. RumPluginWorker interface

```
1. public interface RumPluginWorker {
2.     public int runWork(String configuration, File outputParent);
3. }
```

The RuM will use the instance of the *RumPluginWorker* and will invoke the *runWork* method. The *runWork* method is the point where the functionality of the bundle is located and the access point of the MINERful algorithm implementation. The *runWork* method accepts two arguments: *configuration* and *outputParent*. The *configuration* string contains the JSON string produced by the *getPluginInfoJSON* method. The *runWork* method is expected to process the string and parse the parameter values provided by the user. The second parameter is the *outputParent,* is a parent directory for the outputs of the given bundle. The bundle is responsible for creating the output files and verifying that the output file type is compliant with the output types specified in the *plugininfo.json*.

3.3 Monitoring Bundle Development

This section provides an overview of the implementation of the Monitoring bundle (Monitoring bundle refers to all the tools involved in the runtime monitoring process). The bundle grants the user abilities to understand and analyse the process conformity at the run-time. In order to achieve it, the following main features should be supported by the bundle:

- Providing capabilities to replay an event log and validate it over a process model, thus providing real time verification.
- Accepting streams of events from third party providers
- Providing support for the different runtime monitoring algorithms.
- The Monitoring bundle should be able to receive information from multiple log replayers at the same time.

Figure 12 shows the general architecture of the monitoring bundle. The bundle is split into four components, to separate the concerns, and make it more scalable and flexible. At the moment bundle supports the visualization of runtime monitoring algorithms.

[5] https://osgi.org/javadoc/r4v43/core/org/osgi/framework/BundleActivator.html

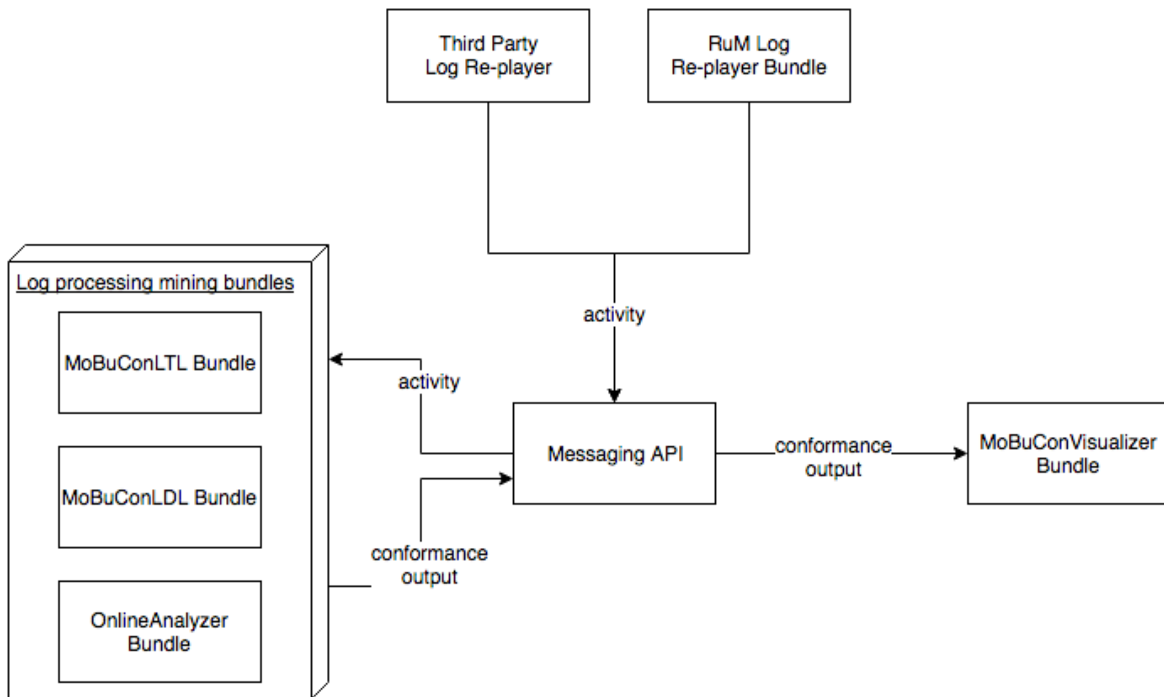


Figure 12. Monitoring bundle architectural overview

The essential part in the design is the Messaging API. To provide flexible and reliable messaging capabilities, between the monitoring components RabbitMQ[6] is selected. RabbitMQ is an open source message broker application, which provides the means of communication for the different application parts or distinct applications. The motivation for choosing RabbitMQ is that it can be used with a wide range of programming languages, thus making it easy to connect third party tools. Applications can communicate via a common protocol such as AMPQ 0-9-1, 0-9, 0-8, STOMP, MQTT etc., thus making the bundle communication protocol independent. Last but not least the message broker provides a wide range of routing capabilities. Figure 13 shows two message routing type (*queue* and *exchange*) which are used in the current implementation. *queue* routes the message from one producer to a single consumer, *exchange* allows to send messages from multiple producers to a single consumer. Producers send the information i.e. the log replayer sends the event stream. Consumers receive, i.e. a runtime verification bundle receives the event stream sent by the replayer. *queue* and *exchange* are assigned unique name.

Another important part of the bundle is the monitoring user interface. The user interface is developed via Remote Application Platform[7] (RAP). The RuM user interface is developed with RAP, therefore the choice of the user interface tool for the Monitoring bundle was limited to RAP. RAP is a powerful platform providing a rich widget toolkit to develop user interfaces. Additionally, RAP adopts the Standard Widget Toolkit[8] (SWT) API, thus the user interface development follows the principles of the SWT Java API used for desktop application development in Java programing language.

[6] https://www.rabbitmq.com/
[7] http://www.eclipse.org/rap/
[8] http://www.eclipse.org/swt/

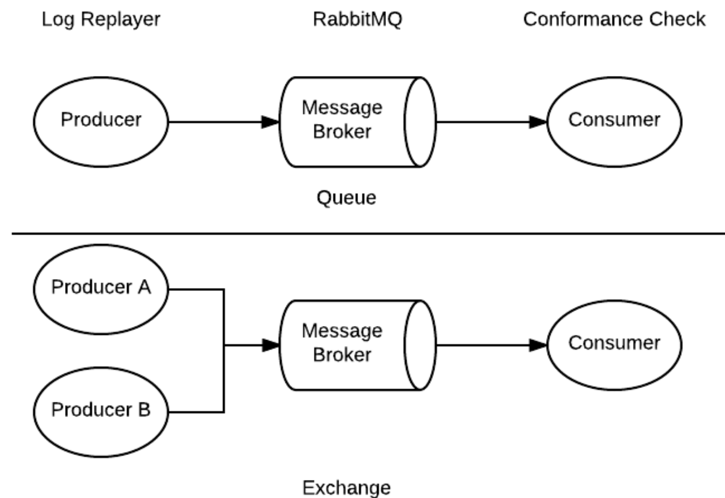


Figure 13. RabbitMQ message routing types

Log replaying is the final part of the bundle, and is implemented as a mining bundle. The essential functionality of the log re-player is to simulate the process execution from an event log which the bundle accepts as an input parameter.

The starting point of the flow is the runtime verification bundle. The bundle accepts two parameters. First, the process model, to verify conformity against. The second parameter is the routing type. The possible routing types are supported queue and exchange. To support different implementation of runtime verification tools, standardized output "Fluents string" is used. The "Fluents string" contains the identifier of the ongoing trace under analysis, the current event processed in the stream and the state of all constraints after the occurrence of the current event**.** The output of the runtime verification tool is a *.tx*t file which contains a queue name which should be used for as an input parameter for the Visualizer and queue/exchange name which is used as an input parameter for the log replayer.

The visualizer requires two parameters. First, the parent canvas where the Visualizer displays its user interface. Second, the queue name to which it connects and listens for the incoming "Fluents strings", which are processed and displayed in the designated area.

The Log re-player can be any software application which has an access to the RuM's RabbitMQ server. The input for the log re-player is a log file, a routing type which can be either queue or exchange and a queue/exchange name. The log re-player connects to the queue/exchange and send the activities in the order of their occurrence in the log.

3.3.1 Monitoring Bundle Implementation

This section discusses the implementation details of the solutions described in Section 3.3. Section 3.3.1.1 describes the implementation details of the runtime verification bundle and Section 3.3.1.2 overviews the implementation details of the Visualizer.

3.3.1.1 Runtime Verification Bundle

Runtime verification bundle implementation process fully absorbs the steps described in the Section 3.2 but in order to integrate it with the Visualizer and the message broker it introduces additional *messaging* package. The *messaging* package contains two classes:

- *RabbitMQConnection*
- *EventManager*.

RabbitMQConnection provides a connection to the message broker. As the connection instance of the class can and is encouraged to be reused, the Singleton design pattern is chosen for the class definition. Singleton design pattern ensures that there is only one instance of the class and the class provides globally available access point.

EventManager receives the messages produced by the log re-player, processes them and sends them to the visualizer. In order to receive messages from the RabbitMQ instance the class extends the *com.rabbitmq.client.DefaultConsumer* class shown in Table 18. *EventManager* instance has to explicitly call the *DefaultConsumer* constructor and pass the *Channel* instance. *Channel* is a TCP connection provider to the RabbitMQ which enables message sending and receiving abilities. To process messages from the message broker *EventManager* overrides the implementation of the *handleDelivery* method. The *handleDelivery* method is executed asynchronously every time the message is delivered.

Table 17. Partial *DefaultConsumer* class interface

```
1.  public class DefaultConsumer {
2.    public DefaultConsumer(Channel channel);
3.    public void handleDelivery(String consumerTag, Envelope enve-
    lope, AMPQ.BasicProperties properties, Byte[] body);
4.  }
```

Afterwards, the received message is validated whether it is properly formatted event and is deserialized to XES Event class instance. Next an event is passed to the runtime verification algorithm. The algorithm produces the Fluents string, which contains the information whether the event is conformant with process model or not. Additionally, the string holds the trace identifier and event name information. As a final step the Fluents string is dispatched to the message broker, which delivers it to the visualizer.

3.3.1.2 Conformance Visualizer Bundle Implementation

The visualizer provides a user interface displaying whether an ongoing trace is conformant or not with the constraints in the input process model. Figure 14 shows the mockup of visualizer user interface. The user interface window is divided into two columns: Traces and Monitor**.** Traces column, contains buttons, which identify different traces in the re-played event log. Pressing the Trace button displays the sequence of events in the trace and the corresponding state of each constraint in the process model in the Monitor column. The Monitor column provides the user with the graphical information. Each constraint contained in the process model is displayed as a separate chart. The constraint template name and involved events are displayed on top of each chart. Each column in the bar represents the event received from the log replayer, and can be in one of the four states: Satisfied - blue, Possible Satisfied - green, Possible Violated - orange and Violated - red. On a mouse hover over the column the event name is displayed as a tooltip.

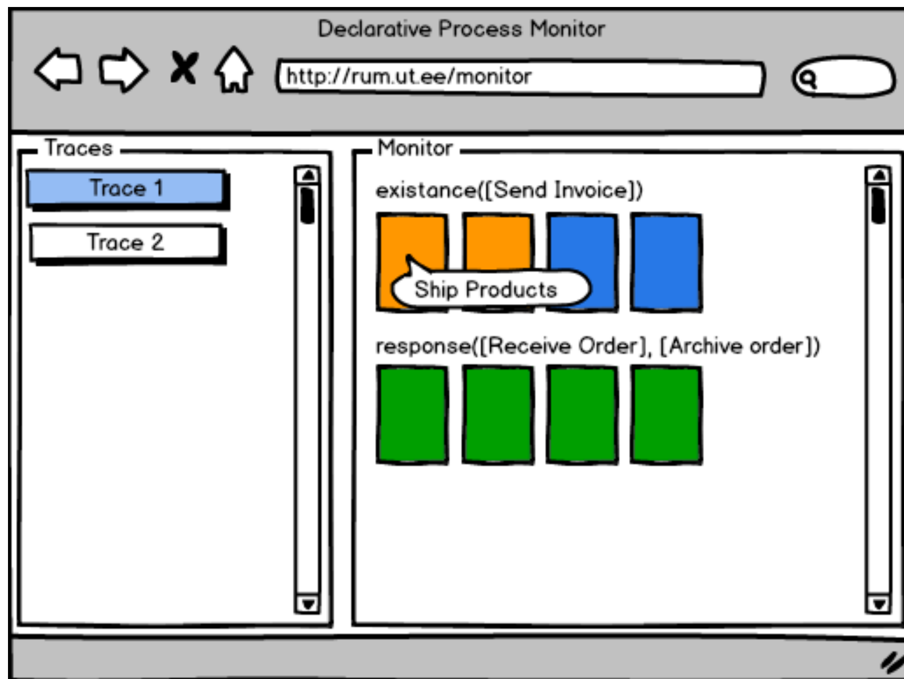


Figure 14. User interface mockup for Visualizer bundle

The implementation of the visualizer differs from the mining bundle implementation, but as visualizer is still the part of the OSGi framework it should provide the MANIFEST.MF and *BundleActivator* interface implementation.

MANIFEST.MF file is similar to the mining bundle's manifest file except that there is no need for the bundle to be registered as mining bundle and execute the algorithm with *runWork* method from *RumPluginWorker* interface, therefore *Service-Component* header is omitted.

The implementation of the *BundleActivator* interface, has a slight change as well, in order to ensure that all the connections to the RabbitMQ are closed, during the state change from Active to Resolved, the *stop* method is executed which assures that the connection is aborted properly.

To notify the RuM platform that the bundle provides the visual content, the bundle extends the *org.eclipse.swt.widgets.Composite* class, which is part of the RAP package. The *Composite* is a "container" and a graphical user interface wrapper class for the graphical elements, which are displayed to the user of the system.

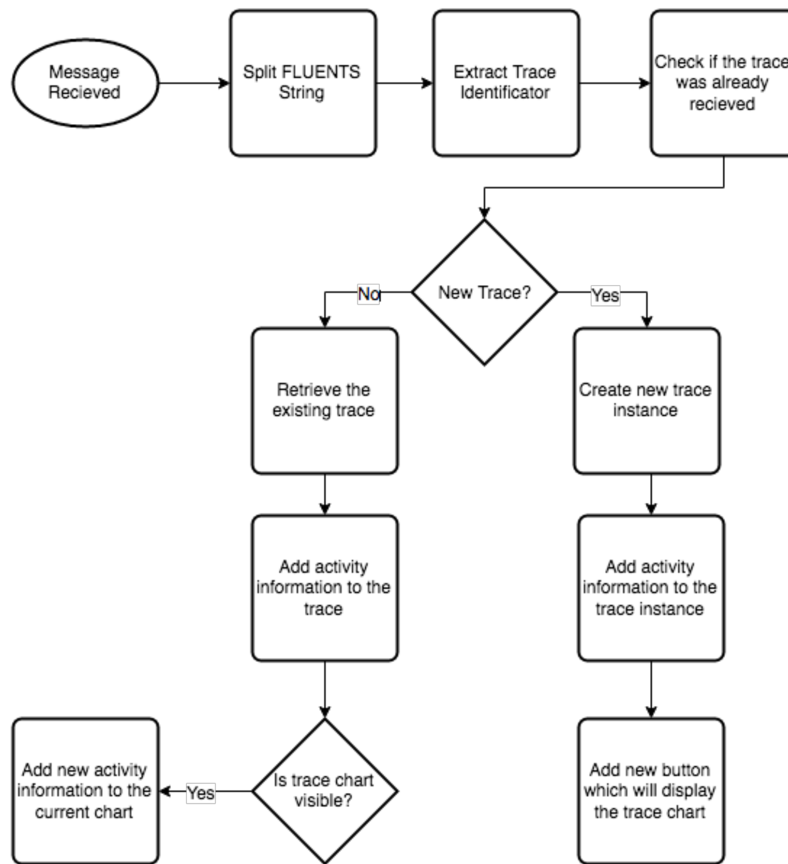


Figure 15. Fluents string processing flow chart

The bundle enters Active state as soon as the user accesses the URL of the bundle, which is decided by the RuM administrator. The bundle requires two input parameters. The first one is the instance of the *Composite* class, which will serve as a canvas area, where the bundle is allowed to display its graphical user interface. The second parameter is the RabbitMQ queue name, which is generated by the runtime verification bundle.

The Visualizer bundle connects to the RabbitMQ and subscribes to the incoming messages in the queue. The visualizer in the same way as runtime verification bundle implements the RabbitMQ connection and message delivery mechanisms.

Figure 15 is a flowchart diagram which depicts the activity conformance information retrieval process from the Fluents string. As soon as the Fluents string is received it is split on the designated delimiter, to extract trace identifier. The trace collection is checked if the trace with the same identifier already exists in the collection.

If the trace exists, it is retrieved from the collection and the current information is added to the trace. Additionally, the check is performed whether currently chart is displayed in the Monitor column of the user interface. If it is displayed the chart is checked if it belongs to the current trace. If the check is positive the chart is immediately updated with newly received activity information.

If the trace is new, the new instance of the trace is created. The trace instance is assigned the extracted trace identifier and the activity information. Furthermore, the button is created in the Traces column with the trace identifying text. The button click draws the conformity information chart for the corresponding trace.

4 Evaluation

This chapter provides examples and shows the capabilities of the implemented bundles on the cloud platform RuM.

4.1 MINERful

This section provides an overview on the use of the MINERful bundle. This bundle allows users to generate a process model from the event log. Additionally, it is possible to specify explicitly what activities are excluded, remove the redundant and inconsistent constraints and to specify the activity support and confidence threshold. The user interface of the bundle is shown in the Figure 16.

Figure 16. MINERful bundle user interface

To evaluate the bundle, the BPIC 2012 financial event log is used for process discovery. Support threshold is 0.9, Confidence 0.25 and Interest Factor 0.125. “Crop redundant and inconsistent constraints” parameter is set to “False”. The discovered model contains 291 constraints and 24 activities.

4.2 MINERful Simplification

This section provides an overview on the use of the MINERful Simplification bundle. This bundle allows a user to remove the redundancies and conflicting constraints from the process model. The bundle accepts a process model as an input. A model can be in Declare, JSON or MINERful format. It is possible to specify support, confidence and interest factor thresholds. Additionally, post processing analysis can be applied and one of the following can be chosen: Hierarchy – subsumptional constraint pruning and conflict check. Hierarchy Conflict Redundancy – Additionally to Hierarchy eliminates single-pass automata-based redundancies. Hierarchy Conflict Redundancy Double - Additionally to Hierarchy eliminates double-pass automata-based redundancies. Figure 18 shows a user interface of the MINERful Simplification bundle.

Plugin description: MINERful is a fast miner for discovering declarative process models out of logs. Logs ca

Plugin configuration:

Process Model	▾ Upload
Support Threshold	1.0
Confidence Threshold	0.0
Interest Factor Threshold	0.0
Crop Redundant and Inconsistent Constraints	False ▾
Input Model Encoding	Declare Map ▾
Post Processing Analysis Type	HIERARCHY ▾

Plugin outputs:

process_model (xml, csv, json)

Figure 17. MINERful Simplification user interface

To evaluate the MINERful Simplification bundle process model generated in Section 4.1 was used. Figure 19 shows a simplified BPIC 2012 process model.

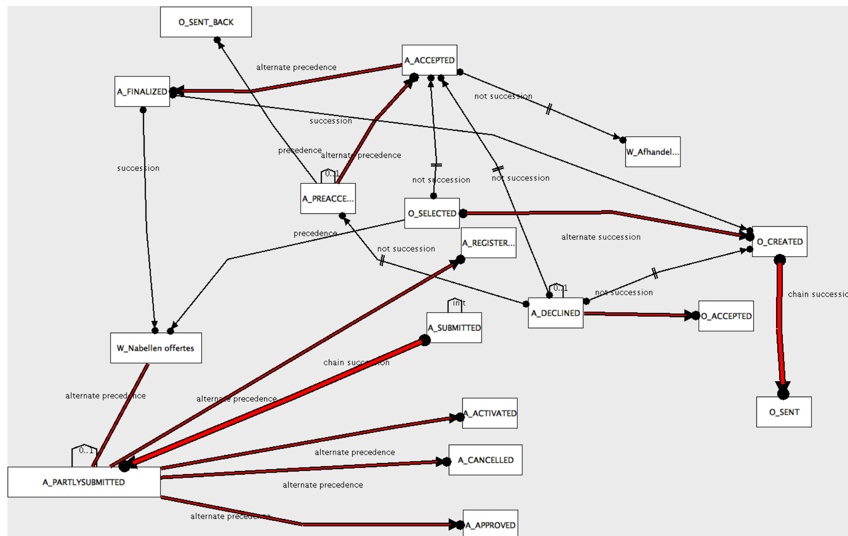


Figure 18. Discovered process model before and after simplification

4.3 MINERful Log Generation

This section provides an overview on the use of the MINERful log generator bundle. The bundle is capable of simulating the event log from the input process model. It is possible to generate event logs from Declare and JSON format process models. The type is specified as an “Input model type” parameter. Additionally, it is possible to specify minimum and maximum events per trace, as well as the maximum number of traces in the log. Generated log can be in XES, MXML or String format. Figure 20 shows the user interface of the bundle.

Plugin description: MINERful is a fast miner for discovering declarative process models out of logs. Logs can be either real or

Plugin configuration:

Process Model: BPIC2012-disco.json | Upload

Input model type: JSON

Minimum Events Per Trace: 5

Maximum Events Per Trace: 45

Traces In Log: 50

Output file encoding: XES file

Plugin outputs:

generated_log (xml)

Figure 19. User interface of Log generation bundle

To evaluate the functionality of the bundle the simple process model of fraction treatment is used, shown in the Figure 21. The generated log contains 500 traces, each trace contains 5 to 50 events, log is in XES format. To evaluate the correctness of the generated log the Declare Miner a ProM plug-in is used. Declare Miner discovers a process model from the event log. The discovered process is identical to the initial model and is shown in Figure 22.

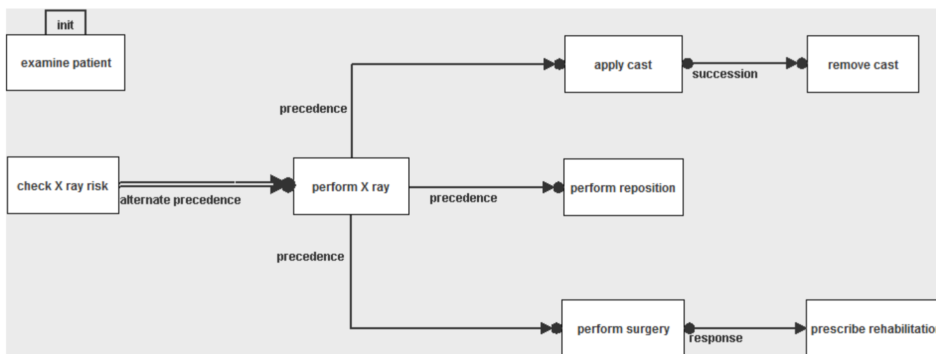


Figure 20. Fracture treatment simple process(taken from [FRACT])

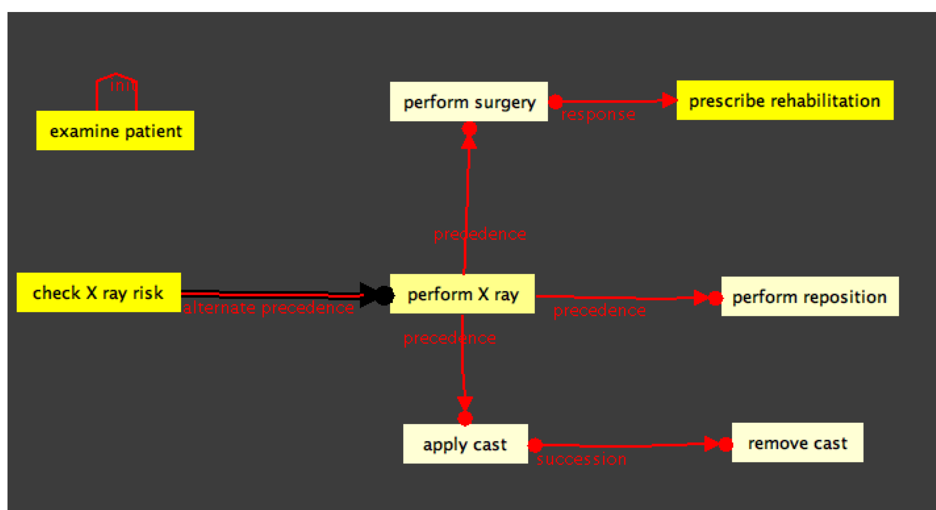


Figure 21. Discovered fracture treatment process

4.4 Declare Model Verbalization

This section provides an overview on the use of the DeclareModelVerbalization bundle. Figure 23 shows the user interface of the bundle. The bundle accepts as an input a process model in Declare format and produced the .dotx Microsoft Word document.

Plugin description: Model Verbalization, gives ability to produce human readable file, from ConDec Process Model
Plugin configuration:
Process model [ ] Upload
Plugin outputs:
verbilized_output (docx)

Figure 22. Declare Model Verbalization user interface

A description of each constraint is extracted from the input process model. Additionally, if present a constraint support, confidence and interest factor values are displayed. To evaluate the functionality of the bundle, process model discovered from the BPIC 2012 log is used. Figure 24 shows the partial result of the verbalized BPIC 2012 process model.

Declare Model Verbalization

Template name: absence2

Description: A_ACCEPTED can happen at most once.

Support: 1.000000

Confidence: 0.390693

Interest factor: 0.152641

Template name: alternate precedence

Description: A_ACCEPTED cannot happen before A_PREACCEPTED. After it happens, it can not happen before the next A_PREACCEPTED again.

Support: 1.000000

Confidence: 0.562925

Interest factor: 0.219931

Figure 23. Verbalized BPIC 2012 model

4.5 FLLOAT

This section provides an overview on the use of the FLLOAT bundle, which generates the automaton for the LTL or LDL formula. Additionally, satisfiability, validity and logical implication of the automaton can be determined by selecting the appropriate option from the bundle's user interface which is shown in the Figure 25.

Figure 24. User interface of the FLOATT bundle

To evaluate the automaton generation functionality, formulas for *response(A,B)* and *precedence(A,B)* constraints are selected shown in the Table 19. The alphabet of the events is *A; B; C; D*. Each event is “;” separated. The “Automata type” value specifies whether the input formula is LTL or LDL. The “Trim” Automaton option defines if permanently violated states should be removed from the final result or not. The output of the bundle is a *.dot* file. Furthermore, if any of satisfiability, validity or logical implication options are enabled *.txt* file is generated containing the respected information for each operation. The logical implication operations require two formulas, to state if the formulas are logical implicit, therefore the second formula field is added.

Table 18. FLOATT evaluation formulas

Constraint	**Formula**
response(A,B)	[](A -> (<>B))
precedence(A,B)	[](!B) || (!B U A)

Figure 26-A shows the visualized result of the *response(A,B)* constraint, additionally the automaton is satisfiable but not valid. Figure 26-B shows the *precedence(A,B)* with the “Trim” parameter set to *true*, generated automaton is satisfiable but not valid. Figure 26-C shows the *precedence(A,B)* with the “Trim” parameter set to *false*, generated automaton is satisfiable but not valid.

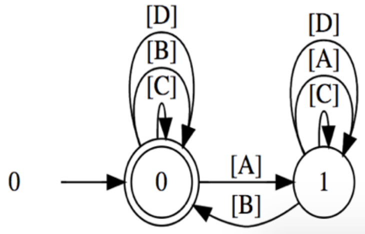


Figure 26-A. Results of automaton generation

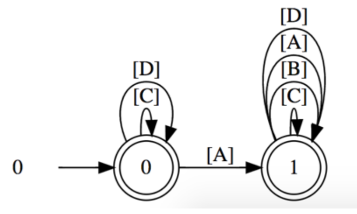


Figure 26-B. Results of automaton generation

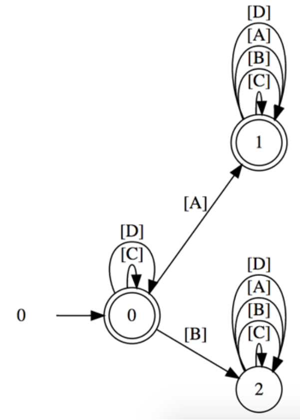


Figure 26-C. Results of automaton generation

Figure 25. Results of automaton generation

4.6 MoBuConLTL, MoBuConLDL, Online Analyser

This section provides an overview on a use of the Monitoring bundle. In order to evaluate the bundle, two runtime verification bundles are used, MoBuConLTL and Online Analyser. Figure 27 shows the order management process model. Two event logs were generated, each log contains 2 to 10 traces, and each trace contains 2 to 20 events. Four possible use case scenarios are used to demonstrate bundle capabilities.

1. One log replayer sends data to a single monitoring bundle
2. One log replayers sends data to multiple monitoring bundles
3. Multiple log replayers send data to a single monitoring bundle
4. Multiple log replayers send data to multiple monitoring bundles

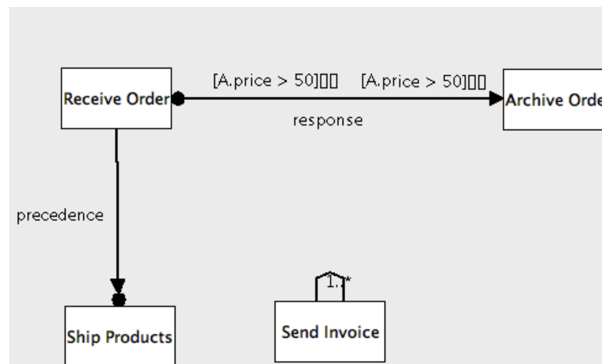


Figure 26. Order management process model

The user interface of the runtime verification bundle is shown in Figure 28. The "Messaging type" parameter can be "Queue" or "Exchange", "Queue" should only be selected for the first use case for others "Exchange", additionally to connect runtime verification bundle to already created "Exchange", "Exchange" name must be provided. A sample output of runtime verification bundle is shown in Figure 29.

Plugin description: MoBuConLTL Server plugin for RuM
Plugin configuration:
Messaging type Queue
Process model Upload
Messaging Name
Plugin outputs:
mobuconltl.out (txt)

Figure 27. Runtime verification bundle user interface

Log replayer requires the Queue/Exchange name to which it will send the event stream, routing type and an event log which will be streamed to the runtime verification bundle.

```
Queue/Exchange name for the log replayer - 0676eca6-3a91-4b21-9e25-f39ac4fbb333
Queue name for the visualiser - 24d4540b-ca37-4104-aebc-ea7997994f2a
```

Figure 28. Runtime verification bundle output

Use case 1: MoBuConLTL is used as a runtime verification bundle, the produced results are shown in Figure 30.

First occurring event in the first trace is *Archive Order*, which possible violates constraint *existence*([*Send Invoice*]) as according to the model *Send Invoice* should appear at least once during the process execution. Constraints *response*([*Receive Order*], [*Archive Order*]) and *precedence*([*Receive Order*], [*Ship Products*]), are possible satisfied.

Afterwards *Ship Products* event occurs, state of *existence* and *response* templates remain the same. On the other hand, *precedence*([*Receive Order*], [*Ship Products*]) is violated, *Ship Products* should not occur before Receive Order.

Third event in the trace is *Receive Order*, the state of the response changed to possible violated and state of *precedence* constraint is reset to satisfied. The constraint is reset to possibly violated but since since *Receive Order* occurs it becomes permanently satisfied.

Next *Send Invoice* is executed, thus satisfying the state of *existence,* does not affecting the state of the rest of constraints.

Finally, event *complete* is sent by the log replayer indicating the end of the trace, as the trace is completed and an *Archive Order* was not executed after *Receive* Order, state of the *response* changes to violated.

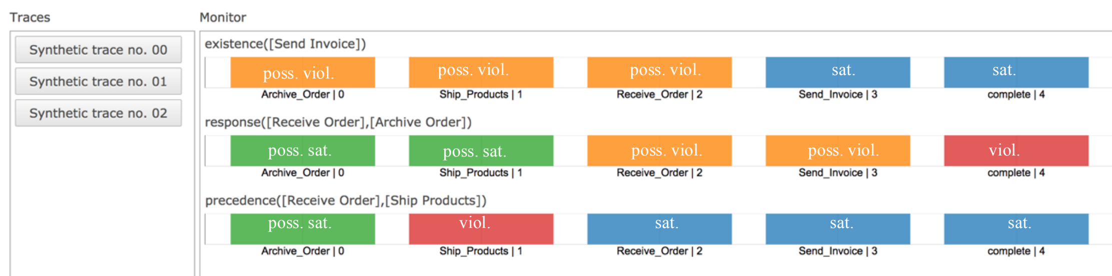


Figure 29. Runtime verification results. Use case

Use case 2: MoBuConLTL and Online Analyzer is used as a runtime verification bundle. The event log used for this use case is same as in the use case 1. The "Messaging type"

parameter in runtime verification and log replayer bundles is set to “Exchange”, to allow message delivery to multiple consumers. Produced results are shown in Figure 31.

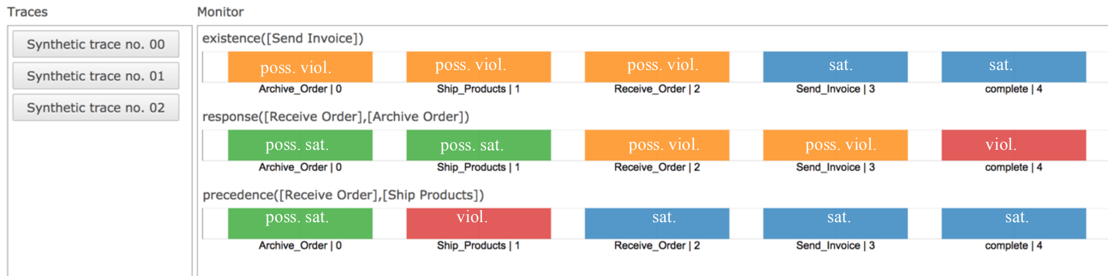


Figure 31-A. MoBuConLTL result

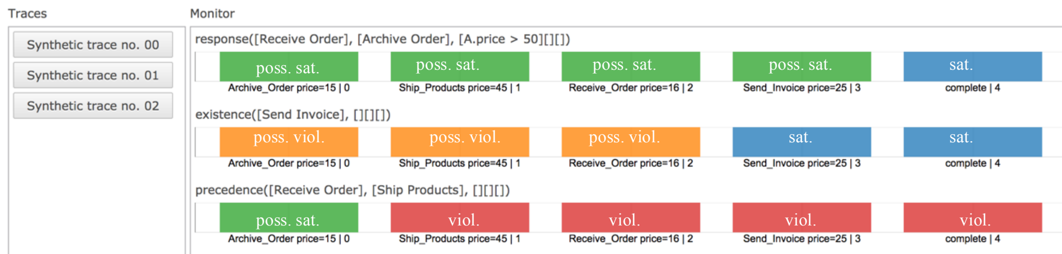


Figure 31-B. Online Analyzer result. *trace no. 00*

Figure 31-A shows the result produced by the MoBuConLTL, the results are identical as in the use case 1. Figure 31-B depicts the results of the trace no. 00 for the Online Analyzer. The events in these two traces are the same the difference is *activation* condition. This additional information is used by the Online Analyzer. Therefore, in the Figure 31-B the response constraint is never activated because when Receive Order occurs the condition A.price> 50 is not satisfied as a result constraint is satisfied after the trace is completed.

Use case 3: Two log replayers produce the event stream which is passed to a single MoBuConLTL instance, the traces are differentiated according to the trace ID specified in the event log. In this case trace ID’s are matching therefore, both log replayers are sending events of the same trace. “Messaging type” parameter as in the second use case is set to “Exchange” allowing multiple producers deliver the event stream to a single consumer. The result is shown in Figure 32.

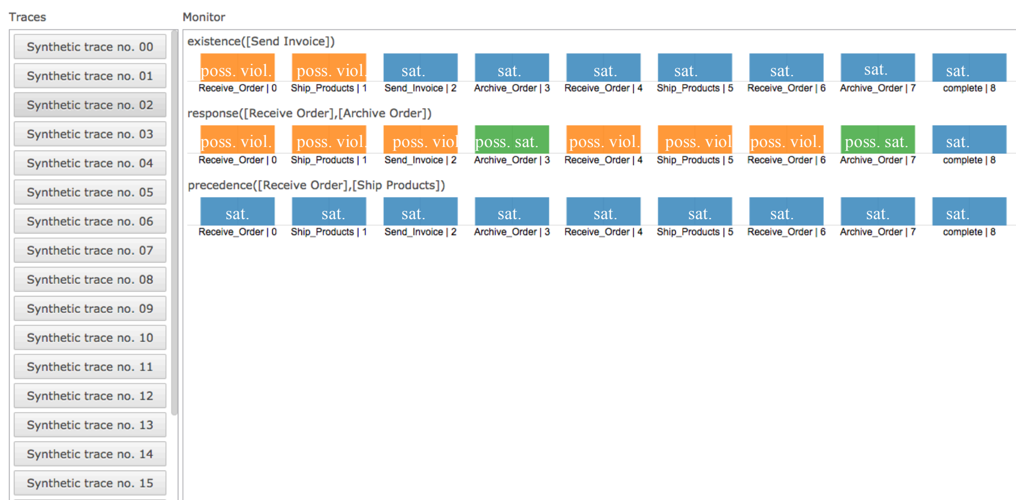


Figure 30. Runtime verification results. Use case 3

Use case 4: Two log replayers produce the event stream which is passed to MoBuConLTL and Online Analyzer instances. “Messaging type” parameter as in the second and third use cases is set to “Exchange” allowing multiple producers deliver the event stream multiple consumers. The results are shown in Figure 33.

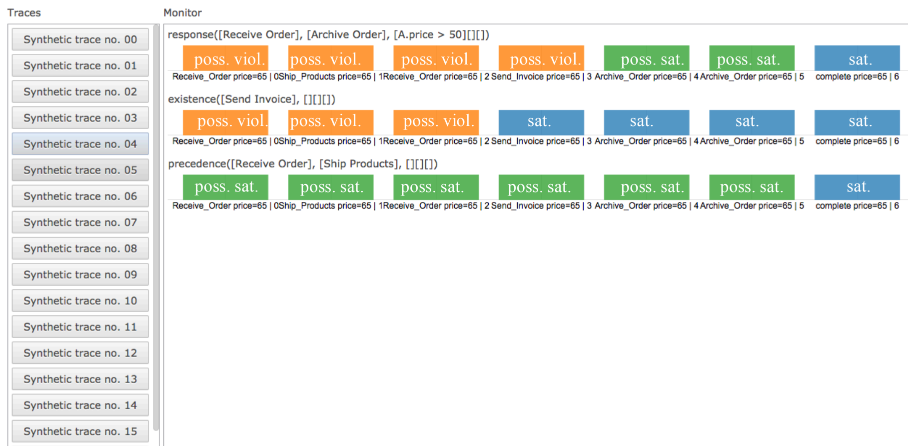


Figure 31-A. MoBuConLTL results.

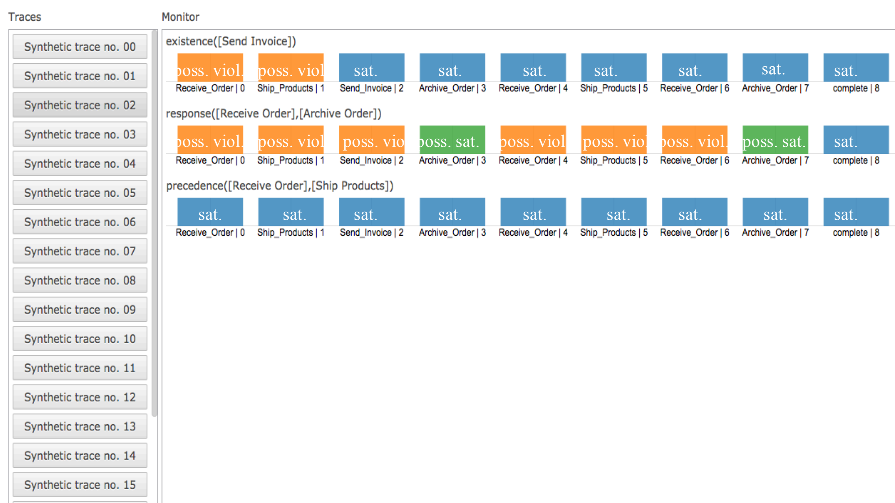


Figure 32-B. Online Analyzer result.

Figure 33. Runtime verification results. Use case 3

The demo video of the monitoring tool can found at https://youtu.be/1BHRI8B0EBQ

5 Conclusion

In this thesis we have described the Declarative process mining and currently acceptable Declarative process mining tools and their capabilities. As a result, we provided all the functionalities of the tools on the cloud platform.

We discussed the modular OSGI framework in which the cloud platform RuM is implemented, additionally we provided the description of RuM specific development requirements. As a result, we demonstrated how to implement Declarative process mining tools as OSGI bundles. We evaluated those bundles with artificial and real life logs.

Additionally, we introduced new process monitoring solution which provides functionalities for online monitoring of declarative specifications on the cloud. We briefly discussed the message routing broker RabbitMQ and its functionalities which where used for implementing the monitoring bundle. Finally, we presented Online Monitoring bundle capabilities and evaluated it using four different use case scenarios.

6 References

[1] M. Dumas, M.L. Rosa, J. Mendling, H.A. Reijers, Fundamentals of Business Process Management, Springer, Berlin, Heidelberg, 2013.

[2] W. van der Aalst, W. et al. Process mining manifesto. In Daniel, F., Barkaoui, K., & Dustdar, S. (Eds.), Business Process Management Workshops, Lect Notes Bus Inf, (Vol. 99 pp. 169–194). Berlin: Springer, 2012

[3] Claudio Di Ciccio, Fabrizio Maria Maggi, Jan Mendling. Efficient discovery of Target-Branched Declare constraints. Inf. Syst. 56, 258–283, 2016.

[4] M. Pesic. Constraint-based workflow management systems: shifting control to users. Eindhoven : Technische Universiteit Eindhoven, 2008.

[5] M. Pesic, H. Schonenberg, and W. van der Aalst, "Declare: Full Support for Loosely Structured Processes," Proc. 11th IEEE Int'l Enterprise Distributed Object Computing Conf. (EDOC 07), IEEE CS, pp. 287–300, 2007.

[6] Buijs, J.C.A.M.: Mapping Data Sources to XES in a Generic Way. Master's thesis, Eindhoven University of Technology, 2010.

[7] Christian W. Günther, Eric Verbeek. XES Standard Definition Version 2.0, Eindhoven University of Technology. 2014.

[8] Claudio Di Ciccio, Massimo Mecella. On the Discovery of Declarative Control Flows for Artful Processes. ACM Trans. Manag. Inform. Syst. 5, 4, Article 24. 2015.

[9] Taavi Kala, Fabrizio M. Maggi, Claudio Di Ciccio, Chiara Di Francescomarino. Appriori and Sequence Analysis for Discovering Declarative Process Models. Enterprise Distributed Object Computing Conf. (EDOC). 2016.

[10] R. Agrawal and R. Srikant, "Fast algorithms for mining association rules in large databases," in *VLDB*. Morgan Kaufmann. pp. 487–499. 1994.

[11] Hoang Nguyen, Marlon Dumas, Marcello La Rosa, Fabrizio Maria Maggi, Suriadi Suriadi. Business Process Deviance Mining: Review and Evaluation. ACM Transactions on Management Information Systems. 2016.

[12] Fabrizio Maria Maggia, Michael Westergaard, Marco Montali, Wil M.P. van der Aalst. Runtime Verification of LTL-Based Declarative Process Models. Springer International Publishing. 2012

[13] Claudio Di Ciccio, Fabrizio Maria Maggi, Marco Montali, Jan Mendling. Ensuring Model Consistency in Declarative Process Discovery. Springer International Publishing. 2015

[14] Claudio Di Ciccio, Massimo Mecella. On the discovery of declarative control flows for artful processes. ACM Trans. Manage. Inf. Syst. 5(4), 24:1–24:37. 2015

[15] Claudio Di Ciccio, Mario Luca Bernardi, Marta Cimitile, Fabrizio Maria Maggi. Generating Event Logs through the Simulation of Declare Models. Springer International Publishing. 2015

[16] Guiseppe De Giacomo, Riccardo De Masellis, Marco Grasso, Fabrizio Maria Maggi, Marco Montali. Monitoring Business Metaconstraints Based on LTL and LDL for Finite Traces. S. Sadiq, P. Soffer, and H. Vo ̈lzer (Eds.): BPM 2014, LNCS 8659, pp. 1–17. Springer International Publishing. 2014.

[17] Andrea Burattin, Fbarizio Maria Maggi, Alessandro Sperduti. Conformance checking based on multi-perspective declarative process models. Expert System Applications. Volume 65. pp. 194–211. 2016.

[18] Fabrizio Maria Maggi, Marco Montali, Michael Westergaar, W. van der Aalst. Monitoring Business Constraints with Linear Temporal Logic: An Approach Based on Colored Automata. S. Rinderle-Ma, F. Toumani, and K. Wolf (Eds.): BPM 2011, LNCS 6896, pp. 132–147. 2011.

[19] The OSGi Alliance OSGi Core. http://www.osgi.org. OSGi Alliance. 2014

Appendix

I. License

Non-exclusive licence to reproduce thesis and make thesis public

I, **Ilia Aphtsiauri**,

1. herewith grant the University of Tartu a free permit (non-exclusive licence) to:
   1.1. reproduce, for the purpose of preservation and making available to the public, including for addition to the DSpace digital archives until expiry of the term of validity of the copyright, and
   1.2. make available to the public via the web environment of the University of Tartu, including via the DSpace digital archives until expiry of the term of validity of the copyright,

of my thesis

Declarative Process Mining on the Cloud

supervised by **Fabrizio Maria Maggi**

2. I am aware of the fact that the author retains these rights.

3. I certify that granting the non-exclusive licence does not infringe the intellectual property rights or rights arising from the Personal Data Protection Act.

Tartu, **02.03.2017**