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

Margus Räim

Discovering Declarative Process Models from Event Logs through Temporal Logic Query Checking

Master's thesis (30 ECTS)

Supervisor: PhD. Fabrizio M. Maggi

Tartu 2014

Discovering Declarative Process Models from Event Logs through Temporal Logic Query Checking

Abstract:

This thesis will focus on the discovery of temporal logic constraints from an event log. The constraints are the description of the behavior of a business process. We will use Temporal Logic Query Checking for this purpose. A temporal logic query is a type of modal logic expression containing one or more placeholders that are checked against a transition system. The transition system is built from an event log. The result lists all possible activities that can replace the placeholders to satisfy the constraints described by the query in the log. This approach does not require (as many other approaches in the literature) negative examples as (additional) input and it provides the possibility of discovering a wider range of constraints to describe the process with respect to the existing approaches.

Keywords:

Process mining, declarative process models, temporal logic, LTL, CTL, model checking, query checking

Deklaratiivsete protsessimudelite avastamine sündmuste logist kasutades temporaalloogika päringuid

Lühikokkuvõte:

Käesolev magistritöö keskendub protsessile seatud piirangute avastamisele sündmuste logist, mida saab väljendada temporaalloogika abil. Piirangute avastamise meetodina kasutame temporaalloogika päringute kontrollimist sündmuste logi vastu. Temporaalloogika päring on modaalloogika avaldis, mis sisaldab muutujaid, mis võtavad oma väärtuse automaarpropositsioonide hulgast. Temporaalloogika päring käivitatakse vastu olekumasinat, mis on konstrueeritud sündmuste logi järgi. Päringu tulemuseks on kõik temporaalloogika avaldised, kus muutujad on asendatud kõikvõimalike automaarpropositsioonidega, mis muudavad avaldise tõeseks antud olekumasinas. See meetod ei vaja protsessi piirangute avastamiseks negatiivseid näiteid (protsessi juhtumid, mis ei tohi aset leida) sündmuste logis nagu osa avaldatuid meetodeid vajab. See meetod samuti laiendab võimalike avastatavate piirangute hulka võrreldes olemas olevate meetoditega.

Võtmesõnad:

Protsesside kaeve, deklaratiivsed mudelid, temporaalloogika, LTL. CTL, mudeli kontrollimine, päringu kontrollimine

Table of Contents

Introduction

Organizations run processes to create value for their customers. In small organizations usually one person can run all the processes involved in the daily activities. It is enough to have a pen and paper or a spreadsheet application to organize the work – all the processes are very simple or there are very few number of processes to execute. When the number of processes grow and the complexity increases, this is not possible anymore.

To help people execute complex business processes with a low amount of errors, a lot of work is performed by computers. With the help of computer software, it is also possible to automate several steps in the process, guide users to choose from a subset of activities and disallows them to start activities that are forbidden at a certain state. For example, in a process of invoicing a customer for provided services, the accountant can prepare the invoice, but cannot send it out, before somebody from the board has not approved it.

Figure 1: Simple process model

For communication purposes, it is required to convey information about processes: what are the activities involved, in which sequence activities are executed, who is doing what, etc. One way to describe a business process is to draw a diagram – creating a visual model of the process. Process models are essential to introduce the process to new people, analyze the process for efficiency, make decisions. The more precise and up to date the models are, the more useful they are.

Figure 1 is a model of a very simple process for sending out invoices. The first activity in the process is to prepare the invoice, then it must be approved and after that it can be sent out. This model is represented using the Business Process Modeling Notation (BPMN) [1]. BPMN is an imperative modeling language – only the transitions that are presented in the model are allowed, everything else is forbidden. The case, when a person prepares an invoice and sends it out without getting an approval, is illegal with respect to the model in Figure 1. Other such

modeling languages are, but not limited to, Event-driven Process Chains [2] (EPC), Activity diagrams [3] and Petri net [4].

Another way to describe a process model is to present the restrictions and allow everything that is not in the model opposed to imperative models. These models are classified as declarative process models – the activity flow is not implicitly defined.

To help organizations to make better decisions about their internal processes, the analysts must evaluate the effectiveness of the processes employed. The more the process model reflects the real situation the better decisions can be made. There are several ways of generating a process model, but two “extremes” are: interviewing people involved in the process or getting information from the event logs generated by the application supporting the process. The first approach could produce models that are far from the reality, because people tend to have a subjective point of view about how the process is executed. The second approach produces models that are closer to the reality (if the quality of available data is good enough). Usually mixture of activities are involved during the process model discovery [5].

Discovering process models from event logs is one of the three main process mining branches. The event log (used as input) is produced by a business application and the output is the discovered process model [5]. For example, when a company sells a product, all activities (e.g. “Receive Order”, “Receive Payment”, “Ship Products”, “Send Invoice” and “Archive Order”) are recorded by an information system. Each time somebody enters data to the system an entry to the event log is created. At some point in time the log is archived and it can be analyzed with process mining tools to produce a process model.

To advertise the field of process mining, a manifesto [6] has been written to describe guidelines to developers and research challenges. The manifesto is an attempt to get all people involved in process mining acknowledged about the efforts done so far and to continue further research with the same goals in mind.

Traditional process discovery techniques are based on imperative process models. Imperative models better describe well structured processes, where each activity has few transitions between activities. However, in flexible processes a person may freely choose in which order activities are performed. For example a doctor in a hospital can choose from variety of treatments to be provided to the patient.

When the number of possible sequences of activities grows in a flexible process, the

conventional imperative models start to become unreadable and produce the so-called spaghetti-like models (see Figure 2: Imperative model of a flexible process).

Flexible processes can be effectively described using declarative models. In the case of the hospital example, the declarative model will contain all possible treatments and constraints to restrict the use of some of them. A declarative model can be defined using Temporal Logics [7], Regular Expressions [8] or Logic Programming [9].

This thesis will try to prove that through Temporal Logic Query Checking it is possible to widen the amount of possible temporal logic constraints that can be discovered with respect to the ones that can be discovered with the approaches developed so far [10], [11] and [12] thus being able to produce models with a higher expressive power. The existing tools, for discovering declarative process models, have the limitation of discovering only a predefined (hard-coded) set of constraints. Adding the possibility of discovering new constraints requires development effort. Through user-defined queries, the constraints are constructed at run-time based on the query and the model. Another advantage is that this approach (differently from others, like the one proposed in [12]) does not need negative examples and they can easily be used on logs (that typically only contain positive information, i.e., how things have been done).

As proof of concept of our approach, a query checking algorithm similar to the one presented in [13] is implemented, but optimized to discover process models. The user provides a Temporal Logic query (i.e., a Temporal Logic formula including one or two placeholders) and, then, this query is evaluated against a model (in our case the event log) to replace the placeholders with Temporal Logic sub-formulas. The result is a Temporal Logic formula that is valid on the log.

The main research question addressed in this thesis is:

> ***RQ1: Can Temporal Logic Query Checking improve the discovery of declarative process models from event logs?***

The definition of ‘improve’ is intentionally not given. In fact, RQ1 consists of two questions. (a) the improvement could be in terms of performance, i.e., the developed tool can produce models faster than other existing tools, but more interesting is (b) the tool can make more insights of the process to be discovered.

Another goal is also to improve the usability of existing solutions, but this is out of the scope of the thesis.

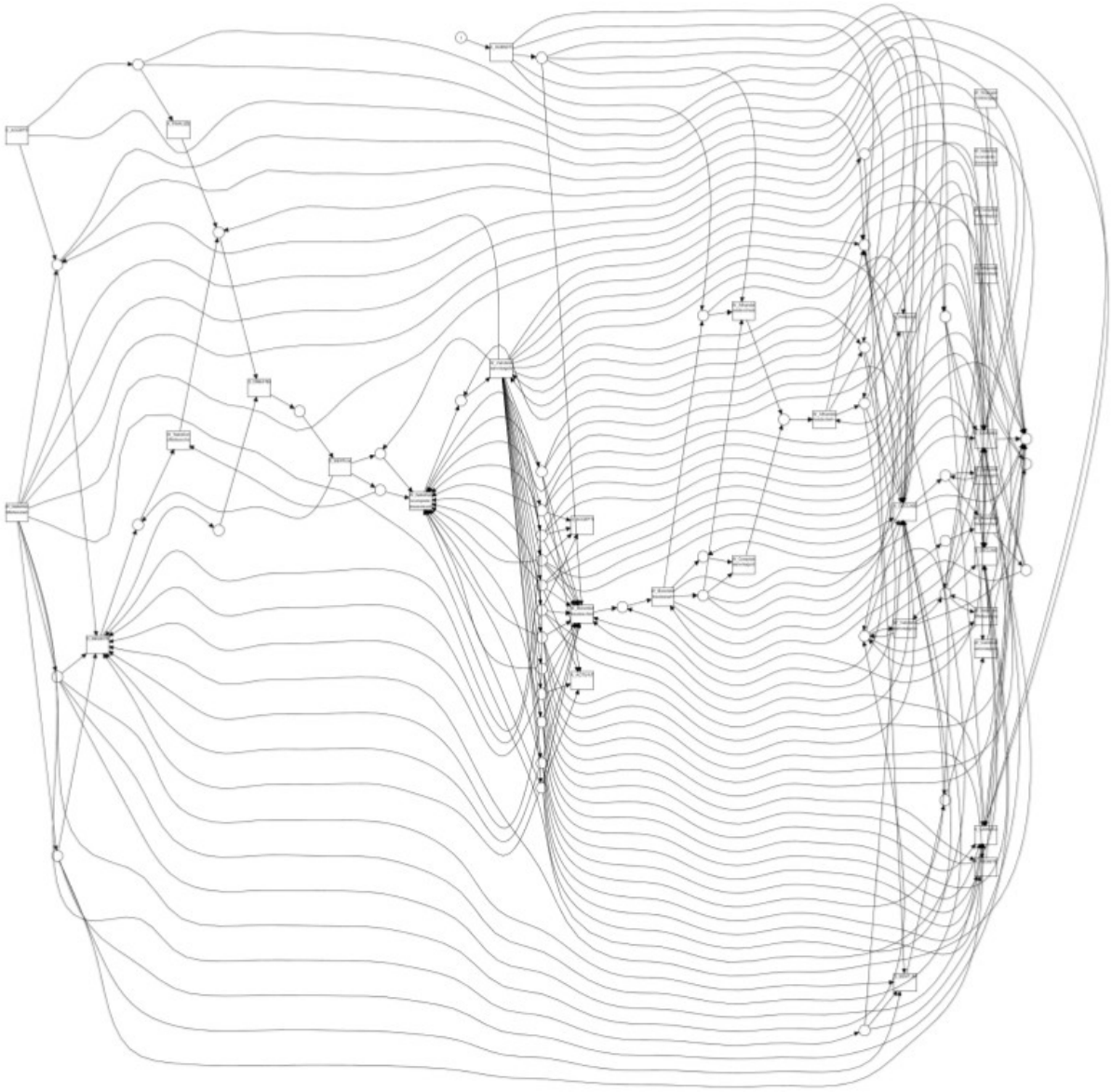


Figure 2: Imperative model of a flexible process

In summary the improvement is expected to come from the fact that this approach does not need negative examples and can use much wider range of constraints with respect to the state of the art.

Besides the main research question the second question is:

RQ2: Does the tool have a business value?

The question is implicitly answered, when there is evidence, that there is significant improvement of existing tools. But this is still theoretical conclusion. The real answer is to understand whether some companies or organizations are willing to invest money for using this approach.

The business value of the solution for companies and organizations would be to use their existing log files to model through our developed tool, what is really happening in the company. Upon the generated models concrete decisions can be made. Also the tool would raise questions to the analysts about the process.

A study [14] was conducted to evaluate whether declarative process modeling is applicable in industry. The study starts by reflecting doubts, whether it can be used outside academic world. During the experiment, group of specialist, who were experts in process modeling, where given different tasks using declarative process modeling tools. Overall the participants agreed that there is the benefit of using declarative process models, but not in every case. The conclusion was that industry would benefit the most from hybrid systems.

The proof of concept of the approach has been implemented as a standalone software package. The application takes an event log and a list of queries as input. The log is converted to a transition system and each query is checked against it. The result is a list of LTL constraints, which hold in the log.

The thesis continues with the overview of process mining and it connection to temporal logic; description of the solutions; evaluation of developed application and a case study. Finally related work is and a conclusion is discussed.

Background

Business process models are a good resource for convening information about processes in an organization or between organizations. Different stakeholders are using models for different purposes – executives use them for decision making, human resources to introduce a process to new people and system developers during implementation. One property all the stakeholders alike are expecting from process models is that the process models must reflect the reality as accurately as possible.

Process Mining

Process mining is a field between process modeling and analysis on one side and data mining and machine learning on the other. Usually the process mining starts with an event log. Event log is just a set of process instances and a process instance is a trace of events. Event is an atomic action with at least a name and a time stamp. The techniques to process an event log come from the field of data mining, but no existing algorithm was sufficient to discover the relations between two activities in a process model. From data mining the main ideas used are the apriori algorithm for discovering frequent items and association rules, sequential pattern discovery [15] and episode mining [16]. The main drawback is that these algorithms are good detecting local patterns in the log, but are unable to generate an overall process model from it.

Process mining consists of three major parts: model discovery, conformance checking and enhancing existing models. The most challenging is to discover the process model, which the traces belongs to – from sequences of events re-engineer the process that produced the log [5]. This is similar to a machine learning problem. Process model discovery algorithms get an event log as an input and produces a process model. The model is not restricted to a specific modeling language; any of the available languages can be used to graphically represent the process model. The most common outputs are a Petri nets, Workflow nets, BPMN, EPCs, BPEL. Less common is to have a declarative model as output. Many algorithms produce Petri net diagrams, which can be converted to any other notation [5].

In conformance checking an event log and a process model are used as inputs. The event log is compared against an existing process model. Each trace is replayed on the given process model. The result is a report on the amount of the traces aligned with the model and description of how other traces deviate respect to the given model [5].

Model enhancement also takes an event log and an existing model as inputs, but the result is, in this case, a model enriched with additional data. For example, when it is possible to extract the cost or duration of activities, then the averages of these attributes can be added to the model.

Using process mining algorithms, one can describe the data at hand and also predict the unseen data. So process mining is not used only on off-line data processing, but can be used on on-line data. With the help of machine learning, process mining can be used to support user to make decisions, estimate the outcome of a process case or detect deviations from a given process model.

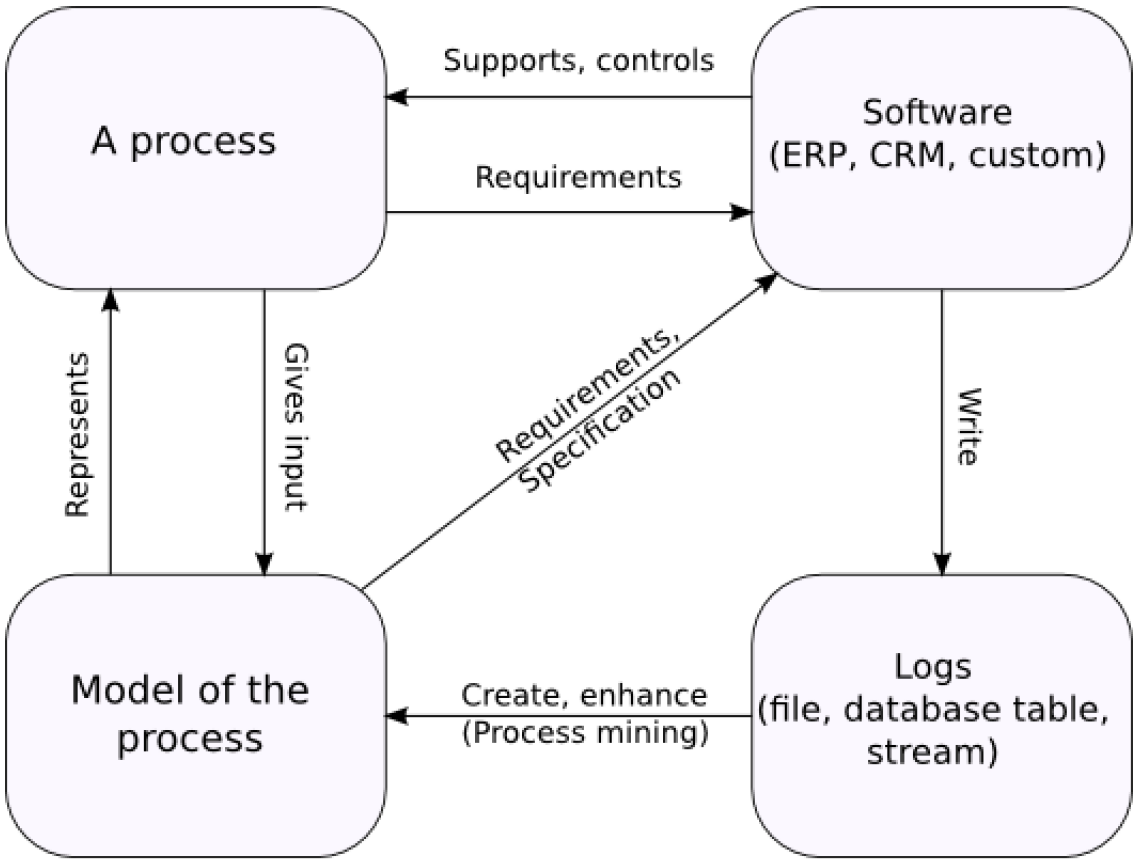


Figure 3: Entities involved in process mining

There are 4 main entities involved in process mining (see Figure 3): the process under investigation; the software that supports and/or controls the process; the event logs, where the information about execution of activities are stored; and a process model – sometimes it exists

before mining and sometimes the model is generated from scratch using discovery algorithms.

The process is a real world phenomenon in which people do their daily tasks using any tools to get their tasks done.

The software is usually an information system used by an organization. Such system can be for (but not limited to) keeping track of inventory, doing daily bookkeeping or managing sales. Usually the software is implemented in a centralized way – there is a server system, which handles requests from client systems. The server system usually records all incoming requests and outgoing responses to a log file. Some systems even write the log that is ready for process mining without pre-processing.

The event logs, as stated before, are the main prerequisites for process mining and the quality of the result depends directly from the quality of the logs.

Level	Characterization
*****	Highest level: the event log is of excellent quality (i.e., trustworthy and complete) and events are well-defined. Events are recorded in an automatic, systematic, reliable, and safe manner. Privacy and security considerations are addressed adequately. Moreover, the events recorded (and all of their attributes) have clear semantics. This implies the existence of one or more ontologies. Events and their attributes point to this ontology. Example: semantically annotated logs of BPM systems.
****	Events are recorded automatically and in a systematic and reliable manner, i.e., logs are trustworthy and complete. Unlike the systems operating at level ***, notions such as process instance (case) and activity are supported in an explicit manner. Example: the events logs of traditional BPM/workflow systems.
***	Events are recorded automatically, but no systematic approach is followed to record events. However, unlike logs at level , there is some level of guarantee that the events recorded match reality (i.e., the event log is trustworthy but not necessarily complete). Consider, for example, the events recorded by an ERP system. Although events need to be extracted from a variety of tables, the information can be assumed to be correct (e.g., it is safe to assume that a payment recorded by the ERP actually exists and vice versa). Examples: tables in ERP systems, events logs of CRM systems, transaction logs of messaging systems, event logs of high-tech systems, etc.
**	Events are recorded automatically, i.e., as a by-product of some information system. Coverage varies, i.e., no systematic approach is followed to decide which events are recorded. Moreover, it is possible to bypass the information system. Hence, events may be missing or not recorded properly. Examples: event logs of document and product management systems, error logs of embedded systems, worksheets of service engineers, etc.
*	Lowest level: event logs are of poor quality. Recorded events may not correspond to reality and events may be missing. Event logs for which events are recorded by hand typically have such characteristics. Examples: trails left in paper documents routed through the organization (“yellow notes”), paper-based medical records, etc.

Table 1: Maturity levels for event logs [6]

In process mining, the most common log formats are MXML [17] and XES [18]. There are very few software solutions that produce their logs in those formats. Usually a log is a flat text file, where the events not always appear in the order they happened. Indeed, log recording is usually implemented in an asynchronous manner – software sends all log requests to an interface and the interface decides, when to write the log entries to a file. However there are tools [19] [20] for converting different log formats (flat file, database table) to XES or MXML formats.

MXML was the first format introduced for process mining. MXML was developed mainly to hold simple information about the process – set of traces, where each trace has a set of events.

Each event is described with a fixed set of attributes. Each organization is a bit different and requires different set of attributes to be recorded with each event. To overcome this issue, a new format was proposed -XES [20]. The application developed for this thesis expects that the log is already converted in XES format.

The process models describe the process using abstractions. A process can be represented graphically using boxes and arrows; in written forms: story, temporal logic expressions or regular languages; or verbally, in which case the process model is in somebody's head.

Process Mining and Business Process Management

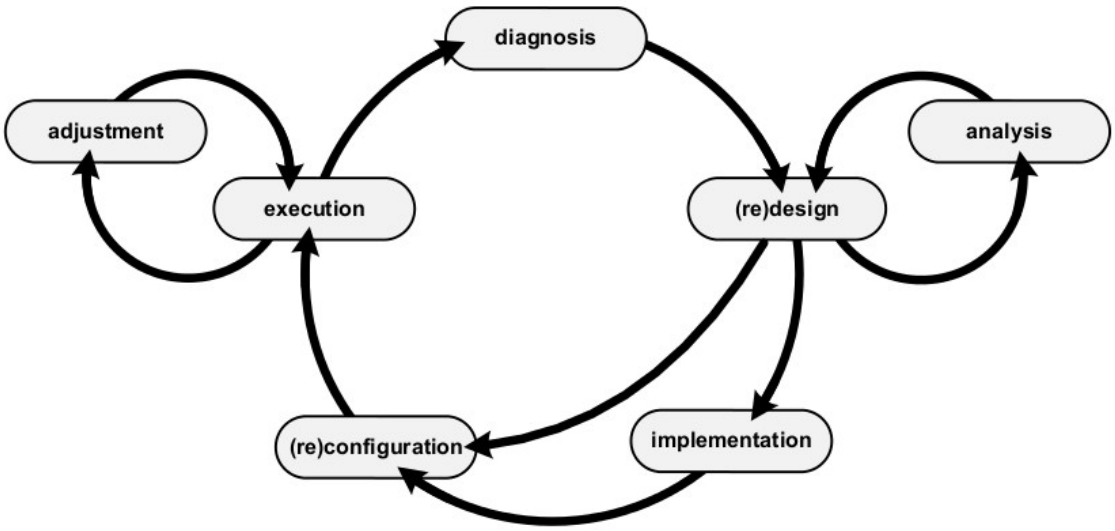


Figure 4: BPM lifecycle [6]

In organizations a process has a lifecycle (see Figure 4). The lifecycle starts with a design of a process –someone has to come up with an idea about what has to be done to achieve a goal. Then, these ideas are verified and validated – analysis phase. When the steps are defined, a plan is introduced to participants and necessary tools are acquired – implementation phase. Configuration is mostly necessary to allocate resources to activities. When all process parameters are confirmed, the process is executed. In the execution phase the process is monitored and small adjustments can be done, which do not require redesign. When the process has run for some time, it goes to the diagnosis phase, where the process is analyzed for efficiency. The output of this phase may trigger the process to be redesigned.

In the process mining manifesto [6], the authors point out that process mining can be used in every phase of the business process management lifecycle except from implementation. However, mainly, it is used for diagnosing the process, but in other phases process mining can give operational support for users.

Declare

Declare is a declarative language based on LTL (see next chapter) to formally specify the semantics of a constraint (a declaration). A constraint is a property that is meaningful in the context of process modeling. Since LTL formulas can be difficult to understand, Declare associates a graphical representation to each constraint. By using this approach, the users do not need to have knowledge of temporal logic. Instead, they can learn the intuitive meanings of names and graphical representations of constraints. [21] There are four classes of constraints in the Declare language: existence constraints, relation constraints, negation constraints and choice constraints.

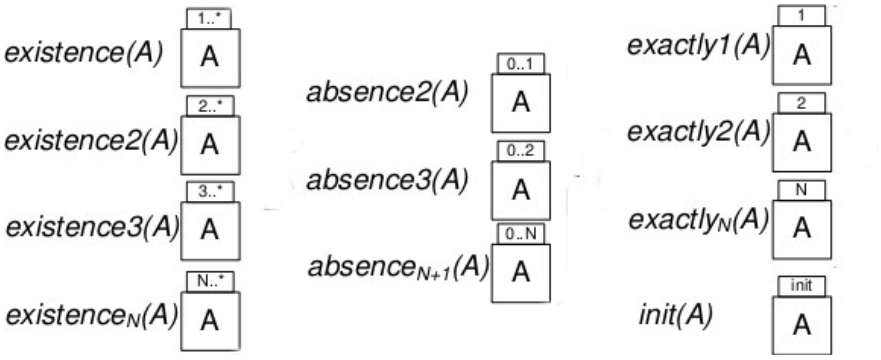


Figure 5: Existence templates [21]

Existence constraints specify how many times an activity may occur or may not occur in a trace. There are four types of existence constraints (see Figure 5): $existence_N(A)$ specifies the minimum number of activities in a trace; $absence_N(A)$ specifies the maximum number of activities in a trace; $exactly_N(A)$ specifies the exact number of activities in a trace; init(A) specifies the activity to be the first activity in a trace.

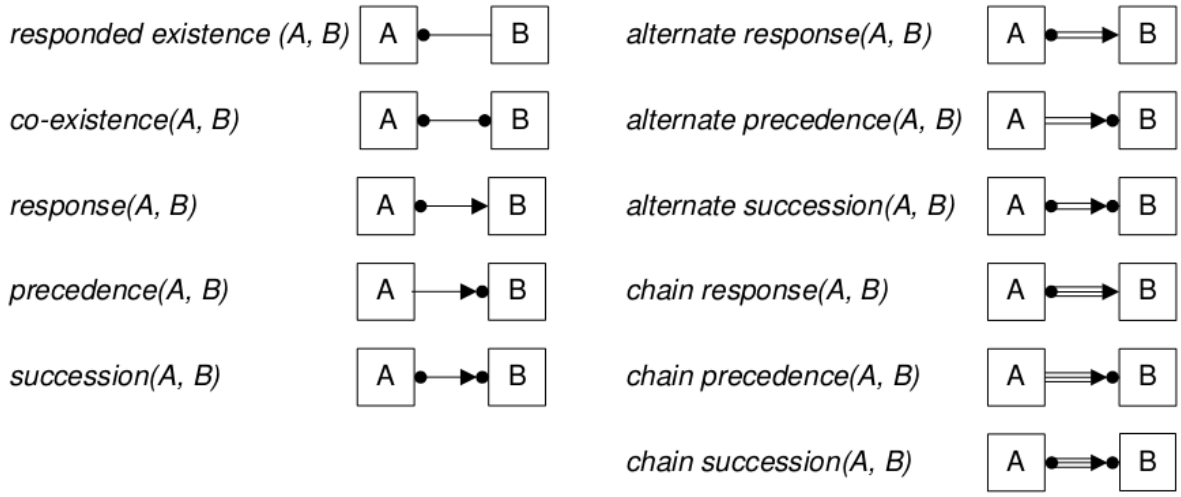


Figure 6: Relation templates[20]

Relation constraints (see Figure 6) define the dependency between two activities in a trace. The responded existence constraint specifies that if activity A is present, activity B also has to be present in the trace – the order of A and B does not matter, B can appear before or after A. The co-existence constraint specifies that the two activities must both be present in the trace. Again the order is not specified –co-existence(A, B) is equivalent to co-existence(B, A).

Response, precedence and succession relation constraints specify also the order of two activities. The response constraint specifies that if A is in the trace, activity B must be eventually present in the trace, but after A. The precedence constraint specifies that if B is present in the trace, then activity A must be present in the trace somewhere before B. Response and precedence seem to specify the same constraint, but the difference is, that response(A, B) is also true, when there is only A in the trace. In contrast, precedence(A, B) is true, when only B is present in the trace. The succession constraint requires both A and B to be present in the trace and A must be eventually followed by B.

Alternate and chain constraints make the response, precedence and succession stronger. Alternate response requires, that after each A there must be a B (alternate response(A, B) is true in trace <A, C, B, B, F, A, B>, but false in <A, A, B>). The same applies to precedence and succession. Alternate precedence requires, that there is always A before each B. Alternate succession specifies that for every A in the trace, there must be exactly a B eventually following it.

Chain response specifies, that every A in thetrace has to be immediately followed by B. Chain

precedence specifies, that there must be an activity A immediately before B. Chain succession requires both A and B to be in the trace and A is immediately followed by B.

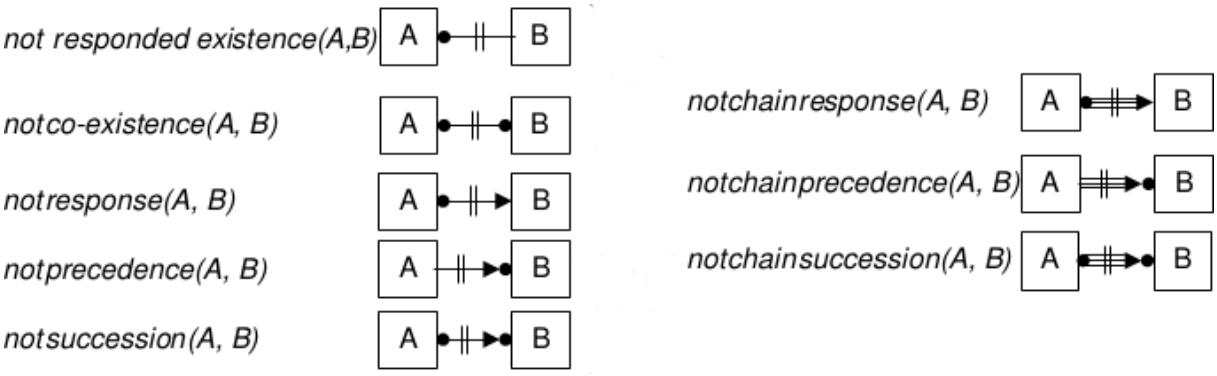


Figure 7: Negation templates [20]

Negation constraints (see Figure 7) are negated versions of relation constraints. But it is important to note that the negation is not be interpreted as logical negation, For example, both responded existence and not responded existence can be true in the same trace.

Not responded existence constraint specifies that if activity A is present, activity B must not be present in the trace not before not after. Not co-existence specifies that not responded existence(A, B) and not responded existence(B, A) must be true in the trace.

The not response constraint specifies that if A is in the trace, activity B must not be present in the trace after A. The not precedence constraint specifies that if B is present in the trace, then activity A must not be present in the trace before B. The not succession constraint requires that both not response and not precedence to be true in the trace.

Not chain response specifies, that every A in the trace must not to be immediately followed by B. Not chain precedence specifies, that every B must not be immediately preceded by A. Not Chain succession requires both not chain response and not chain precedence to be true in the trace.

Choice constraints (see Figure 8) specifies a) a number of activities that can follow a specific activity (inclusive choice) or b) only a certain activity can follow a specific activity (exclusive choice).

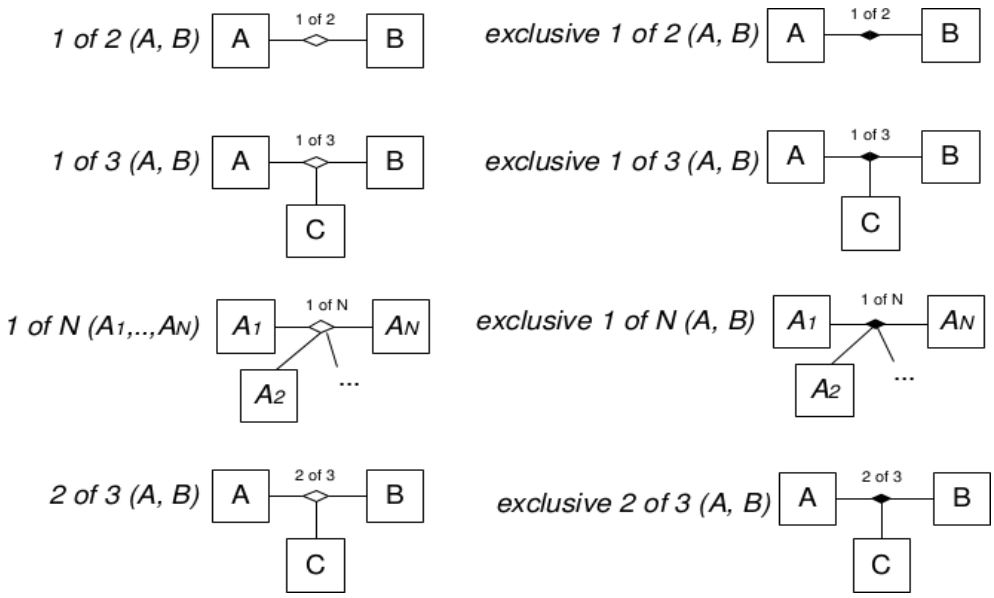


Figure 8: Choice templates [20]

The 1 of 2 constraint specifies that at least one of the two activities A and B has to be present in the trace, but both can be present and each of then can be present an arbitrary number of times without any order restriction. Similarly, 1 of N specifies that all the specified activities can be present in the trace, but at least one has to be present. 2 of 3 adds the restriction that at least two activities must be present in the trace.

Exclusive 1 of N choice specifies that only one of the given activities can be present in the trace. Exclusive 2 of 3 specifies that exactly 2 of 3 activities must be present in the trace.

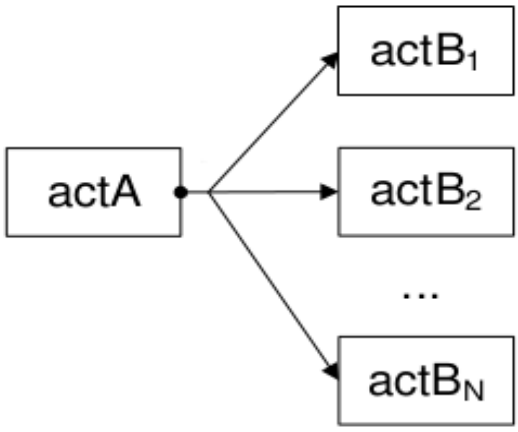


Figure 9: Branched response template [20]

Each Declare templates involves a specific number of activities: existence(A) has one,

response(A, B) has two and 1 of 3(A, B, C) has three. Some times there is a need to assign multiple activities as a parameter. Template parameter is called branching, when it is assigned more than one activity.

Temporal Logic

Besides describing a process using graphical notations (boxes and arrows), the model can be constructed using different types of expressions or (machine readable) text formats. For example imperative models can be converted to Business Process Execution Language (BPEL) [22]. BPEL is used to exchange information about processes between computers and configure Business Process Management systems. Declare models can be also converted to similar formats, for example, they can be also represented by a set of temporal logic expressions.

Temporal Logic is a type of modal logic with modalities referring to time – truth value of the expressions varies over the time. Temporal logic adds two binary operators and five unary operators to propositional logic. The binary operators are until and release, the unary operators are next, future, globally, all and exists. In a temporal logic formula we can then express statements like "I am always hungry", "I will eventually be hungry", or "I will be hungry until I eat something".

Linear Temporal Logic (LTL) [23] was first proposed by Amir Pnueli in 1977. In 1981, E. M. Clarke and E. A. Emerson defined Computational Tree Logic (CTL) [24]. LTL considers time to be linear - "I will eventually be hungry", whereas CTL allows branching of time - "It is possible that I will eventually be hungry" or "This is always the case, I will eventually be hungry". Each branch in time can be viewed as a possible path of events and the future is not determined. These logics where developed independently until 1986, when E. A. Emerson and Joseph Y. Halpern defined CTL* [25] to combine both LTL and CTL.

The syntax of LTL is built up from constants **true** and **false,** a finite set of atomic propositions, the logical operators ¬, ∨, ∧, →, ↔, and the temporal modal operators **X** (next), **F** (finally or eventually), **G** (globally or always) and **U** (until). Formally, the set of LTL formulas over set of atomic propositions *AP* is inductively 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.

Consider the following traces in an event log:

An event log
OrderGoods → ReceiveInvoice → PayInvoice → ReceiveGoods → RecordTransaction;
OrderGoods → ReceiveInvoice → RejectInvoice → RecordTransaction;
OrderGoods → ReceiveInvoice → PayInvoice → RejectGoods → RecordTransaction;

Table 2: Example of an fictional event log

It is said that an LTL (check everywhere AN LTL) formula holds in a log, when it is true in all initial states.

The following LTL formulas are true on all the above mentioned traces (*AP* is the set {OrderGoods, ReceiveInvoice, PayInvoice, ReceiveGoods, RejectGoods, RecordTransaction}):

X(ReceiveInvoice) - “The invoice is received next after initial event”

F(RecordTransaction) - “The transaction is eventually recorded”

G(OrderGoods → F(RecordTransaction)) - “After ordering goods a transaction is recorded”.

The syntax of CTL is similar to LTL, but CTL introduces path quantifiers **E** (exists a path) and **A** (in all paths). Formally, the set of CTL formulas over a set of atomic propositions *AP* is inductively defined as follows:

- **true** and **false** are CTL formulas;
- if p ∈ *AP* then p is a CTL formula;
- if ψ and φ are CTL formulas then ¬ψ, φ ∨ ψ, φ ∧ ψ, φ → ψ, φ ↔ ψ, **EX** ψ, **EF** ψ, **EG** ψ, **E**[φ **U** ψ], **AX** ψ, **AF** ψ, **AG** ψ, **A**[φ **U** ψ] are CTL formulas.

In most cases, appending an LTL formula with the quantifier **A** will turn the LTL expression to the CTL equivalent.

AX(ReceiveInvoice) - “Always an invoice is received next after initial event”

AF(RecordTransaction) - “It is always the case that a transaction is eventually recorded”

AG(OrderGoods → F(RecordTransaction)) - “It is always the case, that after ordering goods a transaction is recorded”. This is smilar to previous statement, but stronger.

But CTL does not supersede LTL, for example **FG**p is a formula in LTL, but there is no equivalent in CTL [24].

Temporal logic is used in model verification in hardware design phase where a hardware designer needs to test the developed model against the specification using model checking with formal methods [26]. The main driver in model checking is that it is much less expensive to get the design right before starting the implementation phase. There are several examples of bugs [27] that cost a lot to the vendor. To minimize such risks, many companies have implemented formal verification methods, to check the correctness of a design or an algorithm before the implementation phase.

Declarative process models have several things in common with formal verification methods: there is a model, properties and execution paths of the model. Formal verification methods start with a modeled system and properties describing the system. The task is to automatically prove that a property holds in the system. Declarative process models are described with a set of properties, which are called constraints (in many cases property and constraint are synonyms) and the system is allowed only to execute such paths, which satisfy the constraints.

Each execution of a process path produces one trace in a log. Given a set of logged execution paths, the task of finding the constraints of the (declarative) model, which could have produced the traces is called (declarative) process discovery.

The basis of the approach used in this thesis to discover the process model is to build a transition system from an event log and implement an algorithm to check temporal logic expressions against the transition system.

Model Checking and Query Checking

In context of model checking, a transition system represents the real system as a set of states, the transition relation between the states and a set of variables that hold the information about the states. Each state can have zero or more incoming transitions from other states and can have zero or more transitions going out from the state. This is similar to what happens in activity diagrams that represent a process, where the process is the real system, the transition

system is the process model and an activity is a state.

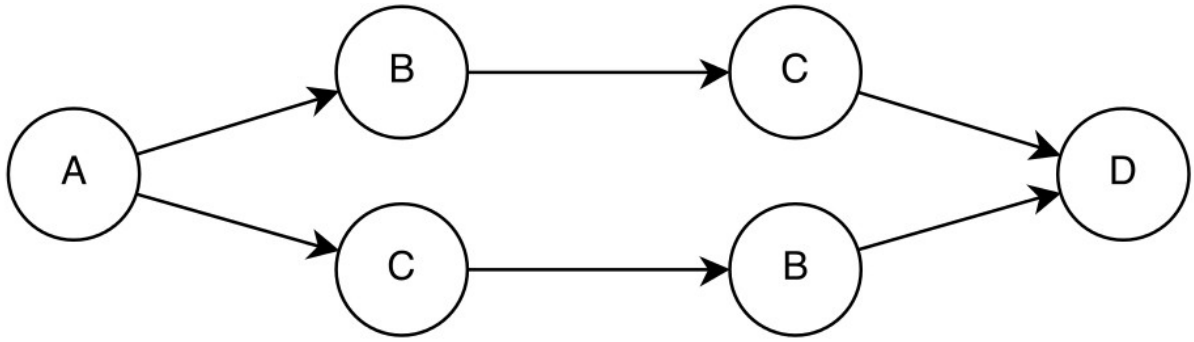


Figure 10: A simple transition system with 6 states, 6 transition relations and 4 variables

Model checking requires a more specific type of transition system. There is one criterion: every state must have at least one outgoing transition. This enables to generate infinite execution paths – this is needed because temporal logic can represent only infinite traces. To get such model from process model, which has finite paths, it is enough to add an artificial transition from the final state to itself – a self loop. [26].

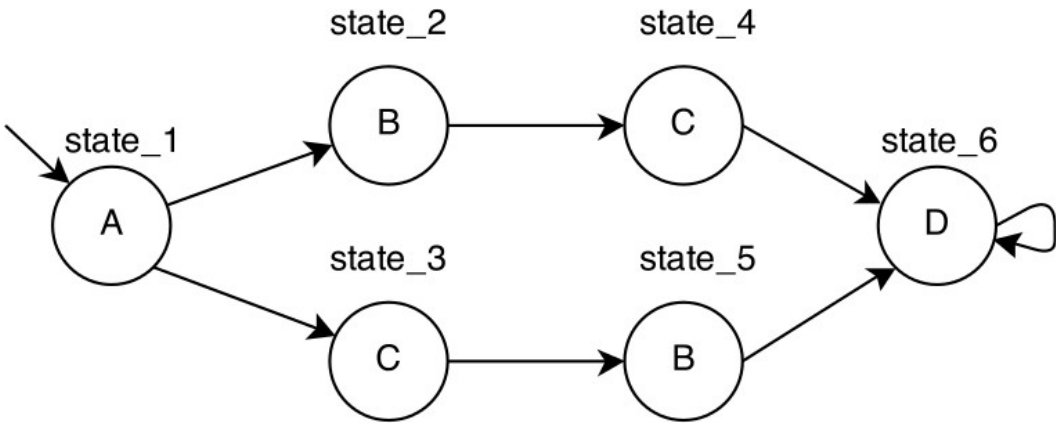


Figure 11: Simple Kripke structure

The transition system in model checking context is always represented by a Kripke structure named after Saul Kripke, who proposed the idea. A Kripke structure is a variation of nondeterministic automaton [28] and is defined as a tuple $M=(A, S, I, R, L)$. Where:

- A is the set of atomic propositions used in the system, e.g., {A, B, C, D} in Figure 13
- S is the set of all states, e.g., {state_1, state_2, state_3, state_4, state_5, state_6}

- I is the set of initial states s.t. $I \subseteq S$, e.g., {state_1}
- R is the set of all transition relations in the system s.t. $R \subseteq S \times S$, e.g., ({(state_1, state_2), (state_1, state_3), (state_2, state_4), (state_3, state_5), (state_4, state_6), (state_5, state_6), (state_6, state_6)}.
- L the labeling function that maps each state onto the propositional variables which hold in it, e.g., {A → {state_1}, B → {state_2, state_5}, C → {state_3, state_4}, D → {state_6})}.

For convenience the transition relations are left unlabeled and only those relations that have value “true” are shown.

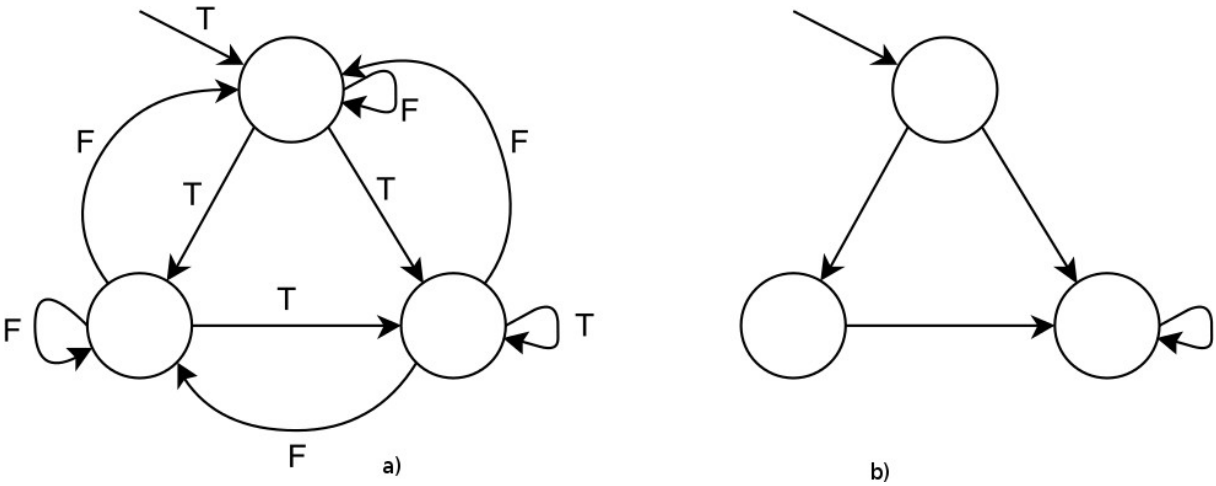


Figure 12: a) Transition system with labelled arcs b) arcs labelled as False removed

For example, in case of hardware design the system can have more than one proposition set to true at the same time. To continue with the analogy with activity diagrams, when the process is in a particular state, then only one activity can be enabled. In this thesis an activity is presented as an atomic proposition.

The input for model checking is a Kripke structure and a specification, expressing which properties must hold in the system. For the model checking process to be automatic, the properties must be formally described. The properties can be also viewed as questions about the system (see Table 2)

1. „Is OrderGoods always followed by RecordTransaction?“,
2. „Is the next activity after ReceiveInvoice PayInvoice or RejectInvoice?“,

3. „Do all processes start with OrderGoods immediatelly followed by ReceiveInvoice?“.

There are several ways to define the properties, but often a temporal logic expression is used.

For example, the questions above can be expressed using LTL as:

Q1. G(OrderGoods -> F(RecordTransaction))

Q2. G(ReceiveInvoice -> X(PayInvoice V RejectInvoice)

Q3. OrderGoods -> X(ReceiveInvoice)

A Temporal Logic Query is a temporal logic expression with placeholders and the task is to find the solutions to the placeholders. A solution to the query is any propositional formula such that, when substituting the placeholder with it, it yields to a temporal logic expression that is satisfied in the model. Usually there are more than one solution to a query.

Temporal logic query checking (or query checking for short) was proposed by William Chan [29] to speed up design understanding by discovering properties not known beforehand. In his work he used only one placeholder and presented it with a question mark (?). In this work, he used CTL. For example, querying all activities immediately following ReceiveInvoice can be written as AG(ReceiveInvoice -> AX(?)). The solutions to the query is {PayInvoice, RejectInvoice, PayInvoice ∨ RejectInvoice} (see Table 2). Since PayInvoice and RejectInvoice can be derived from PayInvoice ∨ RejectInvoice, the first two solutions can be derived from the last one. The last solution is also called the strongest solution for the query.

The thesis is about defining and implementing the task of running temporal logic queries against an event log and getting back a set of LTL constraints that describe the process at hand.

Solution

The solution for query checking proposed by Chan [29] is to directly substitute all possible propositional formulas to the placeholders and check the resulting expression against the model. The main problem with this approach is the scalability, when the model contains k atomic propositions, then there are 2^{2^k} substitutions needed.

To mitigate the issue, Bruns and Godefroid [30] provide a mechanism for computing all solutions to arbitrary queries with a single placeholder, occurring either in a negative ($\neg ?_x$) or a positive position ($?_x$) in the query, using extended alternating automata (EAA) [31]. Hornus and Schnoebelen [32] generalize the problem further to make no such restriction to the temporal logic queries. They introduce an algorithm to efficiently produce some of the maximally strong solutions for positive queries with a single placeholder. The negative queries can be turned to positive queries and the solutions again turn back. Their algorithm computes one solution to the query using a linear number of calls to the model checker, two solutions using a quadratic number of calls to the model checker, etc.

In xChek [13] query checking is implemented on top of multi-valued (see Figure 15) model checking. The model checking discussed so far uses Boolean logic as algebra – the atomic proposition in the model and in the temporal logic expressions take their value from the set {true, false}. Boolean logic can also be referred to as 2-valued logic. In multi-valued logic, the algebra is any De Morgan algebra, which can be represented by a finite distributive lattice. The values for the atomic propositions are taken from the set of all lattice elements and the logic operators “and” and “or” are defined through lattice operations meet and join respectively. Negation is a function, which preserves involution ($f(f(x))=x$, in classical logic, negation operator preserves involution $\neg\neg x=x$).

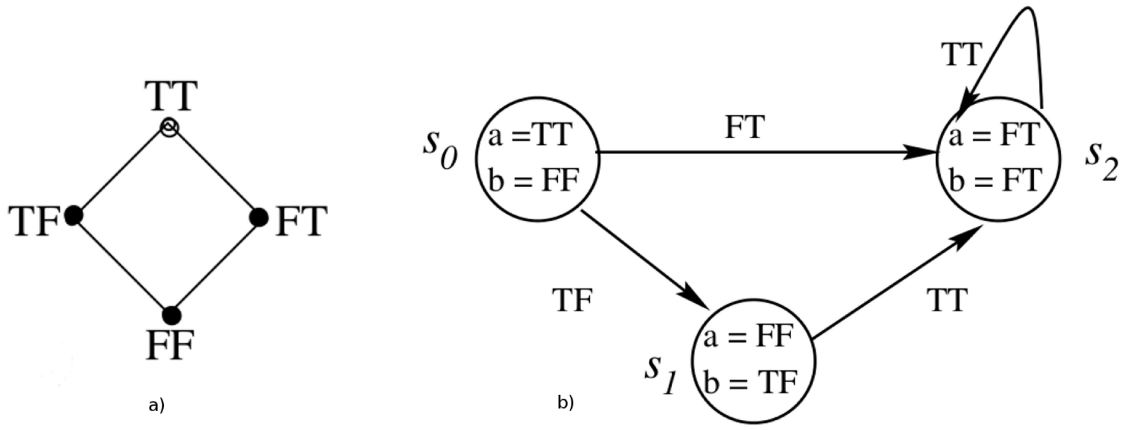


Figure 13: a) 4 valued logic as a lattice; b) transition system with 4 valued logic

For query checking, the propositions in the model and in the expression are two valued, but they allow the lattice elements to appear as constants in CTL expressions. The lattice used is an upset lattice, which is constructed from the set of all propositional formulas of atomic propositions used by the system. For query checking, the placeholders have to be substituted only with values in the set of all join-irreducible elements of the upset lattice. The intermediate solution is all join-irreducible elements, which hold in the model. The final solution is constructed by combining the join-irreducible elements. More information can be found in [13] [30].

All the components and algorithms were already available to implement this approach. There is an application to convert event log to transition system [33] and another application to run temporal logic queries against that system [13]. The output can be used to manually sketch the model or format it in a way it could be used as an input to a modeling software. There are many drawbacks of this approach, the task is tedious, error prone and takes lot of time to complete on cycle.

The first idea to implement the query checking solution for process mining was to implement a simple facade on top of xChek. The input for the facade is an event log and a list of temporal logic queries and the output is a list of LTL constraints which hold on the traces of the log. xChek is a GUI based desktop application written in Java. It is a tool for model checking and exploration. It also supports fairness[1] as input and can generate counter

1 Fairness is a concept of filtering out a subset of states that are not considered during model checking

examples. In theory it should also support witness[2] generation, but this seems not to be implemented.

xChek is developed for "true" model checking, where all possible solutions to a query are produced. In the context of business process management, the expectation for a solution is a bit less strict and mostly it is sufficient to get a list of single events as solution for the query.

As output xChek produces all possible propositional formulas that can be substituted with the placeholders in the input query. For example, when the minimal solution to the query is (a ∨ b ∨ ¬c) then the solutions to the query are also {a, b, ¬c, a ∨ b, a ∨ ¬c, b ∨ ¬c}. In Declare, a solution with two or more atomic propositions[3] joined by a disjunction is a branched constraint. When the amount of atomic propositions and the log size grows, the xChek output will contain several solutions to the query, which all are valid, but may not be interesting to the user.

To resolve this, the facade also takes an additional input, a configuration parameter, which indicates, how many branches the user expects at most. 0 for single events only, 1 for choice between two events, etc. Before converting xChek result to LTL formulas, the facade post processes the result to filter out solutions, which has a greater number of branches than the one specified by the user.

xChek takes the model as an input in SMV [34], GClang [35] or XML [36] format. xChek does not support the construction of a transition system from an event log. Therefore, the first task for the facade is to read in the log file and convert it to a model description of supported format and initialize xChek. Next a list of temporal queries are taken from a file and for each query a call to xChek is made. The output is post-processed and returned to the user.

The solution worked fine, until it was tested against real life logs -even for a simple query, xChek took hours to compute the solution on quad core CPU at 2GHz with 8GB of memory. Before discarding completely the idea to use xChek, an attempt to modify the xChek source code was made. Because of the complexity of the code and failure to get contact with the developer, it was decided to start implementing a symbolic model checking package from scratch.

Indeed, the original implementation of xChek makes it a very general framework, in the sense

2 A path in the model, where a CTL expression holds.

3 An event in a process is represented as an atomic proposition – for each event class in the log, there is one atomic proposition in constructed Kripke structure.

that it uses multi-terminal decision diagrams (MDD) -the terminal nodes of the diagram, can take their values from an arbitrary set. This is needed to construct functions which are used to reason about uncertainty in a model or even more complex situations in hardware (or software) design verification phase. The uncertainty is defined as a third value added to the boolean values: true, false, unknown. For example, a value of an atomic proposition may be unknown in some states or the state does not depend on the value at all. The second generalization comes from the fact that the decision diagrams has to be multi-terminated. To support that they need other algebras, beside the classical one. Because of these generalizations the source code is very complex and maybe has also impact on the performance.

Implementation

Model checking is a graph traversal problem, where the graph is a Kripke structure. Recall that the Kripke structure consists of set of states, a set of relations and a labeling function, which maps an atomic proposition to the set of states, where the atomic proposition is true. In model checking, it is important to be able to calculate predecessors of a given set on states and perform set operations in the state space. Both requirements can be implemented using boolean functions. A boolean function is $f : B^k \rightarrow B$, where $B = \{0, 1\}$.

Model checking is called symbolic, when the model is represented using boolean functions. To represent a Kripke structure using boolean functions, there has to be a function for transition relations and one function for each atomic proposition, which represent the set of states in which the atomic proposition is true. We write the symbolic model as $\langle f_R, f_{L(x_0)}, \ldots, f_{L(x_n)} \rangle$, where R is the set of transition relation, L is the labeling function and x_i ($i \in \mathbb{N}$)is an atomic proposition.

For example, the transition system in Figure 16 has following properties:

$$R = \{(1,2), (1,5), (1,6), (1,8), (2,3), (3,4), (4,4), (5,3), (6,7), (7,4), (8,9), (9,10), (10,10)\}$$

$$L = \{A \rightarrow \{1\}, B \rightarrow \{2,6,9\}, C \rightarrow \{3,5\}, D \rightarrow \{4\}, E \rightarrow \{7,8\}\}$$

and is symbolically represented by $\langle f_R, f_{L(A)}, f_{L(B)}, f_{L(C)}, f_{L(D)}, f_{L(E)} \rangle$.

The boolean functions represent a set of states and set of pair of states, where each state is represented by a boolean function. To represent 10 states using boolean functions, we need at

least 4 binary variables ($\lceil \log_2(10) \rceil = 4$). State 1 is represented by a function $f(x_{1,}x_{2,}x_{3,}x_4) = \neg x_1 \wedge \neg x_2 \wedge \neg x_3 \wedge x_4$ or $f_1(\hat{x}) = \bar{x}_1 \bar{x}_2 \bar{x}_3 x_4$ for short. State 2 is a function $f_2(\hat{x}) = \bar{x}_1 \bar{x}_2 x_3 \bar{x}_4$, etc. Now to encode the set of states, it is just the matter of "joining" the encoded states, so $f_{L(A)} = f_1$, $f_{L(B)} = f_2 \vee f_6 \vee f_9$, $f_{L(C)} = f_3 \vee f_5$, $f_{L(D)} = f_4$ and $f_{L(E)} = f_7 \vee f_8$.

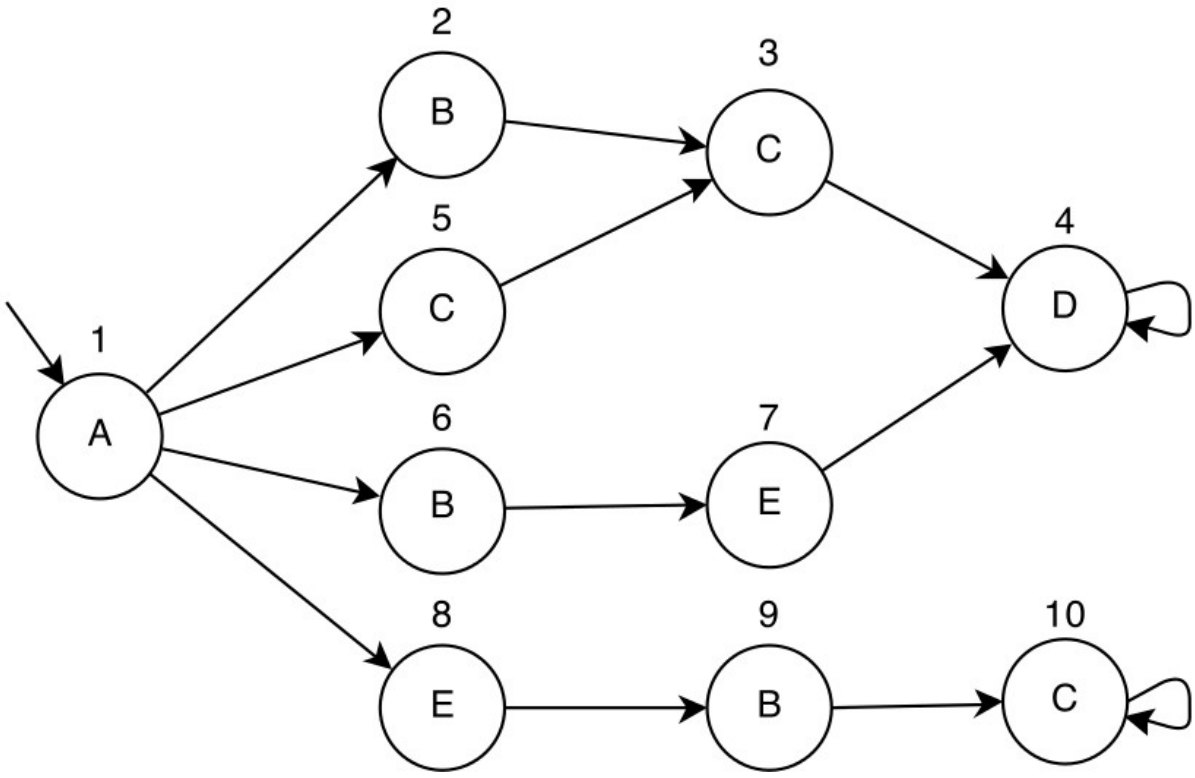
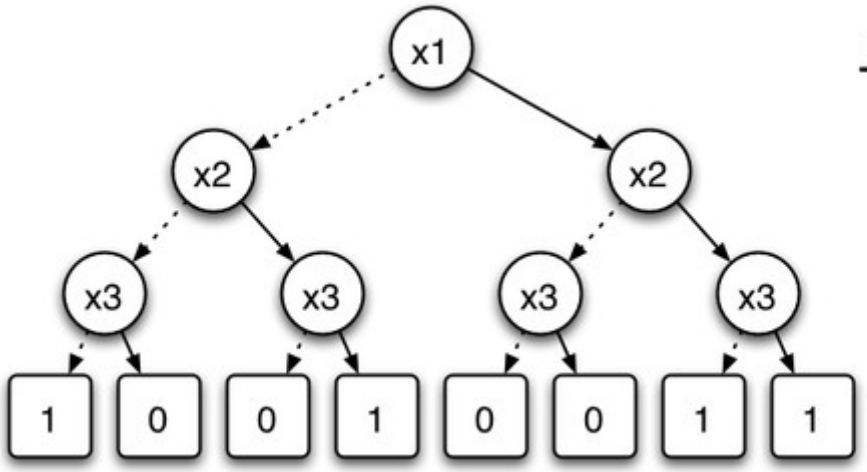


Figure 14: Kripke structure from traces <a, b, c, d>, <a, b, c, d> <a, c, c, d>, <a, b, e, d>, <a, e, b, c>

To encode transition relations, it is needed to encode the source state and the destination state differently. For the source state the existing state encodings are used, but for destination states additional 4 bits are needed - $\hat{x}'$. The destination state 2 is encoded as $f'_2(\hat{x}') = \bar{x}'_1 \bar{x}'_2 x'_3 \bar{x}'_4$ and the transition (1, 2) is $f_{1,2} = f_1 \wedge f'_2$. Other transitions are encoded similarly. The set of all transitions $B_{\rightarrow}$ is then $f_{\rightarrow} = f_{1,2} \vee f_{1,5} \vee f_{1,5} \vee f_{1,8} \vee f_{2,3} \vee f_{3,4} \vee f_{4,4} \vee f_{5,3} \vee f_{6,7} \vee f_{7,4} \vee f_{8,9} \vee f_{9,10} \vee f_{10,10}$.

A boolean function can be represented as a full and complete binary decision tree [37], where leaves are terminal nodes (labeled with 0 or 1) that represents the value of the function on a

given input – this is the same as the value in the truth table.



x1	x2	x3	f
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

Figure 15: Binary decision tree and truth table for $f(x_{1,}x_{2,}x_3)=\bar{x}_1\bar{x}_2\bar{x}_3+x_1x_2+x_2x_3$ *[38]*

The problem of using Binary Decision Tree is that the number of nodes is exponential with respect of the number of variables in the function. The number of terminal nodes is equal to 2 to the power of the number of the variables in a function. A function with 8 variables has 256 terminal nodes and 256 + 128 + 64 + 32 + 16 + 8 + 4 + 2 + 1 = 511 nodes in total. Fortunately the Binary Decision Tree can be reduced to a Binary Decision Diagram, which does not depend directly on the number of input variables. The first reduction rule is to merge the terminal nodes so that there are only two of them, one for 0 and one for 1. Next all subgraphs are removed, which do not affect the outcome.

Using reduction rules, a Binary Decision Tree can be reduced to Binary Decision Diagrams (BDD) [38] In general a Binary Decision Diagram is a directed acyclic graph that is used to represent a boolean function. Non-leaf nodes or non-terminal nodes are labeled with variable names. Each non-terminal node has two outgoing arcs labeled with 0 and 1. To evaluate a function on a given input, the graph is traversed starting from the root to a terminal node. In each non-terminal node the next transition is chosen according to the input value of a variable labeling the node. When the terminal node is reached, then the value of the node is returned. BDD is efficient data structure to evaluate boolean functions [38].

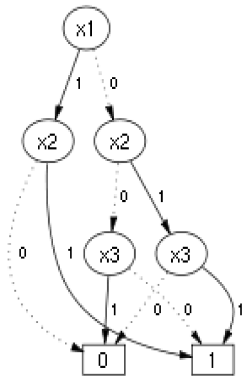


Figure 16: Binary Decision Diagram for $f(x_{1,}x_{2,}x_3)=\bar{x}_1\bar{x}_2\bar{x}_3+x_1x_2+x_2x_3$ *[37]*

The implementation is written in ANSI C. The reasons for selecting C over other languages was that the selected BDD library BuDDy [39] is implemented in C.

We start by converting a XES log file to a transition system. The main goal for producing the transition system is to keep the number of states as low as possible. This means to reuse the transitions which do not affect the outcome. In case of traces <a, b, c, d>, <a, b, c, d> <a, c, c, d>, <a, b, e, d>, <a, e, b, c> the resulting system will have 10 states (see Figure 16).

After the log is parsed to a transition system, it will be flattened to a set of transitions and a map from atomic proposition to set of states the proposition holds.

The next step is to parse the second input for CTL expressions. The expressions are given in a text file separated by semicolon (;) or new line. The CTL expression is converted to a tree structure where unary operations have single child, binary operators have two children and atomic proposition, constant value and placeholders are the leave nodes.

For example a Declare response(A, B) constraint is represented by a CTL expression $EG(A \rightarrow EF(B))$, see Figure x for the tree after parsing the expression.

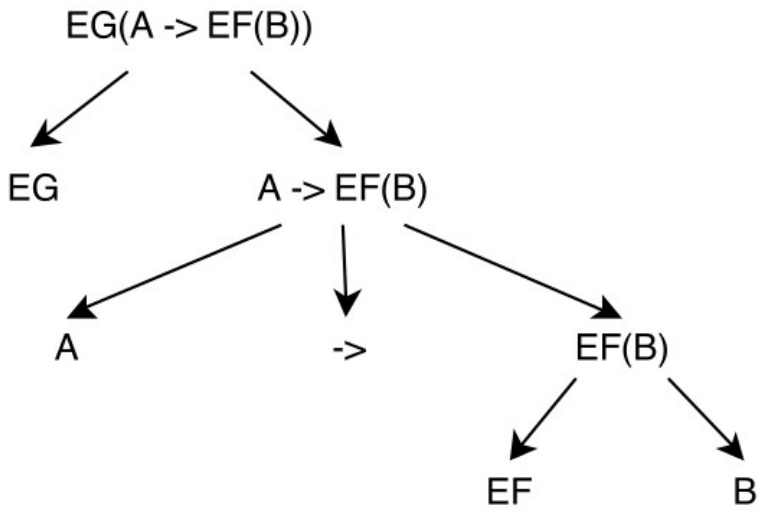


Figure 17: $EG(A \rightarrow EF(B))$ as a tree

Third step is the model checking. Currently the implementation is direct: every different placeholder is assigned value from the set A (all atomic propositions in the system or all events in the process). When there are 20 events and 3 placeholders, there will be about 20^3 calls to the model checking. The number is a bit lower, since the current implementation exploits the fact that it is not interesting to assign same atomic proposition to different placeholders. In case **E**[?x **U** ?y], we are not interested in to evaluating **E**[e **U** e], i.e., there exist a trace, where all activities are e before activity e starts, which is exactly the same as **EG**(e).

A CTL expression is evaluated against the transition system from inside out. Take, for example, the expression !**EG**(a → **EX**(b)). First the result for "b" is evaluated, then "**EX"**, "a", "→", "**EG**" and finally "!". The support is calculated so that all initial states form the resulting set are counted. The support is the count of initial states in which the formula holds (is true) divided by the total number of initial states.

The output is a list of constraints which hold in the model, with support. For example a single query **EX**(?x) on model in Figure 16, the output is:

0.00 **X**(A)
0.50 **X**(B)
0.25 **X**(C)

0.25 **X**(D)

0.00 **X**(E)

The result of an atomic proposition is the set of states, where the atomic proposition holds. In Figure 16 all states where B s true are: 2, 6, 9. This is the same as $L(B)$.

The result of negation operator is the complement of set, where formula φ holds.

The results of conjunction and disjunction of formulas φ and ψ are the intersection and union of sets where φ holds and ψ holds respectively.

Implication (φ → ψ) and equivalence (φ ↔ ψ) are represented by (! φ ∨ ψ) and

((! φ ∨ ψ) ∧ (φ ∨ !ψ)) respectively.

To evaluate temporal operators a helper function preExists(φ) is needed to calculate and return all predecessors for a set of states.

EX(φ) returns the set of states where φ holds plus all predecessors of the states -

EX(φ) = φ ∨ preExists(φ)

The evaluation of **EG**(φ) starts by first finding the set of states where φ is true and appends all predecessors of these states. Next again all predecessors of the predecessors are appended. The algorithm returns, when there are no predecessors to append.

The evaluation of **EF**(φ) is done using equivalent expression **E[**true **U** φ**]**.

The evaluation of **E[**φ **U** ψ**]** starts with the set of states where ψ is true and appends all predecessors of the set, where φ is true. Next again all predecessors the predecessors where φ is true are appended. The algorithm returns, when there are no predecessors to append.

Evaluation

To evaluate the implementation, its performances are measured with respect to log size, number of activities, number of events in a trace, number of placeholders in a CTL query and complexity of the CTL query. The results are compared against dedicated Declare constraint mining applications: MINERful [11] and Declare Maps Miner [10]. Also we provide some results of the first solution described in this thesis using xChek. The tests were executed in Windows 7 operating system on Intel Core i7 CPU and 8GB of memory.

There are three sets of logs: a set with variable number of traces (with fixed number of activities and fixed number of events in a trace.), a set with variable number of activities (with a fixed number of traces and fixed number of activities in a trace) and a set with variable number of events in a trace (with a fixed number of traces and fixed number of activities).

Test set	Traces	Activities	Activities in a trace
Variable number of traces	400 - 4000	10	10
Variable number of activities	100	5 - 50	10
Variable number of activities in a trace	100	10	5 - 50

Table 3: Properties of test sets

There are 10 different configurations of log files in each set and each configuration is generated 3 times, so there are 30 log files in each set. All test logs where started after one after another. The test runner started the application with a log file. The application checked a predefined set of queries against the log (see Table 4). The time was measured from the beginning of query checking – not including the time for log parsing and constructiong of Kripke structure. The test runner waited until the application was finished and shutdown; and started next run with next log file. The set of queries is based on the list of Declare constraints supported by the MINERful algorithm [11].

Declare constraint	CTL query
Init(?x)	?x
Existence(?x)	EF ?x
Absence2(?x)	!(EF(?x & EX(EF?x)))
CoExistence(?x, ?y)	(EF?x) <-> (EF?y)
RespondedExistence(?x, ?y)	EF(?x) -> (EF?y)
Response(?x, ?y)	EG(?x -> EF?y)
Precedence(?x, ?y)	(E[!(?y) U ?x]) | !(EG?y)
Succession(?x, ?y)	(EG(?x -> EF?y)) & ((E[!(?y) U ?x]) | !(EG?y))
AlternateResponse(?x, ?y)	EG(?x -> EX(E[!(?x) U ?y]))
AlternatePrecedence(?x, ?y)	((E[!(?y) U ?x]) | !(EG?y)) & (EG(?y -> EX((E[!(?y) U ?x]) | !(EG?y))))
AlternateSuccession(?x, ?y)	(EG(?x -> EX(E[!(?x) U ?y]))) & (((E[!(?y) U ?x]) | !(EG?y)) & (EG(?y -> EX((E[!(?y) U ?x]) | !(EG?y)))))
ChainResponse(?x, ?y)	EG(?x -> EX(?y))
ChainPrecedence(?x, ?y)	EG(EX(?y) -> ?x)
ChainSuccession(?x, ?y)	EG(?x <-> EX(?y))
NotCoExistence(?x, ?y)	!(EF?x & EF?y)
NotSuccession(?x, ?y)	EG(?x -> !(EF?y))
NotChainSuccession(?x, ?y)	EG(?x -> EX(!(?y)))

Table 4: Queries used for performance testing

The first test (see Chart 1 and 2) used a set of synthetic event logs, where each log file has the number of traces and number of events in a trace fixed, and the size of the log alphabet (number of activities) changes.

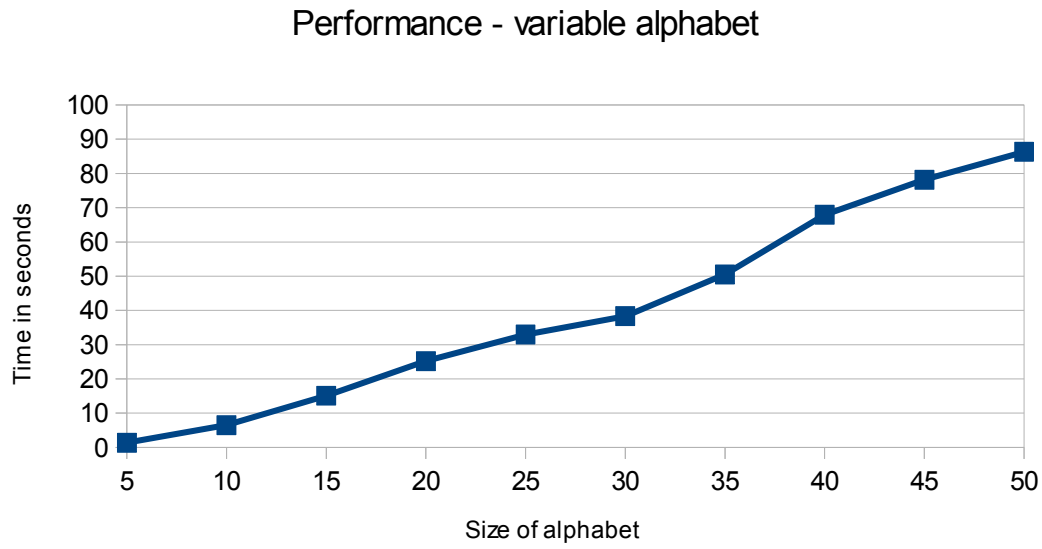


Chart 1: Performance test using variable alphabet size

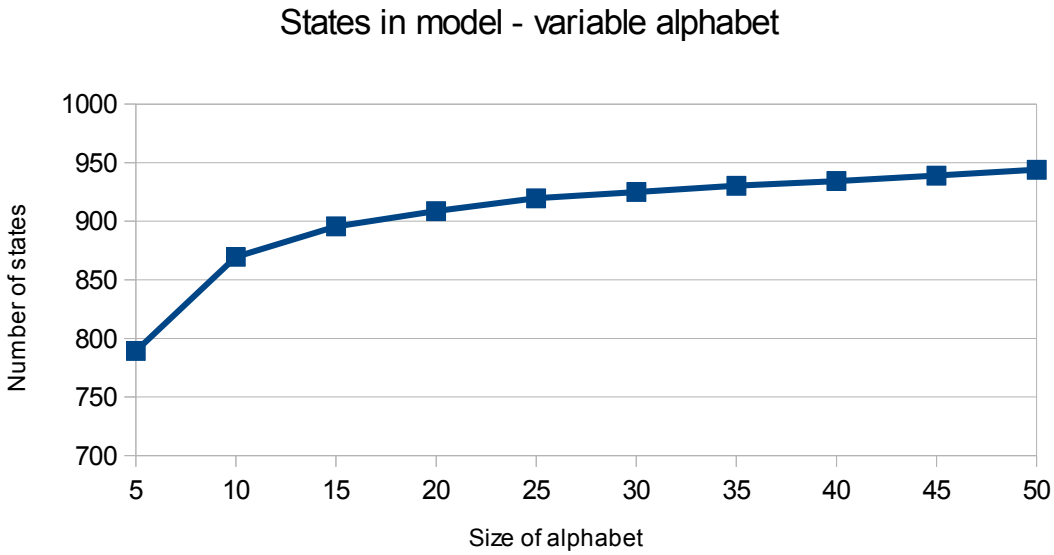


Chart 2: The trend of number of states in transition model respect to the growth in the alphabet

The second test (see Charts 3 and 4) used a set of synthetic event logs, where each log file has the number of events in a trace and the size of the alphabet fixed, and the number of traces changes.

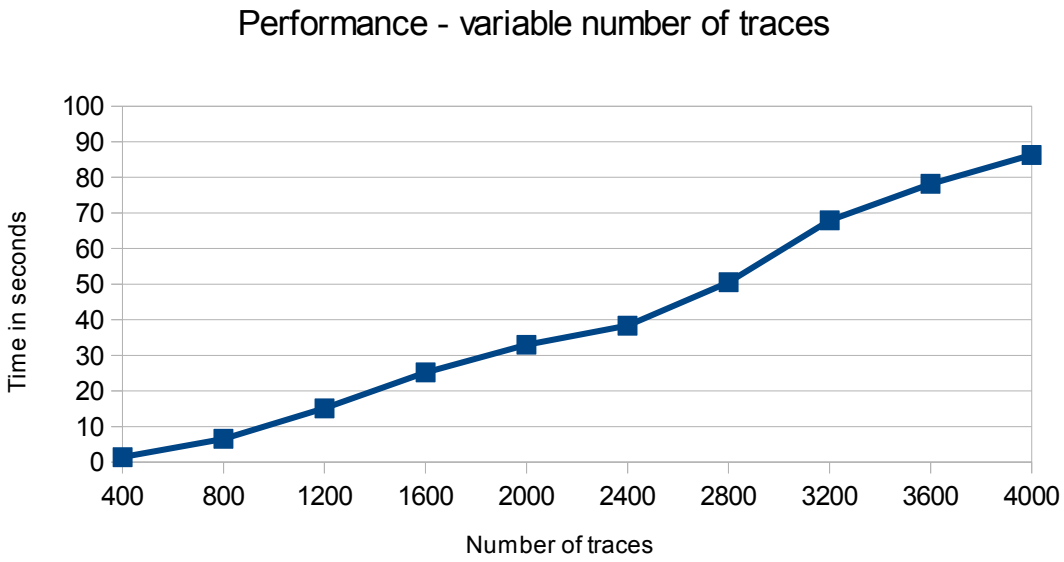


Chart 3: Performance test using variable number of traces

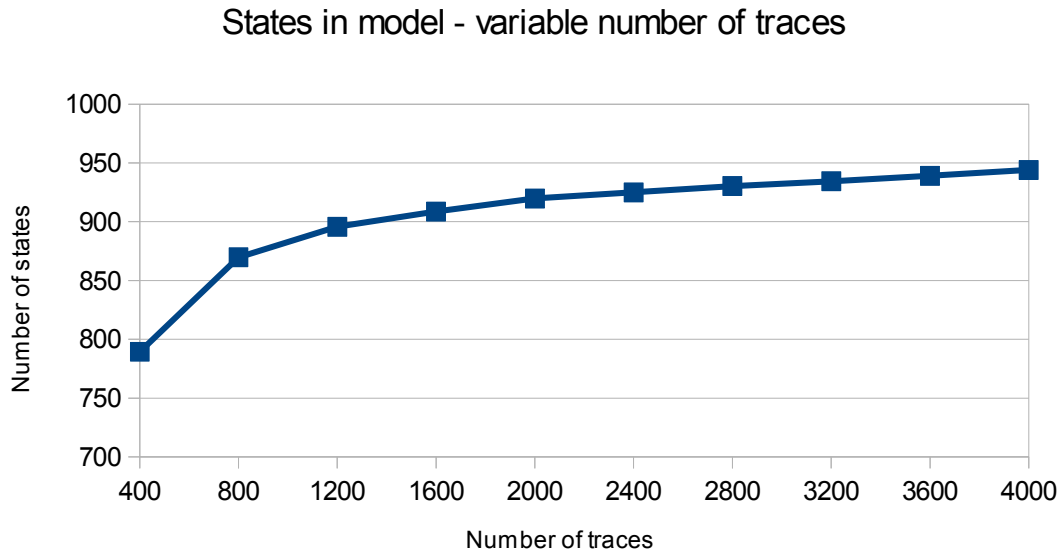


Chart 4: The trend of number of states in transition model respect to the growth in the number of traces

The third test (see Charts 5 and 6) used a set of synthetic event logs, where each log file has the number of traces and the size of the alphabet fixed, and the number of events in a trace changes.

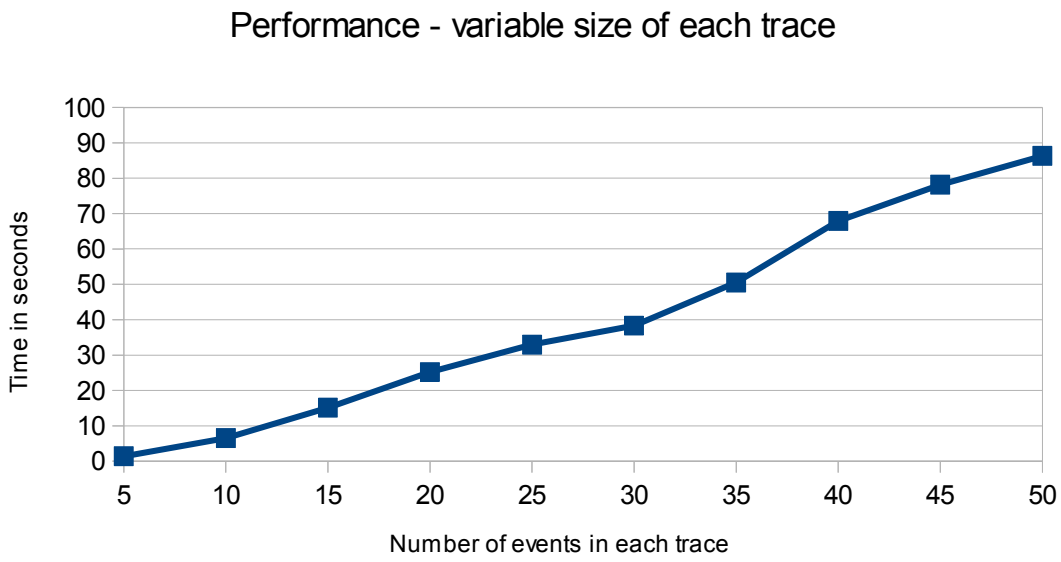


Chart 5: Performance test using variable number of events in each trace

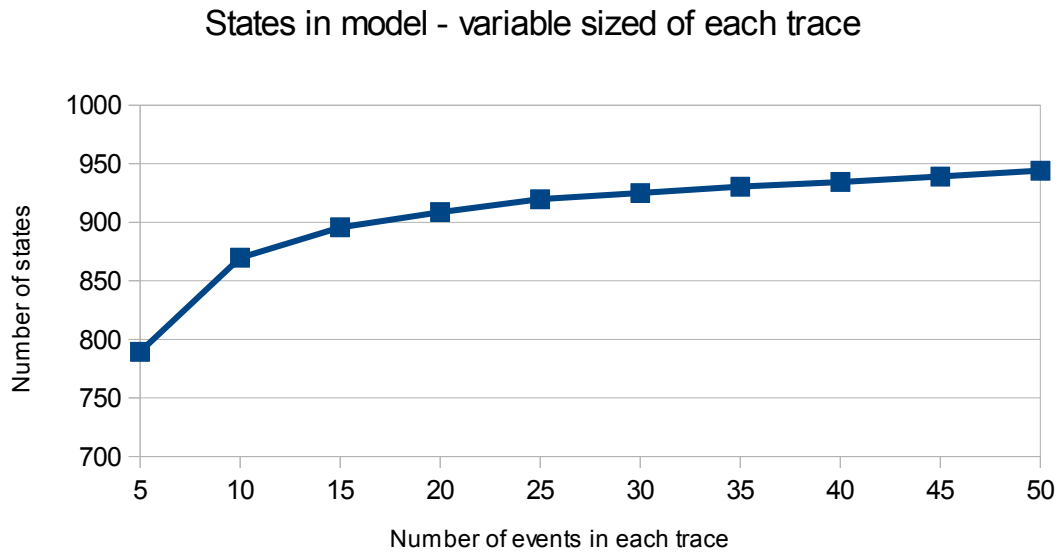


Chart 6: The trend of number of states in transition model respect to the growth in the number of events in each trace

All experiments display linear trend respect to the growth in the log and logarithmic growth in the size of the Kripke structure.

Fourth test was made to measure the performance of different queries. Chart 7 clearly shows that the most expensive query (in terms of time) is **E[**?x **U** ?y**]**. The second one is **EF**(?x) that is evaluated as **E[**true **U** ?x**]**.

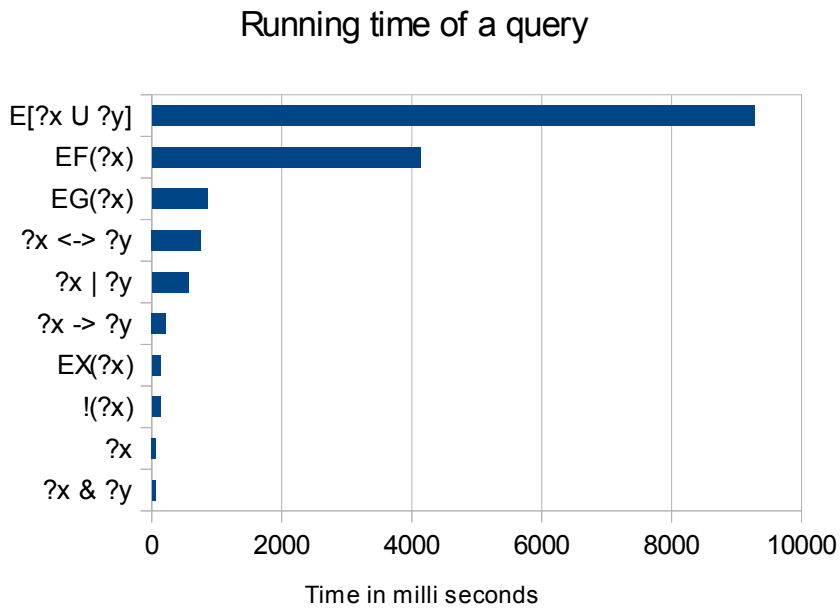
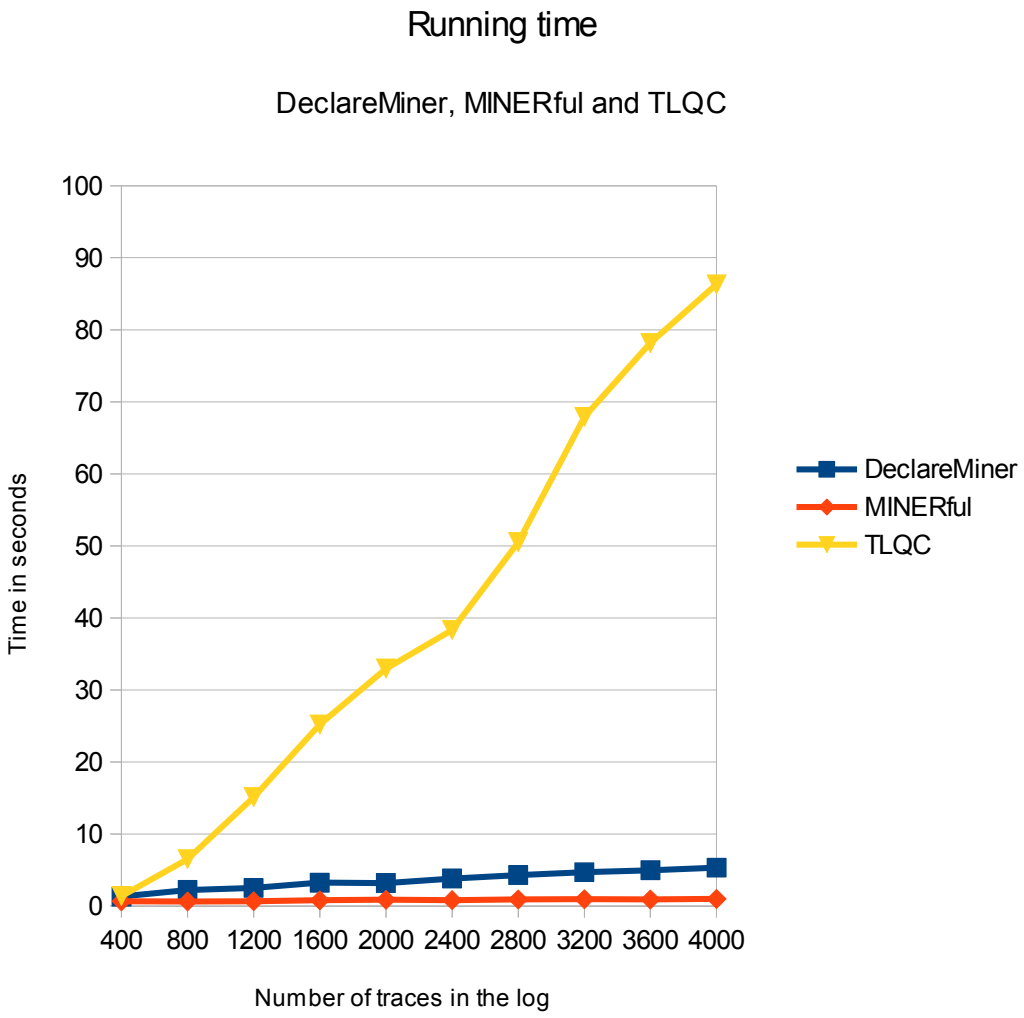


Chart 7: Running time of different queries in a log with 4000 traces, 10 event classes and 10 events in each trace



The benchmark reveals that when discovering the same set of constraints using the existing approaches with respect to the one presented in this thesis, the time difference is significant. For a log with 4000 traces, 10 event classes and 10 events in each trace, the discovery task requires 5.3 seconds with the Declare Maps Miner and 1 second for MINERful to complete the task, while our software requires 86 seconds. From this it is safe to conclude that the approach presented in this thesis do not improve the performance of mining Declare constraints with respect to existing solutions.

In addition, there is a clear performance improvement with respect to xChek: the tests with xChek, showed extremely low performance or even run out of memory.

Case Study

For the case study it was decided to take a real life event log that was given as the source log for the Business Process Intelligence Challenge (BPIC) 2012. BPIC is an event founded by the Eindhoven University of Technology. The main goal of the event is to promote process mining. The organizers provide a real life log file with sensitive information removed or obfuscated. The log file is freely available. The aim for the participants is to describe the process to which the log belongs to. There are no guidelines nor restrictions on how to process the data and what to write into the report. It is up to the participant to select the tools and methods.

The case study has been conducted based on the report of a winning entry [40]. The aim is to confirm or invalidate the results provided in the report using temporal logic query checking.

The event log is recorded by the software system for managing loan applications in a bank. The log contains 13 087 process instances spanning over approximately a six months period from October 2011 to March 2012. There are in total of 262,200 events in 23 activities. Each trace contains a single trace level attribute, AMOUNT_REQ, which indicates the amount requested by the applicant. The initial event for all traces is a submission of an application (A_SUBMITTED) and each case ends with a decision: approved, canceled, declined.

There are three categories of activities. which are prefixed in the log eith A_, O_ and W_. Events starting with A_ refer to the state of the application in the process [40].

Event class	Description
A_SUBMITTED	Initial application submission
A_PARTLYSUBMITTED	Pseudo activity. This happens always right after (within seconds) A_SUBMITTED.
A_PREACCEPTED	Application is preaccepted, but needs additional inforamtion
A_ACCEPTED	Application accepted
A_FINALIZED	Application finalized
A_APPROVED	End state of successful (approved) applications
A_REGISTERED	End state of successful (approved) applications
A_ACTIVATED	End state of successful (approved) applications
A_CANCELLED	End state of unsuccessful applications
A_DECLINED	End state of unsuccessful applications

Events starting with O_ refer to the state of the offer from bank to the customer.

Event class	Description
O_SELECTED	Applicant selected to receive offer
O_PREPARED	Offer prepared
O_SENT	Offer sent to applicant
O_SENT BACK	Offer response received from applicant
O_ACCEPTED	End state of successful offers
O_CANCELLED	End state of unsuccessful offers
O_DECLINED	End state of unsuccessful offers

Events with prefix W_ indicates the work, which is done by the clerks in the bank

Event class	Description
W_Afhandelen leads	Follow up incomplete submission
W_Completeren aanvraag	Completing pre-accepted application
W_Nabellen offertes	Follow up after offer is sent
W_Valideren aanvraag	Validating the application
W_Nabellen incomplete dossiers	Querying additional information
W_Beoordelen fraude	Investigating potential fraud
W_Wijzigen contractgegevens	Modifying approved contracts

Each event is of type Schedule, Start and Complete and a timestamp.

Event type	Description
SCHEDULE	Indicates a work item has been scheduled to occur in the future
START	Indicates the commencement of a work item
COMPLETE	Indicates the closing / conclusion of a work item

The authors in [40] used three software tools to conduct their analysis: Disco (http://fluxicon.com/disco/), Microsoft Excel (http://office.microsoft.com/en-us/excel/) and CART (http://www.salford-systems.com/products/cart).

In the report they state that A_SUBMITTED is always the initial state and is immediately followed by A_PARTLYSUBMITED. To verify this the queries ?x and EX(?x) must return A_SUBMITTED and EX(A_PARTLYSUBMITTED) with support of 100%. After running

the queries, indeed the result was as expected.

Temporal logic query checking can be used to ask questions about the process. For example querying invariants in the process – a constraint with support of 100% or what activity is always done before A_ACTIVATED. CTL queries for these questions are !(**AF**(?x) & **AF**(?y)) (see results in Table 5) and **AG**(?x → **AF**(A_ACTIVATED)). The latter did not give any results, so **EG**(?x → **EF**(A_ACTIVATED)) was queried instead (see results in Table 6).

Activity A	Activity B
A-CANCELLED#complete	A_APPROVED#complete
A-CANCELLED#complete	A_ACTIVATED#complete
A-CANCELLED#complete	O_DECLINED#complete
A-CANCELLED#complete	A_DECLINED#complete
A-CANCELLED#complete	A_REGISTERED#complete
A-CANCELLED#complete	W_Wijzigen contractgegevens#schedule
O_DECLINED#complete	O_ACCEPTED#complete
O_DECLINED#complete	A_REGISTERED#complete
O_DECLINED#complete	A_APPROVED#complete
O_DECLINED#complete	A_ACTIVATED#complete
A_DECLINED#complete	A_REGISTERED#complete
A_DECLINED#complete	W_Wijzigen contractgegevens#schedule
A_DECLINED#complete	A_ACTIVATED#complete
A_DECLINED#complete	A_APPROVED#complete

Table 5: Events which never occur together in same trace

Support	LTL expression
0.97	G(O_ACCEPTED#complete -> F(A_ACTIVATED#complete))
0.97	G(W_Nabellen incomplete dossiers#schedule -> F(A_ACTIVATED#complete))
0.97	G(W_Nabellen incomplete dossiers#start -> F(A_ACTIVATED#complete))
0.95	G(A_APPROVED#complete -> F(A_ACTIVATED#complete))
0.95	G(W_Nabellen incomplete dossiers#complete -> F(A_ACTIVATED#complete))

Table 6: Some results for AG(?x → AF(A_ACTIVATED))

Related work

There is a lot of work done to discover conventional process models (BPMN, EPC, workflow nets) from event logs compared with the work done in the context of the discovery of declarative models. In [5] Wil van der Aalst describes the state of the art of process mining, but it concentrates mostly on process mining techniques based on imperative modeling languages. There is an extensive overview in [41] for the declarative approaches in general. Pesic describes in her thesis [21] the need to use constraint based languages in business process management and introduces the Declare language.

One of the first approaches of discovering declarative model form event logs is described in [42]. The authors recognize the need of declarative models to deal with flexible processes in an organization. They use Inductive Logic Programming techniques to learn SCIFF (Social Constrained IFF) rules from event log. The final output is a declarative process model, where the SCIFF rules are mapped to Declare constraints. SCIFF provides a declarative language based on Computational Logic, where constraints are imposed on activities in terms of reactive rules. In SCIFF, an event happened at a particular time, is denoted as H(event, T), where event is a term and T is the variable for time (continuous or discrete). H(Check-in, 0) would mean, that a person was checked in to a hotel at time 0. Another concept in SCIFF is expectation – E(event, T) that means, that an event is expected to happen at time T. H(Check-in, 0) -> E(Check-out, 48), can be read „When a person is checked in, he/she is expected to check out after 2 days, if the time is expressed in hours.

SCIFF rules were developed to specify and verify interaction protocols in multi-agent systems. But these can be also used to define process rules in business process management software and in service oriented architectures –rules for orchestrating activities. SCIFF rules are also used in [12], [43] and [44] to discover process models. The drawback here is that they need also negative traces in the log file and the traces must be labeled as positive or negative beforehand.

In [12], the authors implement a software package DecMiner to show the applicability of SCIFF rules in process mining. [43] uses the work in [42] to extract integrity constraints from an event log. Then, the learned constraints are translated into Markov Logic [45] formulas and the weights of each formula are tuned using the Alchemy system[4]. The resulting theory allows

4 http://alchemy.cs.washington.edu/

for conducting probabilistic classification of traces.

In [44], the authors also uses [42] as a starting point. They introduce Incremental Process Miner that can incrementally update an existing process model, given a new set of traces. The benefit is to modify an existing model instead of building the model to incorporate new traces.

[46] and [47] discover process models using Declare templates. They use LTL expressions to define the Declare templates. In [46] the authors propose an algorithm that first generates candidate constraints from event log. Next, the candidates are evaluated against the event log –checked for conformance. A candidate constraint is considered in the final result, when it holds with respect to the event log. The application also takes a user defined set of constraints as an input, this allows the user to evaluate one or several constraints at the time. The approach in [47] enhances the algorithm in [46] by selecting the most interesting candidates using metrics similar to the ones used for association rule mining.

In [48], the authors represent the constraints in a process using a regular language [49] that can be expressed using regular expressions. They choose regular expressions over temporal logic, because temporal logic is evaluated on infinite paths, while regular expressions can represent finite path.

The authors describe an algorithm named MINERful [11], which is the basis of MailOfMine software package. By artful process they mean a process, which is not defined in detail or not defined at all. The transitions from one activity to another are decided during the execution. For example planning and scheduling a seminar event. The process depends a lot about the duration and the general topic of the seminar. The authors try to mine workflow models out of a collection of email messages. Their main goal is to capture the business process of knowledge workers, who do not follow a strict process plan and for processes for which there is no documentation. Using their technique, the process model can be mined from peoples email conversations.

Conclusion

The practical part of this thesis is a tool for Temporal Logic Query Checking. With this tool, it is possibility to discover all possible constraints that can be represented by temporal logic expression. The answer to RQ1 (Can Temporal Logic Query Checking improve the discovery of declarative process models from event logs?) is yes, improvement in speed was made respect to [13] and improvement in wider range of discoverable constraints was made respect to [10] and [11]. Answering RQ2 (Does the tool have a business value?) is a bit more difficult – the improvement in speed is still not comparable to [10] and [11], but surly there is a place for a such tool.

The future work is to conduct more thorough performance tests. The application is not handling big log files well – in the evaluation all trends respect to log growth in log size are liner, but at some point the algorithm starts to slow down. Also the algorithm for **EU** can be made faster.

References

[1] “Business Process Model and Notation,” *Wikipedia, the free encyclopedia*. 21-May-2014.

[2] “Event-driven process chain,” *Wikipedia, the free encyclopedia*. 21-May-2014.

[3] “Activity diagram,” *Wikipedia, the free encyclopedia*. 21-May-2014.

[4] “Petri net,” *Wikipedia, the free encyclopedia*. 21-May-2014.

[5] W. M. Van der Aalst, *Discovery, Conformance and Enhancement of Business Processes*. Springer, 2011.

[6] W. van der Aalst, A. Adriansyah, A. K. A. de Medeiros, F. Arcieri, T. Baier, T. Blickle, J. C. Bose, P. van den Brand, R. Brandtjen, and J. Buijs, “Process mining manifesto,” presented at the Business process management workshops, 2012, pp. 169–194.

[7] “Temporal logic,” *Wikipedia, the free encyclopedia*. 01-May-2014.

[8] “Regular expression,” *Wikipedia, the free encyclopedia*. 23-May-2014.

[9] “Logic programming,” *Wikipedia, the free encyclopedia*. 24-May-2014.

[10] F. M. Maggi, R. J. C. Bose, and W. M. van der Aalst, “A knowledge-based integrated approach for discovering and repairing declare maps,” presented at the Advanced Information Systems Engineering, 2013, pp. 433–448.

[11] C. Di Ciccio and M. Mecella, “MINERful, a mining algorithm for declarative process constraints in MailOfMine,” *Dep. Comput. Syst. Sci. Antonio Ruberti Tech. Rep.*, vol. 4, no. 3, 2012.

[12] F. Chesani, E. Lamma, P. Mello, M. Montali, F. Riguzzi, and S. Storari, “Exploiting Inductive Logic Programming Techniques for Declarative Process Mining,” in *Transactions on Petri Nets and Other Models of Concurrency II*, vol. 5460, K. Jensen and W. P. Aalst, Eds. Springer Berlin Heidelberg, 2009, pp. 278–295.

[13] A. Gurfinkel, M. Chechik, and B. Devereux, “Temporal logic query checking: A tool for model exploration,” *Softw. Eng. IEEE Trans. On*, vol. 29, no. 10, pp. 898–914, 2003.

[14] H. A. Reijers, T. Slaats, and C. Stahl, “Declarative Modeling–An Academic Dream or the Future for BPM?,” in *Business Process Management*, Springer, 2013, pp. 307–322.

[15] “Sequential Pattern Mining,” *Wikipedia, the free encyclopedia*. 21-Apr-2014.

[16] N. Méger and C. Rigotti, “Constraint-based mining of episode rules and optimal window sizes,” in *Knowledge Discovery in Databases: PKDD 2004*, Springer, 2004, pp. 313–324.

[17] B. F. van Dongen and W. M. van der Aalst, “A Meta Model for Process Mining Data.,”

EMOI-INTEROP, vol. 160, p. 30, 2005.

[18] C. W. Günther and E. Verbeek, “Xes standard definition,” *Fluxicon Process Lab.*, pp. 13–14, 2009.

[19] “The ProM Import Framework.” [Online]. Available: http://www.promtools.org/promimport/. [Accessed: 26-May-2014].

[20] H. Verbeek, J. C. Buijs, B. F. Van Dongen, and W. M. Van Der Aalst, “XES, xESame, and proM 6,” in *Information Systems Evolution*, Springer, 2011, pp. 60–75.

[21] M. Pesic, “Constraint-based workflow management systems: shifting control to users,” 2008.

[22] “Business Process Execution Language,” *Wikipedia, the free encyclopedia*. 24-May-2014.

[23] “Linear temporal logic,” *Wikipedia, the free encyclopedia*. 21-Apr-2014.

[24] “Computation tree logic,” *Wikipedia, the free encyclopedia*. 26-Apr-2014.

[25] “CTL*,” *Wikipedia, the free encyclopedia*. 20-Feb-2014.

[26] S. Merz, “Model checking: A tutorial overview,” in *Modeling and verification of parallel processes*, Springer, 2001, pp. 3–38.

[27] “List of software bugs,” *Wikipedia, the free encyclopedia*. 05-May-2014.

[28] “Kripke structure (model checking),” *Wikipedia, the free encyclopedia*. 18-Apr-2014.

[29] W. Chan, “Temporal-logic queries,” presented at the Computer Aided Verification, 2000, pp. 450–463.

[30] G. Bruns and P. Godefroid, “Temporal logic query checking,” presented at the Logic in Computer Science, 2001. Proceedings. 16th Annual IEEE Symposium on, 2001, pp. 409–417.

[31] O. Kupferman, M. Y. Vardi, and P. Wolper, “An automata-theoretic approach to branching-time model checking,” *J. ACM JACM*, vol. 47, no. 2, pp. 312–360, 2000.

[32] S. Hornus and P. Schnoebelen, “On solving temporal logic queries,” in *Algebraic Methodology and Software Technology*, Springer, 2002, pp. 163–177.

[33] “dk.brics.automaton - finite-state automata and regular expressions for Java.” [Online]. Available: http://www.brics.dk/automaton/. [Accessed: 25-May-2014].

[34] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella, “Nusmv 2: An opensource tool for symbolic model checking,” presented at the Computer Aided Verification, 2002, pp. 359–364.

[35] J. Simmonds and A. Gurfinkel, “χChek RC2 User Manual,” 2007. [Online]. Available: http://www.cs.toronto.edu/~arie/xchek/xchek_user_manual.pdf. [Accessed: 25-May-2014].

[36] “XML,” *Wikipedia, the free encyclopedia*. 24-May-2014.

[37] “Decision tree,” *Wikipedia, the free encyclopedia*. 23-May-2014.

[38] “Binary decision diagram,” *Wikipedia, the free encyclopedia*. 11-May-2014.

[39] “buddy | Free Science & Engineering software downloads at SourceForge.net.” [Online]. Available: http://sourceforge.net/projects/buddy/. [Accessed: 25-May-2014].

[40] A. Bautista, L. Wangikar, and S. Akbar, “Process Mining-Driven Optimization of a Consumer Loan Approvals Process,” 2012.

[41] S. Goedertier, “Declarative techniques for modeling and mining business processes.,” 2008.

[42] E. Lamma, P. Mello, F. Riguzzi, and S. Storari, “Applying inductive logic programming to process mining,” in *Inductive Logic Programming*, Springer, 2008, pp. 132–146.

[43] E. Bellodi, F. Riguzzi, and E. Lamma, “Probabilistic declarative process mining,” in *Knowledge Science, Engineering and Management*, Springer, 2010, pp. 292–303.

[44] M. Cattafi, E. Lamma, F. Riguzzi, and S. Storari, “Incremental declarative process mining,” in *Smart Information and Knowledge Management*, Springer, 2010, pp. 103–127.

[45] “Markov logic network,” *Wikipedia, the free encyclopedia*. 17-May-2014.

[46] F. M. Maggi, A. J. Mooij, and W. M. van der Aalst, “User-guided discovery of declarative process models,” presented at the Computational Intelligence and Data Mining (CIDM), 2011 IEEE Symposium on, 2011, pp. 192–199.

[47] F. M. Maggi, R. J. C. Bose, and W. M. van der Aalst, “Efficient discovery of understandable declarative process models from event logs,” presented at the Advanced Information Systems Engineering, 2012, pp. 270–285.

[48] C. Di Ciccio and M. Mecella, “Mining constraints for artful processes,” presented at the Business Information Systems, 2012, pp. 11–23.

[49] “Regular language,” *Wikipedia, the free encyclopedia*. 23-May-2014.

Appendix

I. Source code

Find source code and synthetic log files for the application at https://github.com/r2im/pickaxe.

II. License

Non-exclusive license to reproduce thesis and make thesis public

I, **Margus Räim** (23.01.1981),

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 Declarative Process Models from Event Logs through Temporal Logic Query Checking,

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, **26.05.2014**