UNIVERSITY OF TARTU
FACULTY OF MATHEMATICS AND COMPUTER SCIENCE
Institute of Computer Science
Computer Science Curriculum

Tõnis Kasekamp

Discovering LTL-based business rules from Event Logs

Bachelor's Thesis (9 ECTS)

Supervisor(s): PhD. Fabrizio M. Maggi

Tartu 2015

Discovering LTL-based business rules from Event Logs

Abstract:

This thesis focuses on the discovery of linear temporal logic business rules from event logs. Linear temporal logic is used to describe business processes in a declarative way. The developed application LTLMiner aims at improving the performance of an existing tool TLQC used for business rule discovery. The thesis explains the solution and compares the performance between the LTLMiner and TLQC.

Keywords:

Process mining, temporal logic, LTL, model checking, query checking

Lineaarsel temporaalloogikal põhinevate ärireeglite avastamine sündmuste logidest

Lühikokkuvõte:

Käesolev bakalaureusetöö keskendub lineaarsel ajaloogikal põhinevate ärireeglite avastamisele sündmuste logidest. Lineaarset temporaalloogikat saab kasutada äriprotsesside väljendamiseks deklaratiivsel viisil. Arendatud rakendus LTLMiner üritab parandada olemasoleva rakenduse TLQC jõudlust, mida kasutatakse ärireeglite avastamiseks. Bakalaureusetöö selgitab lahenduskäiku ning võrdleb LTLMiner'i ning TLQC'i jõudlust.

Võtmesõnad:

Protsesside kaeve, temporaalloogika, LTL, mudeli kontrollimine

Table of Contents

1 Introduction

Business processes can be supported by information systems that record process executions in the so called event logs. An example of a business process execution is the following - a person has a technical issue and then he or she submits a tech support ticket. The ticket is handled by a specialist and is then solved or given to someone who can solve the issue. Every step of the troubleshooting process, from the submission of the ticket to the closing the ticket, can be logged as an event. The events typically have at least a name and a timestamp. Process mining aims at improving business processes through the analysis of event logs. By analysing the event log, valuable insight can be gained into the organisations decision making process.

The three main branches of process mining are process discovery, conformance checking and model enhancement [1]. Process discovery aims at creating a process model from an event log without any prior information. Conformance checking tries to find the discrepancies between an existing process model and an event log. In model enhancement, the goal is to improve an existing process model the information contained in an event log.

Process models can be described in a variety of languages. These languages fall into two classes – imperative and declarative languages. Imperative languages are for example Business Process Model and Notation (BPMN) [2], Event-driven Process Chains (EPC) [3], Activity Diagrams [4] and Petri Nets [5]. In imperative languages only the transitions described in the model are allowed, everything else is forbidden. Imperative models are useful for visually representing a process model, but the models generated using imperative techniques tend to be complex if the processes they describe are complex and unpredictable. Imperative models are good to be used in stable environment where processes rarely deviate from their usual pattern.

Declarative models on the other hand allow everything not explicitly prohibited by the model. Declarative models can be expressed with Temporal Logics [6], Regular Expressions [7] or Logic Programming [8]. Declarative models are more flexible and compact and are better suited for describing complex behaviour where there are several allowed paths for each process execution. Declarative models are more appropriate for processes where there are several exceptions from the main path.

This thesis will focus on discovering declarative models that are expressed by linear temporal logic (LTL) queries. To this end a tool LTLMiner was created and the performance of the LTLMiner was compared to an existing solution [9]. Experiments were done with both synthetic and real-world event logs.

2 Background

2.1 Process mining

The intent of process mining is to create, check and improve process models with the help of logged data [1]. Process models are used to analyse how organisations work and how to improve their effectiveness.

The starting point of any process mining technique is an event log. The log contains many process instances, which in turn contain many events. The event is the building block of the event log and it represents something happening in the real world. An event has a related activity and a timestamp. The event might also have information about the originator (who executed the activity) and the lifecycle transition (has the activity just started or has been completed or cancelled).

There are three types of process mining: process discovery, conformance checking and model enhancement [1]. The most common type is process discovery, which creates a process model with an event log and without any priori information. The challenge is to create the simplest model able to explain all the process behaviours found in the event log. Creating a process model is the first step towards analysing or optimising the process [10].

To check if a process model is compliant with an event log, conformance checking techniques are used. Every trace in the event log is compared against the process model. The result is an alignment between the process model and the event log [1]. Major deviations of the process model from the event log mean that the process model or the process execution must be improved.

Model enhancement is used to improve an existing process model so that it resembles more closely the situation in the real world as described by an event log. The inputs are a process model and an event log and the output is an improved process model.

A very common tool used in the field of process mining is the ProM framework[1]. ProM (Process Mining) framework provides an open source framework for tools related to process mining. There are plugins that implement different mining algorithms, as well as plugins for analysis and visualisation of event logs.

Event logs are usually stored in XES and MXML formats, which are both based on XML. XES is an event log format specifically designed for process mining [11]. The XES format contains one Log object, which in turn contains Trace objects, which in turn contain Event objects. The Log, Trace and Event objects can have multiple Attributes. The Attributes have a string-based key and the elementary Attributes can contain strings, dates, integer numbers, floating-point numbers, Boolean and id values. The Attributes may be part of a List. Furthermore, the Log object may contain Extensions that define more Attributes. An example of a XES log can be found in the Appendix.

Logs created by real-world software are rarely in these formats, therefore they need to be converted before any process mining can be done. The application developed as part of this thesis can use both XES and MXML log files.

[1] http://www.processmining.org/

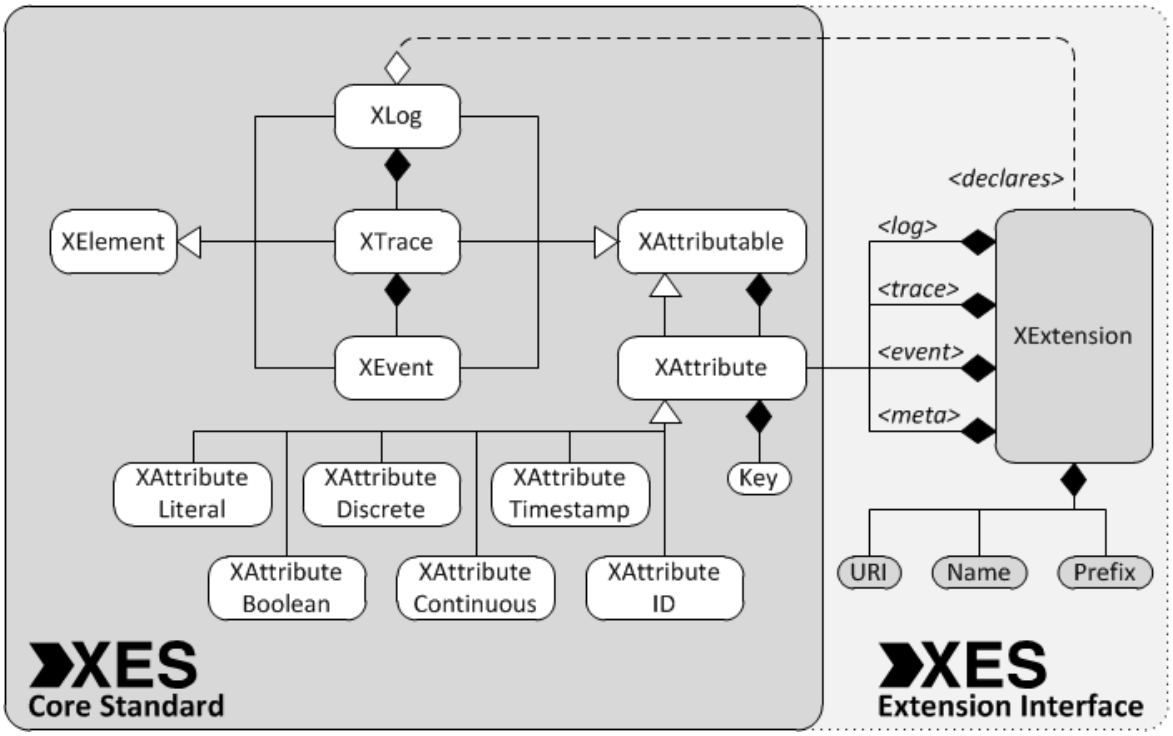


Figure 1 UML Model of the XES Standard [11]

2.2 Linear temporal logic

Linear temporal logic (LTL) is a type of modal logic with modalities referring to time with the truth values of the expression changing over time [12]. LTL is built upon propositional logic by adding five unary and two binary operators related to time.

LTL is defined as a finite set of atomic propositions *AP*, constants **true** and **false,** the logical operators ¬, ∨, ∧, →, ↔, and the temporal modal operators G (globally or always), F (finally or eventually), X (next) and U (until) [12] [9]. A set of LTL formulas over *AP* is defined as follows:

- **true** and **false** are LTL formulas;
- if p ∈ *AP* then p is a LTL formula;
- if ψ and φ are LTL formulas then ¬ψ, φ∨ ψ, φ∧ ψ, φ → ψ, φ ↔ ψ, X ψ, F ψ, G ψ, and φ U ψ are LTL formulas. [9]

In this thesis, we use LTL for finite traces.

G means that the LTL formula is true for all moments. For example if `A` is an activity, `G "A"` means that `A` must occur at any position in the trace. In the following `G` is substituted by `[]`.

X means that the LTL formula is true in the next moment. For example `X "A"` means that the next value must be `A`. In the following we use `_O` for this operator.

F means that the LTL formula is true in some future moment. `F "A"` means that `A` must be present in a trace. `A` must appear eventually. In the following we use `<>` for this operator.

U is a binary operator and it means that the first proposition must be true until the second proposition is true. In the following we use `_U` for this operator.

An example of an LTL query is `G ("A" -> (X "B"))`. The query will return true if always when event `A` is present in a trace it is followed by event `B`.

2.3 Declare

Declare is a declarative language based on LTL that can be used to describe declarative models [13]. In Declare, the declarative constraints are expressed graphically so that the user does not need to write queries with LTL. Declare classifies the constraint templates into four groups: existence, relation, negative relation and choice [13]. Every Declare template is targeted at a certain number of activities. The standard Declare templates and their LTL equivalents can be found in the Appendix.

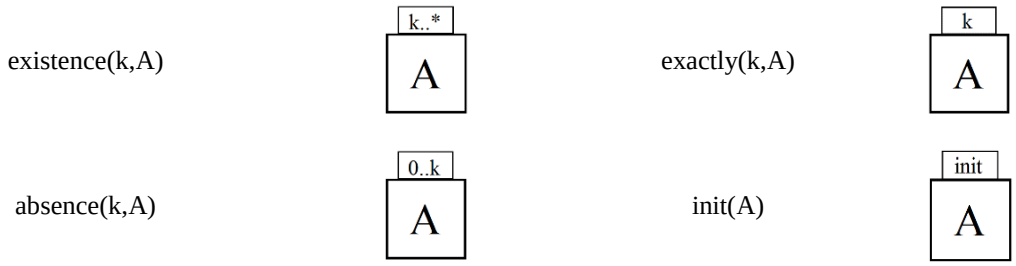


Figure 2. Existence templates [13]

Existence constraints define properties on single activities. The existence constraints are as follows: *existence(k,A)* specifies that *A* should occur at minimum *k* times in a process instance; *absence(k,A)* specifies that *A* should occur at most *k* times in a process instance; *exactly(k,A)* specifies that *A* should occur exactly *k* times in a process instance; *init(A)* specifies that *A* must be the first activity in a every trace [9] [13].

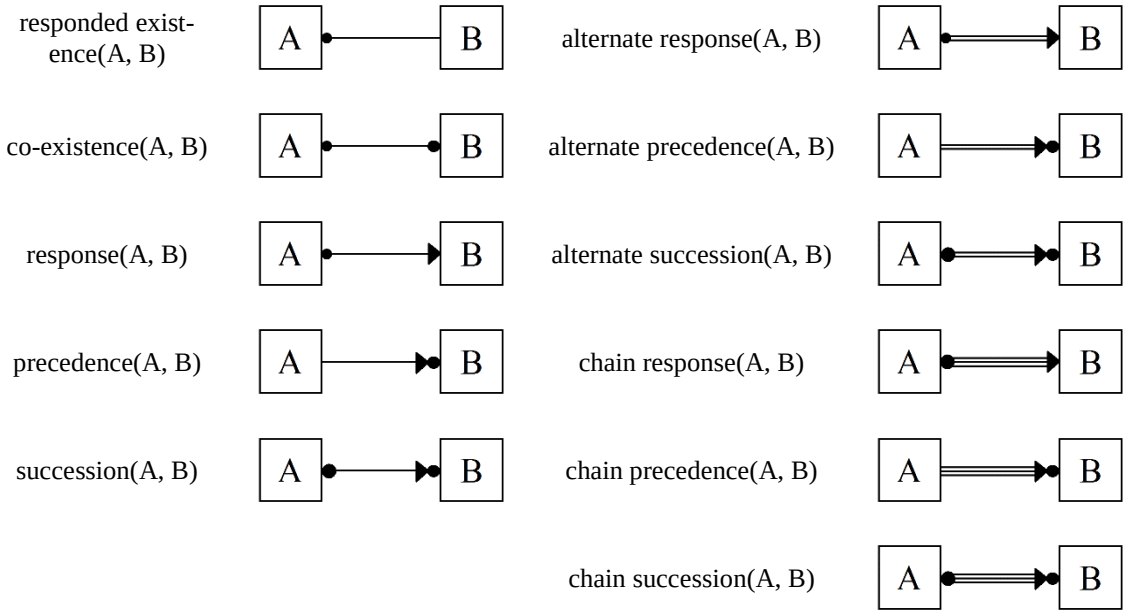


Figure 3. Relation templates [13]

Relation constraints define the relationship between two activities. Constraint *responded existence(A, B)* template means that if a trace contains activity *A* then it must also contain activity *B*. *co-existence(A, B)* specifies that if one of the activities is executed, then the other must also be executed. Constraint *response(A, B)* indicates that if a trace contains *A* then it

must eventually contain *B*. Constraint *precedence(A, B)* specifies that *B* should only be executed when *A* has been executed. Constraint *succession(A, B)* requires that the constraints *precedence* and *response* apply to *A* and *B*.

Furthermore, there are constraints *alternate response(A, B)*, *alternate precedence(A, B)*and *alternate succession(A, B)*, which require that A must be followed by *B* before *A* can occur again. The constraints *chain response(A, B)*, *chain precedence(A, B)*, *chain succession(A, B)* are the strictest as they require that *A* and *B* be executed next to each other.

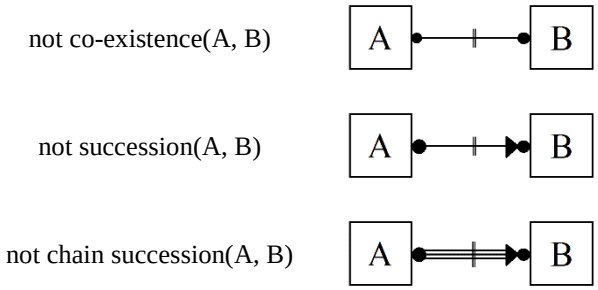


Figure 4. Negative relation templates [13]

The third group consists of the negative relation constraints for *co-existence*, *succession* and *chain succession*. The constraint *not co-existence(A, B)* specifies that *A* and *B* cannot be executed in the same trace together. The constraint *not succession(A, B)* requires that *A* can never be eventually followed by *B*. The constraint *not chain succession(A, B)* specifies that *B* cannot immediately follow *A*.

3 Solution

The aim of this thesis was to create a Java application (henceforth referred to as LTLMiner) that could automatically generate and discover LTL constraints based on an input template. The input template would be populated by using all possible combinations of activity names from the log as template arguments. The application would return only the LTL rules that applied to a pre-set percentage of traces.

We improve the performance of TLQC. TLQC[2] is a tool that was created for Temporal Logic Query Checking [9]. The user provides a Temporal Logic query with placeholders which is then populated with activity names from the event log. The program then returns all LTL formulas which are valid for the event log.

In our implementation, we use the LTLChecker as the core for the LTLMiner. The LTLChecker is a plugin for the ProM framework that is used to verify if a linear temporal logic query holds true for an event log [14]. The LTLChecker uses an extension of LTL that can be used to analyse event logs containing activities, processes, timestamps, activity originators, and other information. The basic algorithm of the LTLChecker is the following [14].

L is an event log and *F* is a formula expressed with linear temporal logic. The main checking function $check(L,F) = \forall\pi \in L(check(F,\pi,i))$ is true if the formula *F* is true for all process traces π in the event log *L*. *i* is the position of the event in the trace π and *i* is a number $(0 \leq i < |\pi|)$. For example, π_0 is the first event in the process trace.

The *check* function recursively evaluates *F* by finding the type of the main operator in *F*. First check will try to match it to an atomic expression such as a Boolean truth value. Then it will try to match it to a logic operator such as ¬, ∨, ∧, → and ↔. Finally it will try to match it to linear temporal logic operators. If a match is found the sub-expression of *F* are again evaluated by *check*.

3.1 The LTLMiner algorithm

The first step of our implementation algorithm is to find all unique activity names from the event log. As an example, we suppose that the log has three unique activities: “A”, “B” and “C”.

The second step is to find how many input arguments the LTL template (formula) needs, which we denote as *k*. The following is a standard LTL rule called co-existence. Both parameters X and Y must be present in the trace for the rule to be valid. In this case, the LTL formula has 2 input arguments, therefore *k* = 2.

```
formula co_existence( X: activity , Y: activity ) :=  {}
   (<>(activity == X) <-> <>(activity == Y));
```

The third step is to generate all possible combinations of the activity names with length k. In this case the generated combinations are <A,B>, <A,C>, <B,A>, <B,C>, <C,A>, <C,B>, we have a total of 6 combinations. By default, combinations with repeating activity names are not generated. The LTLMiner has the possibility to generate combinations with repeating activity names such as <A,A>.

The fourth step uses these combinations to create new default argument values as accepted by the LTLChecker. A copy of the input formula is created for every combination. In this case there are 6 copies.

```
formula co_existence( X: activity :  "A", Y: activity : "B") :=  {}
```

[2] https://github.com/r2im/pickaxe

```
   (<>(activity == X) <-> <>(activity == Y));
formula co_existence( X: activity : "A", Y: activity : "C") :=  {}
   (<>(activity == X) <-> <>(activity == Y));
....
```

The fifth step is to rename the created formulas. This is done because the LTLChecker does not allow multiple rules with the same name.

```
formula rule_0( X: activity :  "A", Y: activity : "B") :=  {}
   (<>(activity == X) <-> <>(activity == Y));
formula rule_1( X: activity : "A", Y: activity : "C") :=  {}
   (<>(activity == X) <-> <>(activity == Y));
....
```

The sixth step is to create a valid LTL formula file that can be used as input for the LTLChecker. Definitions are added ahead of the formulas.

```
set ate.WorkflowModelElement;
rename ate.WorkflowModelElement as activity;
formula rule_0( X: activity :  "A", Y: activity : "B") :=  {}
   (<>(activity == X) <-> <>(activity == Y));
formula rule_1( X: activity : "A", Y: activity : "C") :=  {}
   (<>(activity == X) <-> <>(activity == Y));
....
```

The seventh step is to use the event log and the LTL formula file as inputs for the LTLChecker. The LTLChecker will evaluate the LTL formula for the event log and return the percentage of traces where the formula holds.

The eighth and final step is to filter the LTLChecker output. If a threshold of 0.8 was set, then only rules which are true for 80% of the traces in the event log are returned.

3.2 Implementation

For the LTLChecker to become the core for LTLMiner it first had to be rewritten from a ProM framework plugin into a stand-alone library.

As the LTLChecker parser was very difficult to use for someone with only basic knowledge of LTL formulas, some improvements were made to the parser. Previously the parser could only accept formulas that were surrounded by a single set of parentheses, for example `<> ( activity != B);`. The parser was changed so that it could also now accept valid LTL formulas with multiple parentheses, for example `<> (((( activity != B))));`. The parser could have been improved further, but that would have been outside the scope of this thesis.

The LTLMiner was designed to be an application that would delegate the checking of LTL formulas to the LTLChecker. The LTLMiner can be thought as a wrapper for the LTLChecker that generates the rules, gives them to the LTLChecker for verification, and filters the output. As both the LTLChecker and the LTLMiner are written in Java, the application can be easily used on multiple operating systems. For parsing event logs the OpenXES library was used [11].

The number of rules generated by the LTLMiner is dependent on the number of unique activity names and the number of LTL formula input parameters. If the number of activity names is n and the number of input parameters is k, then the total number of rules generated can be expressed with the following formula.

Equation 1. Number of rules to be generated

$$\prod_{i=1}^{k}(n-k+1)$$

3.3 Using the LTLMiner

The following is a guide of how to use the LTLMiner. Let us assume that the user has an event log about ordering goods and wants to find out which two events are never in the same trace together. We also assume that the log has only one event type "complete". The full event log can be found in the appendix.

Table 1. Example of an event log

Traces
OrderGoods → ReceiveInvoice → PayInvoice → ReceiveGoods → RecordTransaction
OrderGoods → ReceiveInvoice → RejectInvoice → RecordTransaction
OrderGoods → ReceiveInvoice → PayInvoice → RejectGoods → RecordTransaction

In this case, the LTL formula to use is NotCoExistence.

```
formula notCoExistence( A: activity , B: activity ) :=  { }
   !( (<>(activity == A) /\ <>(activity == B) ));
```

The user provides the formula, the log and the percentage of how many traces the formula must satisfy. In this case the user wants only the results which apply to 100% of the traces.

```
   @Test
   public void notCoExistence() throws Exception {
      String formula = "formula notCoExistence( A: activity , B: activity ) :={}"
      +"!( (<>(activity == A) /\\ <>(activity == B) ));";
      LTLMiner miner = new LTLMiner();
      XLog log = XLogReader.openLog("src/test/resources/orderGoodsLog.xes");
      ArrayList<RuleModel> result = miner.mine(log, formula, 1.0);

      for (RuleModel rule : result) {
         System.out.println(rule.getCoverage() + " " + rule.getLtlRule());
      }}
```

Figure 5 LTLMiner usage as a JUnit test case

As there are 7 unique activity names and the formula has 2 input parameters, the total amount of formulas to check is $7 * (7-1) = 42$. How the LTLMiner does this is more thoroughly explained in the section "LTLMiner algorithm".

```
1.0 !( (<>( activity==PayInvoice ) /\ <>( activity==RejectInvoice )) )
1.0 !( (<>( activity==RejectInvoice ) /\ <>( activity==PayInvoice )) )
1.0 !( (<>( activity==RejectInvoice ) /\ <>( activity==RejectGoods )) )
1.0 !( (<>( activity==RejectInvoice ) /\ <>( activity==ReceiveGoods )) )
1.0 !( (<>( activity==RejectGoods ) /\ <>( activity==RejectInvoice )) )
1.0 !( (<>( activity==RejectGoods ) /\ <>( activity==ReceiveGoods )) )
1.0 !( (<>( activity==ReceiveGoods ) /\ <>( activity==RejectInvoice )) )
1.0 !( (<>( activity==ReceiveGoods ) /\ <>( activity==RejectGoods )) )
```

Figure 6 LTLMiner test case output

The user can see that the following events never occur together in a trace:

- PayInvoice and RejectInvoice;
- RejectInvoice and RejectGoods;
- RejectInvoice and ReceiveGoods;
- ReceiveGoods and RejectGoods.

The LTLMiner allows users to select specific activities to be used as replacements for a parameter in a rule. The LTLMiner also supports non-atomic activities and parameters can be replaced by different lifecycle transitions of the same activity.

4 The evaluation

The performance of the LTLMiner was tested against the performance of TLQC. Both programs were given as input identical LTL queries and identical event logs. Both programs generated and checked the same amount of LTL formulas.

Each test was run a total of 3 times and the average was calculated. The total time of the checking process was measured, from reading in the log until the program was finished.

Tests were run on a 64-bit Windows server with 2 cores and 12 GB of memory.

4.1 Test inputs

The event logs were generated using MINERful[3] and they were saved in the XES format [11]. The created synthetic logs differed in the total number of traces, in the number of events per trace and in the number of unique activities. A total of 18 logs were generated.

Table 2. Test log parameters

	Number of traces	Number of events per trace	Number of unique activities
Number of traces	100;200;500;700; 1000;2000;5000;7000	15	20
Number of events per trace	1000	5;10;15;20;25;30	20
Number of unique activities	1000	15	5;10;20;30;40;50

MINERful requires a Declare model with which to build event logs. The following Declare model was used:

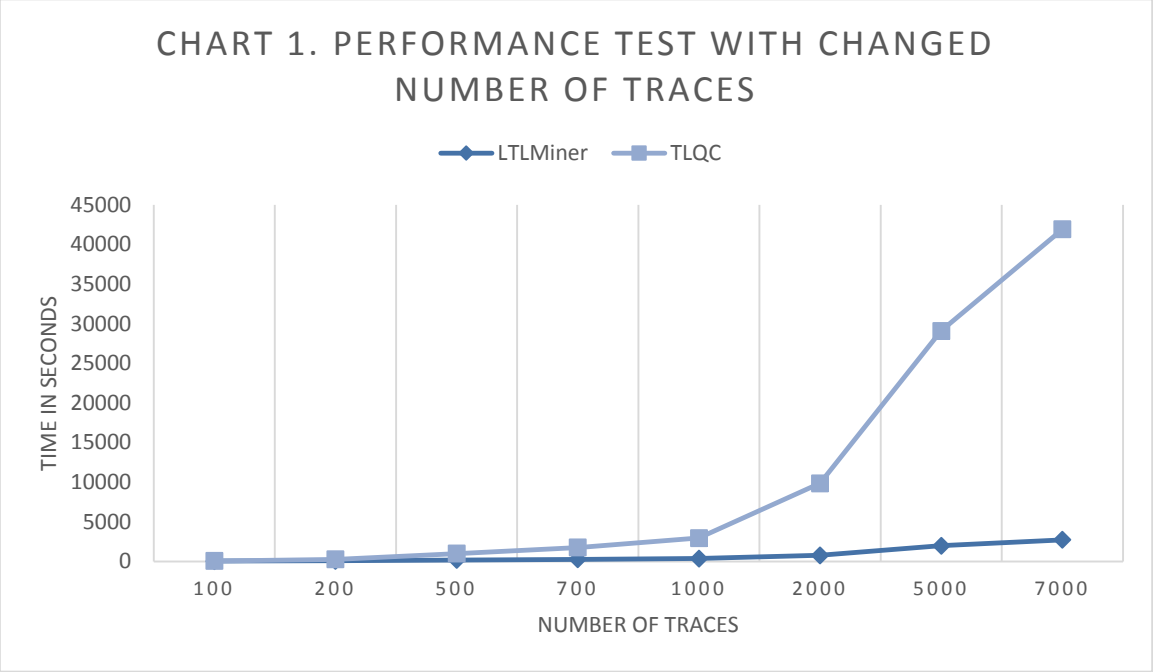
- Chain Precedence({A,B},C)
- Alternate Response (A, {B,C})
- Responded Existence(A,{B,C,D,E})
- Response(A, {B,C})
- Precedence({A,B,C,D},E)

The queries used for performance testing can be found in the Appendix.

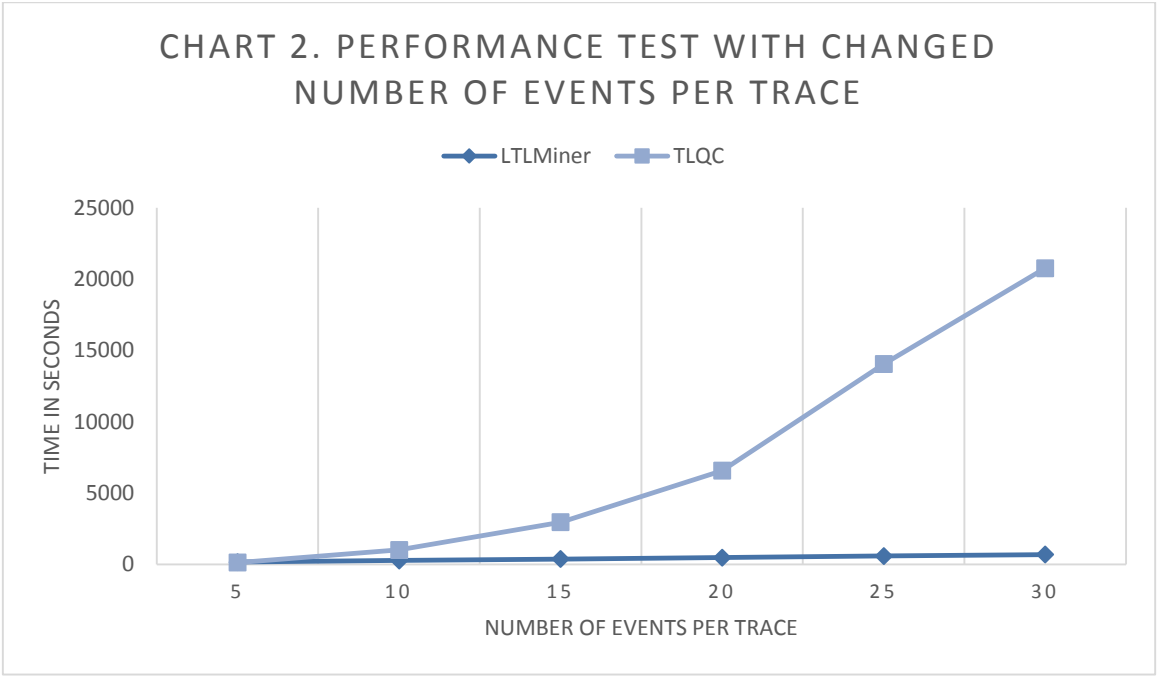
[3] https://github.com/cdc08x/MINERful

4.2 Results

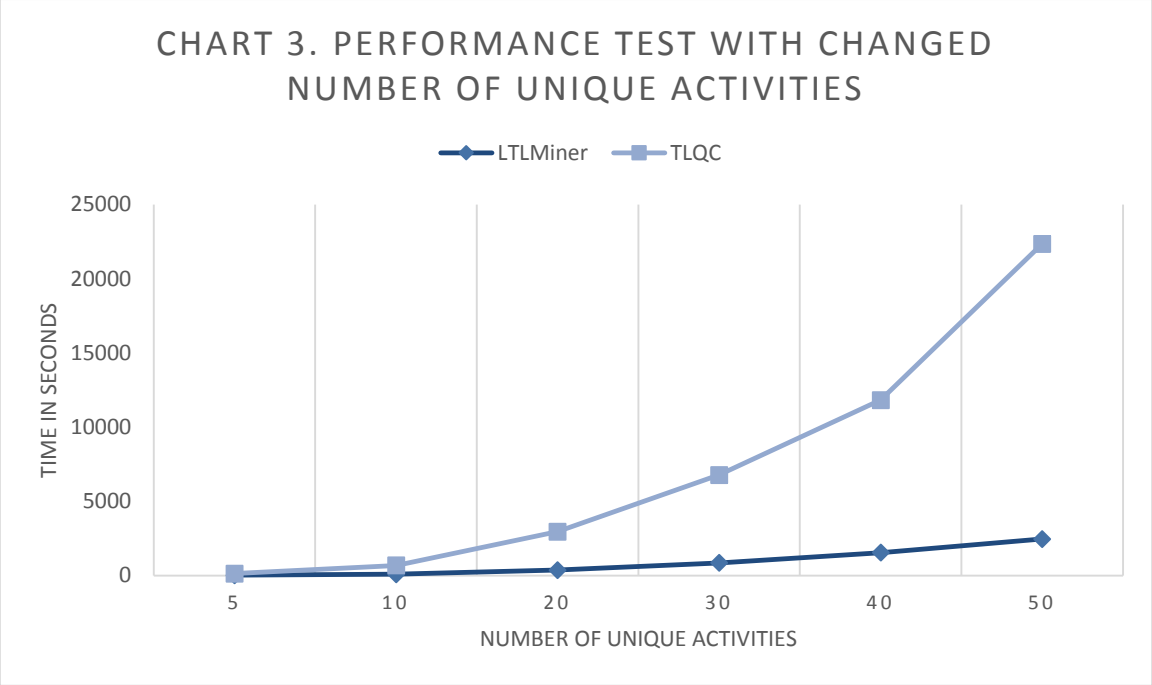
The tests in Chart 1 were run with the number of events per trace fixed at 15 and the number of unique activities fixed at 20. Only the number of traces changes.



The tests in Chart 2 were run with the number of traces fixed at 1000 and the number of unique activities fixed at 20. Only the number of events per trace changes.

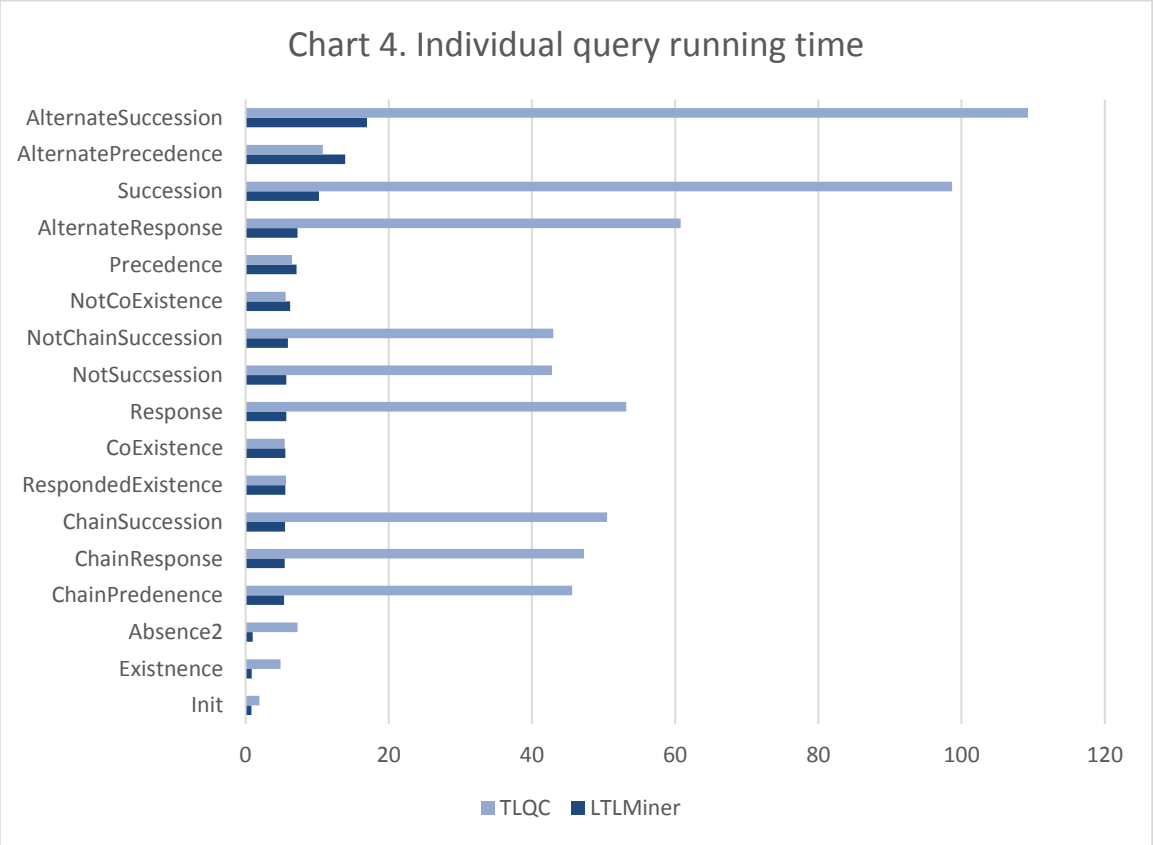


The tests in Chart 3 were run with the number of events per trace fixed at 15 and the number of traces fixed at 1000. Only the number of unique activities changes.



The tests show that TLQC has very poor performance compared to the LTLMiner. Both applications run for a similar amount of time when the test parameters are small, but there is a significant difference in the execution time when the parameters grow. For log with 7000 traces, 15 events per trace and 20 unique activities, the time taken by LTLMiner is 2729 seconds while for TLQC it is 41923 seconds.

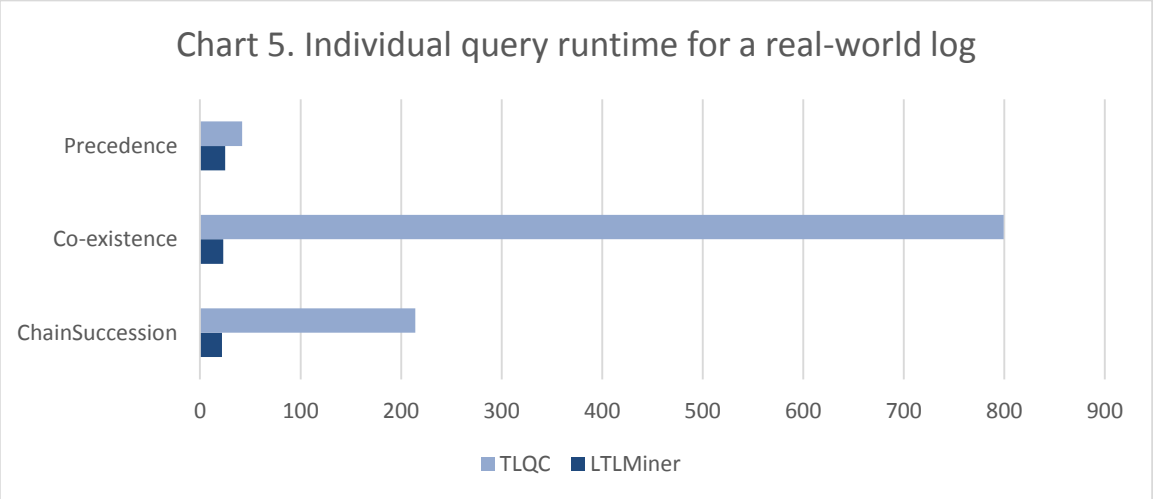
The tests in chart 4 show the performance of individual queries. The tests were executed on a Windows 8.1 PC with 8 GB of memory. The synthetic log file tested had 1000 traces, 15 events per trace and 20 unique activities.



The individual query tests show that the LTLMiner in most cases significantly faster than TLQC. However, TLQC is quicker for AlternatePrecedence, Precedence, NotCoExistence and CoExistence.

The LTLMiner and TLQC performance was also evaluated by using a real-world event log. The log is from Volvo IT Belgium and is part of the BPI Challenge 2013 [15]. The log has 7554 traces, 65533 events, 13 unique activities with 13 event types. The tests were performed on a Windows 8.1 PC with 8 GB of memory.

For every query the number of rules generated and checked for both applications was 156.



5 Conclusions

This thesis describes the application LTLMiner and how it can be used for discovering declarative models with linear temporal logic queries. The LTLMiner generates formulas based on an input template and uses the LTLChecker to check the individual formulas. During testing it was found that the LTLMiner is substantially faster at discovering models than a similar tool TLQC.

The performance of the LTLMiner could further be improved by creating multiple instances of the LTLChecker that would check the event log in parallel. As the LTLChecker requires several gigabytes of memory when running, this solution would probably be bottlenecked by the amount of memory available.

6 References

[1] "Process Mining," *Communications of the ACM,* vol. 55, no. 8, pp. 76-83, 2012.

[2] "Business Process Model and Notation," Wikipedia, the free encyclopedia, [Online]. Available: http://en.wikipedia.org/wiki/Business_Process_Model_and_Notation. [Accessed 10 May 2015].

[3] "Event-driven process chain," Wikipedia, the free encyclopedia, [Online]. Available: http://en.wikipedia.org/wiki/Event-driven_process_chain. [Accessed 10 May 2015].

[4] "Activity diagram," Wikipedia, the free encyclopedia, [Online]. Available: http://en.wikipedia.org/wiki/Activity_diagram. [Accessed 10 May 2015].

[5] "Petri net," Wikipedia, the free encyclopedia, [Online]. Available: http://en.wikipedia.org/wiki/Petri_net. [Accessed 10 May 2015].

[6] "Temporal logic," Wikipedia, the free encyclopedia, [Online]. Available: http://en.wikipedia.org/wiki/Temporal_logic. [Accessed 10 May 2015].

[7] "Regular expression," Wikipedia, the free encyclopedia, [Online]. Available: http://en.wikipedia.org/wiki/Regular_expression. [Accessed 10 May 2015].

[8] "Logic programming," [Online]. Available: http://en.wikipedia.org/wiki/Logic_programming. [Accessed 10 May 2015].

[9] M. Räim, Discovering Declarative Process Models from Event Logs, 2014.

[10] N. Gehrke and M. Werner, "Process mining," *Das Wirtschaftsstudium : wisu ; Zeitschrift für Ausbildung, Prüfung, Berufseinstieg und Fortbildung,* vol. 42, no. 7, pp. 934-943, 2013.

[11] "XES," [Online]. Available: http://www.xes-standard.org/. [Accessed 10 May 2015].

[12] "Linear temporal logic," Wikipedia, the free encyclopedia, [Online]. Available: http://en.wikipedia.org/wiki/Linear_temporal_logic. [Accessed 6 may 2015].

[13] F. M. Maggi, A. J. Mooij and W. M. v. d. Aalst, "User-guided discovery of declerative process models," in *2011 IEEE Symposium on Computational Intelligence & Data Mining (CIDM); p192-199*, 2011.

[14] W. v. d. Aalst, H. d. Beer and B. v. Dongen, "Process Mining and Verification of Properties:," in *On the Move to Meaningful Internet Systems 2005*, 2005, pp. 130-147.

[15] W. Steeman, "BPI Challenge 2013, incidents. Ghent University. Dataset," 2013. [Online]. Available: http://dx.doi.org/10.4121/uuid:500573e6-accc-4b0c-9576-aa5468b10cee.

Appendix

I. XES event log example

```
<?xml version="1.0" encoding="UTF-8" ?>
<!-- This file has been generated with the OpenXES library. It conforms -->
<!-- to the XML serialization of the XES standard for log storage and -->
<!-- management. -->
<!-- XES standard version: 1.0 -->
<!-- OpenXES library version: 1.0RC7 -->
<!-- OpenXES is available from http://www.openxes.org/ -->
<log xes.version="1.0" xes.features="nested-attributes" openxes.version="1.0RC7"
xmlns="http://www.xes-standard.org/">
   <extension name="Lifecycle" prefix="lifecycle" uri="http://www.xes-stand-
ard.org/lifecycle.xesext"/>
   <extension name="Organizational" prefix="org" uri="http://www.xes-stand-
ard.org/org.xesext"/>
   <extension name="Time" prefix="time" uri="http://www.xes-standard.org/time.xe-
sext"/>
   <extension name="Concept" prefix="concept" uri="http://www.xes-stand-
ard.org/concept.xesext"/>
   <extension name="Semantic" prefix="semantic" uri="http://www.xes-stand-
ard.org/semantic.xesext"/>
   <global scope="trace">
      <string key="concept:name" value="__INVALID__"/>
   </global>
   <global scope="event">
      <string key="concept:name" value="__INVALID__"/>
      <string key="lifecycle:transition" value="complete"/>
   </global>
   <classifier name="MXML Legacy Classifier" keys="concept:name lifecycle:transi-
tion"/>
   <classifier name="Event Name" keys="concept:name"/>
   <classifier name="Resource" keys="org:resource"/>
   <string key="source" value="Rapid Synthesizer"/>
   <string key="concept:name" value="excercise1.mxml"/>
   <string key="lifecycle:model" value="standard"/>
   <trace>
      <string key="concept:name" value="Case1"/>
      <event>
         <string key="org:resource" value="UNDEFINED"/>
         <date key="time:timestamp" value="2008-12-09T08:20:01.527+01:00"/>
         <string key="concept:name" value="OrderGoods"/>
         <string key="lifecycle:transition" value="complete"/>
      </event>
      <event>
         <string key="org:resource" value="UNDEFINED"/>
         <date key="time:timestamp" value="2008-12-09T08:21:01.527+01:00"/>
         <string key="concept:name" value="ReceiveInvoice"/>
         <string key="lifecycle:transition" value="complete"/>
      </event>
      <event>
         <string key="org:resource" value="UNDEFINED"/>
         <date key="time:timestamp" value="2008-12-09T08:22:01.527+01:00"/>
         <string key="concept:name" value="PayInvoice"/>
         <string key="lifecycle:transition" value="complete"/>
      </event>
      <event>
         <string key="org:resource" value="UNDEFINED"/>
         <date key="time:timestamp" value="2008-12-09T08:23:01.527+01:00"/>
         <string key="concept:name" value="ReceiveGoods"/>
         <string key="lifecycle:transition" value="complete"/>
      </event>
      <event>
         <string key="org:resource" value="UNDEFINED"/>
         <date key="time:timestamp" value="2008-12-09T08:23:01.527+01:00"/>
         <string key="concept:name" value="RecordTransaction"/>
```

```
        <string key="lifecycle:transition" value="complete"/>
      </event>
   </trace>
   <trace>
      <string key="concept:name" value="Case2"/>
      <event>
         <string key="org:resource" value="UNDEFINED"/>
         <date key="time:timestamp" value="2008-12-09T08:20:01.527+01:00"/>
         <string key="concept:name" value="OrderGoods"/>
         <string key="lifecycle:transition" value="complete"/>
      </event>
      <event>
         <string key="org:resource" value="UNDEFINED"/>
         <date key="time:timestamp" value="2008-12-09T08:21:01.527+01:00"/>
         <string key="concept:name" value="ReceiveInvoice"/>
         <string key="lifecycle:transition" value="complete"/>
      </event>
      <event>
         <string key="org:resource" value="UNDEFINED"/>
         <date key="time:timestamp" value="2008-12-09T08:22:01.527+01:00"/>
         <string key="concept:name" value="RejectInvoice"/>
         <string key="lifecycle:transition" value="complete"/>
      </event>
      <event>
         <string key="org:resource" value="UNDEFINED"/>
         <date key="time:timestamp" value="2008-12-09T08:23:01.527+01:00"/>
         <string key="concept:name" value="RecordTransaction"/>
         <string key="lifecycle:transition" value="complete"/>
      </event>
   </trace>   <trace>
      <string key="concept:name" value="Case3"/>
      <event>
         <string key="org:resource" value="UNDEFINED"/>
         <date key="time:timestamp" value="2008-12-09T08:20:01.527+01:00"/>
         <string key="concept:name" value="OrderGoods"/>
         <string key="lifecycle:transition" value="complete"/>
      </event>
      <event>
         <string key="org:resource" value="UNDEFINED"/>
         <date key="time:timestamp" value="2008-12-09T08:21:01.527+01:00"/>
         <string key="concept:name" value="ReceiveInvoice"/>
         <string key="lifecycle:transition" value="complete"/>
      </event>
      <event>
         <string key="org:resource" value="UNDEFINED"/>
         <date key="time:timestamp" value="2008-12-09T08:22:01.527+01:00"/>
         <string key="concept:name" value="PayInvoice"/>
         <string key="lifecycle:transition" value="complete"/>
      </event>
      <event>
         <string key="org:resource" value="UNDEFINED"/>
         <date key="time:timestamp" value="2008-12-09T08:23:01.527+01:00"/>
         <string key="concept:name" value="RejectGoods"/>
         <string key="lifecycle:transition" value="complete"/>
      </event>
      <event>
         <string key="org:resource" value="UNDEFINED"/>
         <date key="time:timestamp" value="2008-12-09T08:23:01.527+01:00"/>
         <string key="concept:name" value="RecordTransaction"/>
         <string key="lifecycle:transition" value="complete"/>
      </event>
   </trace>
</log>
```

II. Performance test queries

Declare constraint	LTL formula (LTLMiner)	LTL formula (TLQC)
Init(?x)	activity == A	?x
Existence(?x)	<>(activity == A)	F ?x
Absence2(?x)	! (<>(((activity == A) /\ _O(activity == A))))	!(F(?x & X(F?x)))
CoExistence(?x, ?y)	(<>(activity == A) <-> <>(activity == B))	(F?x) <-> (F?y)
RespondedExistence(?x, ?y)	(<>(activity == A) -> <>(activity == B))	(F?x) -> (F?y)
Response(?x, ?y)	[]((activity == A -> <>(activity == B)))	G(?x -> F?y)
Precedence(?x, ?y)	((activity != B _U activity == A) \/ [](activity != B))	([!(?y) U ?x]) | !(G?y)
Succession(?x, ?y)	[](((activity == A -> <>(activity == B)) /\ ((activity != B _U activity == A) \/ [](activity != B))))	(G(?x -> F?y)) & (([!(?y) U ?x]) | !(G?y))
AlternateResponse(?x, ?y)	[]((activity == A -> _O((activity != A _U activity == B))))	G(?x -> X([!(?x) U ?y]))
AlternatePrecedence(?x, ?y)	(((activity != B _U activity == A) \/ [](activity != B)) /\ []((activity == B -> _O(((activity != B _U activity == A) \/ [](activity != B))))))	(([!(?y) U ?x]) | !(G?y)) & (G(?y -> X(([!(?y) U ?x]) | !(G?y))))
AlternateSuccession(?x, ?y)	([]((activity == A -> _O((activity != A _U activity == B)))) /\ (((activity != B _U activity == A) \/ [](activity != B)) /\ []((activity == B -> _O(((activity != B _U activity == A) \/ [](activity != B)))))))	(G(?x -> X([!(?x) U ?y]))) & ((([!(?y) U ?x]) | !(G?y)) & (G(?y -> X(([!(?y) U ?x]) | !(G?y)))))
ChainResponse(?x, ?y)	[]((activity == A -> _O(activity == B)))	G(?x -> X(?y))
ChainPrecedence(?x, ?y)	[]((_O(activity == B) -> activity == A))	G(X(?y) -> ?x)
ChainSuccession(?x, ?y)	[]((activity == A <-> _O(activity == B)))	G(?x <-> X(?y))

NotCoExist-ence(?x, ?y)	!((<>(activity == A) /\ <>(activity == B)))	!(F?x & F?y)
NotSuccession(?x, ?y)	[]((activity == A -> !(<> (activity == B))))	G(?x -> !(F?y))
NotChainSucces-sion(?x, ?y)	[]((activity == A -> _O(activity != B)))	G(?x -> X(!(?y)))

III. Source code

Stand-alone version of LTLChecker https://bitbucket.org/TKasekamp/ltlchecker-alone

LTLMiner https://github.com/TKasekamp/LTLMiner

Performance tests for LTLMiner and TLQC https://github.com/TKasekamp/ltlminer-tests

IV. License

Non-exclusive licence to reproduce thesis and make thesis public

I, **Tõnis Kasekamp** (date of birth: 21.04.1993),

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

Discovering LTL-based business rules from Event Logs,

supervised by PhD. Fabrizio M. 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, **14.05.2015**