User-Guided Discovery of Declarative Process Models

Fabrizio M. Maggi, Arjan J. Mooij and Wil M.P. van der Aalst
Department of Mathematics and Computer Science, Eindhoven University of Technology - The Netherlands
Email: {f.m.maggi, a.j.mooij, w.m.p.v.d.aalst}@tue.nl

***Abstract*—Process mining techniques can be used to effectively discover process models from logs with example behaviour. Cross-correlating a discovered model with information in the log can be used to improve the underlying process. However, existing process discovery techniques have two important drawbacks. The produced models tend to be large and complex, especially in flexible environments where process executions involve multiple alternatives. This "overload" of information is caused by the fact that traditional discovery techniques construct procedural models explicitly showing all possible behaviours. Moreover, existing techniques offer limited possibilities to guide the mining process towards specific properties of interest.**

These problems can be solved by discovering declarative models. Using a declarative model, the discovered process behaviour is described as a (compact) set of rules. Moreover, the discovery of such models can easily be guided in terms of rule templates.

This paper uses DECLARE, a declarative language that provides more flexibility than conventional procedural notations such as BPMN, Petri nets, UML ADs, EPCs and BPEL. We present an approach to automatically discover DECLARE models. This has been implemented in the process mining tool ProM. Our approach and toolset have been applied to a case study provided by the company Thales in the domain of maritime safety and security.

I. Introduction

More and more event data become available as information technology becomes more pervasive. The tight coupling between processes and supporting information systems is generating unprecedented amounts of data. Logs provide detailed information about systems and human behaviour. Therefore, it is possible to evaluate whether observed behaviour is consistent with pre-defined standards or not.

A log consists of a set of process instances, and each process instance is described by a sequence of events. Often a log also contains further information. It can specify, for instance, a timestamp to indicate the time when an event has been recorded, an originator, i.e., the agent (human or system application) triggering the event, and other additional data.

Over the last decade, a variety of techniques and algorithms has been proposed for mining process models from logs [1]. These methods demonstrate that logs can be used to construct models underlying a process execution from scratch (i.e., process discovery [2] [3] [4] [5]), or to identify discrepancies between logs and a given predefined process model (i.e., conformance testing [6]).

Traditional discovery techniques focus on the extraction of procedural models where all possible orderings of events must be specified explicitly. A consequence of this characteristic is that when applying them to real life logs (especially if generated in environments with a lot of variability), they often produce spaghetti-like models that tend to be completely unreadable.

In the literature, several approaches are described [4] [7] to filter the information contained in a log and to simplify less-structured processes. However, they do not allow analysts to guide the discovery process to specific properties they are interested in, e.g., events that cannot occur in sequence, events that always coexist in the same process instance, events that must eventually occur when a given event "a" occurs etc. These properties are simple and help to compactly represent complex behaviours by focusing on specific aspects.

Constraint-based process modeling aims at representing process models in a declarative way. Instead of explicitly specifying all the allowed sequences of events in a business process, the possible ordering of events is implicitly specified with constraints, i.e., rules that must be followed during execution. Anything that does not permanently violate these constraints is possible.

In this paper, we use the declarative language DECLARE [8] [9] [10], which is characterised by templates and which is based on LTL semantics. We propose an approach for the discovery of DECLARE models allowing analysts to specify which kinds of templates they are interested in. This feature allows analysts to shape the discovery process to extract the properties that are most relevant for them.

Recent publications [11] [12] [15] introduce other techniques for declarative process discovery. In these works, SCIFF, an extension of logic programming, is used to specify declarative process models. In particular, the algorithm from [15] repeatedly performs a beam search on all candidate constraints. Each iteration of the algorithm produces a constraint that best fits both the positive traces and the negative traces that are not excluded by the previously discovered constraints. The main difference between these techniques and our approach is that process discovery using SCIFF is based on the assumption that both compliant and non-compliant traces of execution are provided. In contrast, our approach can also be applied when only positive interpretations are available. This represents an advantage of our discovery technique, since, in real life, logs hardly provide clearly-marked negative information.

Apriori-like approaches such as sequence mining [13] and episode mining [14] can discover local patterns in a log, but they cannot generate an overall process model from it.

978-1-4244-9927-4/11/$26.00 ©2011 IEEE

Using such approaches, it is not possible to discover rules representing negative behaviours (what should not happen) and choices. In general, the LTL rules underlying DECLARE templates have more expressive power than the sequences used in sequence mining and the partial orders used in episode mining.

Our proposed approach has been implemented as a plug-in in the process mining tool ProM. This plug-in has been applied to a case study in the domain of maritime safety and security. In this case study, we discover the behaviour of several types of vessels starting from the data recorded by electronic sensors.

Based on practical experiences in this case study, we developed a new mechanism to deal with the noise. Moreover, these experiences also triggered adaptations to the algorithm in order to provide a good performance. The case study also showed that, in order to obtain significant results, it is necessary to address two additional requirements.

First of all, a log is often composed of prefixes of larger process instances. This characteristic can affect the discovered DECLARE models because some constraints can be temporarily violated on the available part of a process instance but satisfied on its continuation. To solve this problem we propose to apply the *truncated semantics* introduced in [19] to discover significant DECLARE constraints also from truncated process instances.

Secondly, using the standard LTL semantics, a DECLARE constraint is discovered when it is non-violated so that it is also discovered when it is trivially valid, i.e., it is independent of the process instances in the log. In this case, the discovered constraint does not capture the desired behaviour. We propose to use techniques for *LTL vacuity detection* [21] [22] [20] to discriminate between instances where a constraint is generically non-violated and instances where the constraint is non-trivially valid.

This paper is organised as follows. Section II introduces the DECLARE formalism. Section III describes the main features of our approach and its implementation in ProM. Section IV explains how truncated LTL semantics and LTL vacuity detection can be used to improve the discovery process. Section V provides an illustrative case study. Section VI concludes the paper.

II. Preliminaries

DECLARE is a declarative language proposed by Pesic and Van der Aalst in [8] [9]. The language has been developed to fulfill two important criteria for a process modeling language: it must be understandable for end-users and it must have a formal semantics in order to be verifiable and executable.

A DECLARE model consists of a set of constraints which, in turn, are based on templates. Templates are abstract entities that define parameterised classes of properties, and constraints are their concrete instantiations. Templates have a user-friendly graphical representation understandable to the user and their semantics are specified through LTL formulas (see TABLE I for the semantics of the LTL operators). Each constraint inherits the graphical representation and semantics from its template. These features allow DECLARE to meet both criteria mentioned before.

TABLE I
LTL Operators Semantics

operator	***semantics***
$\bigcirc\varphi$	*φ has to hold in the next position of a path.*
$\Box\varphi$	*φ has to hold always in the subsequent positions of a path.*
$\Diamond\varphi$	*φ has to hold eventually (somewhere) in the subsequent positions of a path.*
$\varphi\ U\psi$	*φ has to hold in a path at least until ψ holds. ψ must hold in the current or in a future position.*

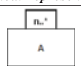
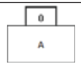
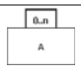
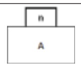
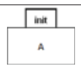
DECLARE is a declarative language, because instead of explicitly specifying the flow of the interactions among process events, it describes a set of constraints which must be satisfied throughout the process execution. In comparison with procedural approaches that produce "closed" models, i.e., all what is not explicitly specified is forbidden, DECLARE models are "open" and tend to offer more possibilities for execution. In this way, DECLARE supports flexibility.

The DECLARE templates characterise the language and highlight different features which are worth being specified in a process model. In particular, the templates are classified according to four groups: *existence*, *relation*, *negative relation* and *choice*. The first three groups are described in the following subsections. For sake of brevity, we do not describe the choice templates and refer to [10] for more information about the DECLARE templates.

A. Existence Templates

The existence templates involve only one event (unary relationship) and define the cardinality or the position of an event in a process instance. Templates of the type $existence(n, A)$ specify that A should occur *at least n* times in a process instance. In contrast, templates of the type $absence(n+1, A)$ specify that A should occur *at most n* times. Templates $exactly(n, A)$ indicate that A should occur *exactly n* times. Finally, $init(A)$ specifies that each process instance should start with event A. The graphical notation and LTL semantics are shown in TABLE II.

TABLE II
Existence Templates

name of template	***LTL semantics***	***graphical representation***
$existence(1, A)$ $existence(2, A)$ ... $existence(n, A)$	$\Diamond A$ $\Diamond(A \wedge \bigcirc(existence(1, A)))$ ... $\Diamond(A \wedge \bigcirc(existence(n-1, A)))$	
$absence(A)$	$\neg existence(1, A)$	
$absence(2, A)$ $absence(3, A)$ ... $absence(n+1, A)$	$\neg existence(2, A)$ $\neg existence(3, A)$ ... $\neg existence(n+1, A)$	
$exactly(1, A)$ $exactly(2, A)$ ... $exactly(n, A)$	$existence(1, A) \wedge absence(2, A)$ $existence(2, A) \wedge absence(3, A)$ ... $existence(n, A) \wedge absence(n+1, A)$	
$init(A)$	A	

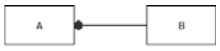
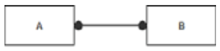
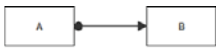
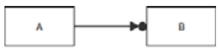
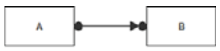
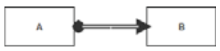
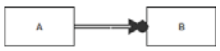
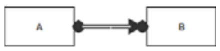
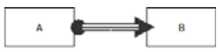
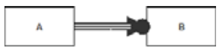
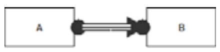
B. Relation Templates

Whereas an existence template describes the cardinality of one event, a relation template defines a dependency between two events. The $responded\ existence(A,B)$ template specifies that if event *A* occurs, event *B* should also occur (either before or after event *A*). The $co\text{-}existence(A,B)$ template specifies that if one of the events *A* or *B* occurs, the other one should also occur.

If event *B* is the $response$ of event *A*, then when event A occurs, event B should eventually occur after A. In contrast, the $precedence(A,B)$ template indicates that event *B* should occur only if event *A* has occurred before. Finally, the $succession(A,B)$ template requires that both response and precedence relations hold between the events *A* and *B*.

Templates alternate response, alternate precedence and alternate succession strengthen the above templates by specifying that events must alternate without repetitions of these events in between. Even more strict ordering relations are specified by templates chain response, chain precedence and chain succession. These templates require that the occurrences of the two events (A and B) are next to each other. See TABLE III for notation and semantics.

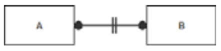
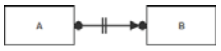
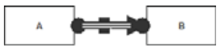
TABLE III
RELATION TEMPLATES

name of template	*LTL semantics*	*graphical representation*
$responded\ existence(A,B)$	$\Diamond A \Rightarrow \Diamond B$	
$co\text{-}existence(A,B)$	$\Diamond A \Leftrightarrow \Diamond B$	
$response(A,B)$	$\Box(A \Rightarrow \Diamond B)$	
$precedence(A,B)$	$(\neg B\ U A) \vee \Box(\neg B)$	
$succession(A,B)$	$response(A,B) \wedge precedence(A,B)$	
$alternate\ response(A,B)$	$\Box(A \Rightarrow \bigcirc(\neg A\ U B))$	
$alternate\ precedence(A,B)$	$precedence(A,B) \wedge \Box(B \Rightarrow \bigcirc(precedence(A,B)))$	
$alternate\ succession(A,B)$	$alternate\ response(A,B) \wedge alternate\ precedence(A,B)$	
$chain\ response(A,B)$	$\Box(A \Rightarrow \bigcirc B)$	
$chain\ precedence(A,B)$	$\Box(\bigcirc B \Rightarrow A)$	
$chain\ succession(A,B)$	$\Box(A \Leftrightarrow \bigcirc B)$	

C. Negative Relation Templates

The $not\ co\text{-}existence(A,B)$ template indicates that event *A* and *B* cannot occur together in the same process instance. According to the $not\ succession(A,B)$ template any occurrence of *A* cannot be followed eventually by *B*. Finally, according to the $not\ chain\ succession(A,B)$, *A* cannot be directly followed by *B*. Negative relation templates are summarised in TABLE IV.

TABLE IV
NEGATIVE RELATION TEMPLATES

name of template	*LTL semantics*	*graphical representation*
$not\ co\text{-}existence(A,B)$	$\neg(\Diamond A \wedge \Diamond B)$	
$not\ succession(A,B)$	$\Box(A \Rightarrow \neg(\Diamond B))$	
$not\ chain\ succession(A,B)$	$\Box(A \Rightarrow \bigcirc(\neg B))$	

III. APPROACH

The starting point of our work was a case study concerned with the monitoring of vessel behaviour in the domain of maritime safety and security. Fig. 1 shows how our approach to mining DECLARE models is embedded in the general approach used for the case study.

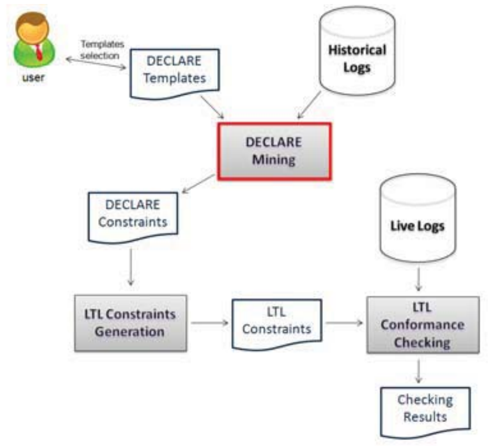


Fig. 1. DECLARE Mining within a general approach for vessel monitoring

Our proposed approach for the discovery of DECLARE models is used in the first phase of the case study to build a declarative reference model (representing the normal behaviour of vessels) starting from historical logs. In this phase, users can specify which DECLARE templates will be used to generate the DECLARE constraints in the discovered model. Accordingly, they can choose which aspects of the vessel behaviour they want to highlight through the discovery process. The discovered DECLARE constraints can be validated and improved by domain experts to completely fit their needs and build a proper reference model.

In a second phase (LTL constraints generation), the resulting DECLARE constraints are translated into LTL constraints. In the last phase (LTL conformance checking), live logs from vessels are checked w.r.t. the obtained LTL constraints by applying approaches for static or run-time LTL checking [16] [17].

In the remainder of this section, we present our approach for the DECLARE mining. First, we introduce a core algorithm to discover DECLARE models, then we extend it with some additional parameters and describe the implementation in ProM.

A. Core Discovery Algorithm

For discovering DECLARE models we need (1) a set T of DECLARE templates and (2) a (historical) log W. The first step of our discovery algorithm is to generate a DECLARE model $D_{candidates}$ consisting of candidate DECLARE constraints. Let $E = \{e_1, .., e_n\}$ be the set of event classes belonging to W. $D_{candidates}$ is generated by instantiating each DECLARE template $t(a_1, .., a_k)$ in T with all the possible dispositions of length k of the n event classes $e_1, .., e_n$. Each template with k parameters produces n^k potential constraints in the model so that the number of constraints in $D_{candidates}$ is $h = \sum_{t \in T} n^{k_t}$ where k_t is the number of parameters of t.

In the second step of the discovery algorithm, $D_{candidates}$ is translated into an LTL model $L_{candidates}$. This model is composed of a list of LTL rules corresponding to $D_{candidates}$.

Each rule l in $L_{candidates}$ is then checked w.r.t. W (we use the algorithm described in [17]) to decide whether it is satisfied in W or not. If l is not satisfied, it is removed from the model. At the end of the checking phase, a filtered LTL model L including the remaining LTL rules is available.

Finally, the model L is translated into a DECLARE model D which is the result of the discovery process.

B. Additional Mining Parameters

To apply the algorithm from subsection III-A to real life logs, we need to deal with the time complexity of the algorithm and the noise in the logs. To address these issues, we introduce some additional parameters to tune the discovery process.

The parameter *Percentage of Events (PoE)* can be used to avoid the discovery of less-relevant constraints referring to event classes which rarely occur in the log. This parameter specifies the percentage of the event classes to be used to generate the candidate constraints. For instance, if PoE = 50% the discovered constraints will only involve 50% of the event classes in the log (the most frequent ones). This parameter has also a positive effect on the time complexity of the algorithm. The number of candidate constraints in $D_{candidates}$, as explained in subsection III-A, can be very large. For instance, for a log including 30 event classes and a single template with 4 parameters, 810.000 candidate constraints are generated and checked. Using PoE = 50%, the number of event classes is reduced to the 15 most frequently-occurring ones; thus, we only need to consider 50.625 candidate constraints, 16 times less than before.

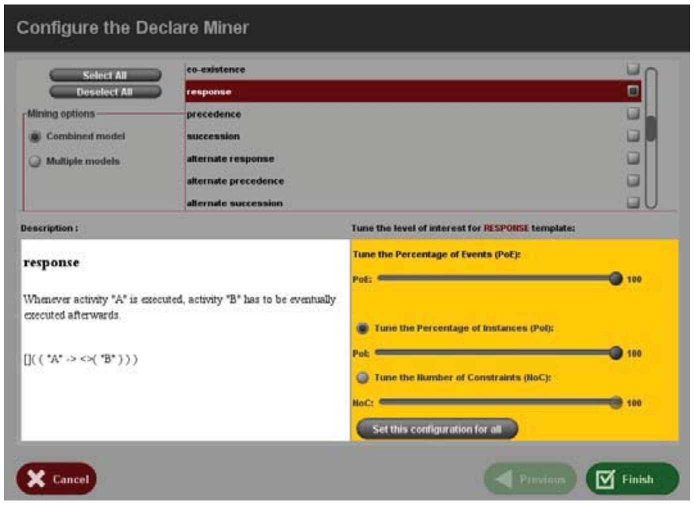
The parameter *Percentage of Instances (PoI)* can be used to specify that a DECLARE constraint can still be discovered even if it does not hold for all process instances of the log. For instance, if PoI = 80%, a constraint will be discovered if at least 80% of the process instances satisfy the constraint. This parameter is useful in case of noisy logs, where rules are violated in exceptional cases, but hold for most cases.

C. Implementation

All phases of the approach shown in Fig. 1 are supported by ProM plug-ins[1].

[1]http://www.win.tue.nl/declare/declare-miner/

The DECLARE mining phase is implemented as the DECLARE Miner. The DECLARE Miner takes as input two ProM objects representing the (historical) log and the set of available DECLARE templates. Before starting the discovery, the DECLARE miner allows users to specify which templates will be used to generate the DECLARE constraints in the discovered model. Moreover, users can set (for each template) the parameters described in subsection III-B to tune the discovery process according to their specific needs (Fig. 2). The DECLARE Miner generates a DECLARE model object by using the algorithm described in subsection III-A.



Mining options
[✓ Combined model] Multiple models

Description :

response

Whenever activity "A" is executed, activity "B" has to be eventually executed afterwards.

[](("A" -> <>("B")))

Tune the level of interest for RESPONSE template:

Tune the Percentage of Events (PoE):
PoE: 100

[✓ Tune the Percentage of Instances (PoI):]
PoI: 100

Tune the Number of Constraints (NoC):
NoC: 100

Fig. 2. GUI of the DECLARE Miner

The LTL constraints generation phase is supported by the DECLARE2LTL plug-in. It takes as input a DECLARE model object and generates the corresponding LTL model object. This object can be used as an input of the LTL Checker to check the conformance of live logs w.r.t. the discovered model (LTL conformance checking phase).

IV. Advanced Mining Techniques

In this section, we introduce two advanced techniques to support the discovery of DECLARE models: the *truncated semantics* for LTL formulas and the *LTL vacuity detection*. These techniques are typically applied in the field of model checking, but we have modified them for process discovery.

A. Truncated Semantics

In our core algorithm, a (candidate) constraint is discovered if it is satisfied in a given percentage of the process instances. However, often the available logs are extracted from larger logs and the process instances are prefixes of larger process instances. This affects the discovered constraints.

For instance, the semantics of the chain response template is defined as:

$$\Box(A \Rightarrow \bigcirc B).$$

This means that, for every occurrence of A, a next event exists and this event is B. If A is the last event of the process instance, there is no next event and this LTL formula is, in any case, not satisfied. If the process instance is a prefix of a larger process instance, using this semantics a constraint can be temporarily violated on the available part of the process instance but satisfied on its continuation. As a result, when partial logs are used as input, some constraints remain undiscovered.

The problem of deciding the truth value of an LTL formula on a truncated path is addressed in [19]. In a truncated path the truth value of an LTL formula can be non-definitive (i.e., temporarily violated or temporarily satisfied). For instance, the formula $\Box A$ is either violated or temporarily satisfied in any truncated path; similarly, the formula $\Diamond A$ is either satisfied or temporarily violated. To address this problem, [19] introduces a *strong semantics* and a *weak semantics* for LTL formulas where a formula is evaluated to false and true respectively if its truth value is non-definitive. In the traditional LTL semantics (called *neutral semantics*), a formula is evaluated to true if it is temporarily satisfied and it is evaluated to false if it is temporarily violated.

Using the weak semantics for the chain response template instead of the neutral one, a chain response constraint is considered satisfied in a process instance also if its truth value is non-definitive. In this way, it is possible to weaken the acceptance criterion used to filter the list of candidate constraints in our discovery algorithm which results in an increased number of discovered constraints.

According to the *strength relation theorem* [19], the strong semantics implies the neutral semantics, which, in turn, implies the weak semantics. From this theorem it follows that using the strong semantics we have maximal reliability about the satisfaction of a constraint; on the other hand, using the weak semantics, we have more flexibility in the discovery process. The weak LTL semantics seems to give more significant results when using logs containing truncated process instances.

B. Vacuity Detection

A constraint is vacuously satisfied for an instance, if the constraint is not really "activated". Formally, a formula φ is *vacuously satisfied* in a path π, if π satisfies φ and there is some sub-formula of φ that does not affect the truth value of φ in π [21]. A typical example of a vacuously satisfied constraint is given by the formula "every request is eventually acknowledged" in a process instance that does not contain requests.

In [22], Kupferman and Vardi propose a general method for detection of vacuity and generation of interesting witnesses for specifications in LTL. A path π is an *interesting witness* for a formula φ if π satisfies φ non-vacuously [21]. Interesting witnesses are identified by checking the formula $witness(\varphi)$ defined in [22]. In particular, a path π is an interesting witness for φ, if π satisfies $witness(\varphi)$.

In the process discovery context, interesting witnesses are process instances where a constraint is non-vacuously satisfied. We need to be careful when combining this with the weak LTL semantics. For instance, using weak LTL semantics the formula "every request is eventually acknowledged" is non-vacuously satisfied in a process instance that contains only one request and that does not contain acknowledgements. To combine the discovery of non-vacuously satisfied constraints with the benefits gained using the weak LTL semantics, we call a (truncated) process instance an interesting witnesses for a constraint φ if:

- using the weak LTL semantics, the process instance satisfies φ, and
- using the neutral LTL semantics, some prefix of the (truncated) process instance satisfies φ non-vacuously (i.e., it satisfies $witness(\varphi)$).

A truncated process instance is an interesting witness for the formula "every request is eventually acknowledged" if it contains at least a request followed by an acknowledgement.

V. Case Study

In this section, we present the results of the DECLARE mining phase of a larger case study on the monitoring of vessel behaviour. The case study has been provided by Thales, a global electronics company delivering mission-critical information systems and services for the Aerospace, Defense, and Security markets. In our experiments, we use a (historical) log describing the behaviour of different types of vessels. The log is recorded through maritime AIS (Automatic Identification System, [18]) receivers. Every vessel has an on-board AIS transponder that uses several message types and reporting frequencies to broadcast information about the vessel. An AIS receiver collects these broadcasted AIS messages, and produces a TCP/IP stream of messages.

In the considered log, each process instance corresponds to a specific vessel. An event is a change in the navigational status of a vessel (e.g., *moored*, *under way using engine*, *at anchor*, *under way sailing*, *not defined*). The log is an extract of a larger log and corresponds to a period of one week.

Vessels are expected to behave differently depending on their type. Therefore, the first step of our experimentation consists of splitting the log by vessel type (e.g., *passenger ship*, *fishing boat*, *cargo/hazard pollutant A vessel*, *cargo/hazard pollutant C vessel*, *pilot boat*, *tanker*). The result of this pre-processing phase is a set of sub-logs where each sub-log contains the instances for one vessel type. Starting from each sub-log, the DECLARE miner is used to construct a DECLARE model representing the observed behaviour of the related vessel type.

A. Dealing with Truncated Process Instances

1) Passenger Ships: TABLE V shows the settings for an experiment aimed at discovering chain response constraints for vessel type *passenger ship* with a PoI of 100% (i.e., constraints satisfied by all instances).

TABLE V
Experiment 1: Experimental Settings

experiment number	*template*	*PoI*	*PoE*
1	*chain response*	100%	100%

According to the *traditional LTL semantics* of the chain response template, this experiment produces no results. However, if we look at the log, it is composed of a regular alternation of events *under way using engine* and *moored*. Yet no chain response constrains are discovered due to the phenomenon described in subsection IV-A; process instances are truncated.

In Fig. 3, the results for experiment 1 (with the same experimental settings) using the *weak LTL semantics* are shown. These results reflect the alternation of events *under way using engine* and *moored* characterizing the sub-log. In the following experiments, we always use the weak LTL semantics.

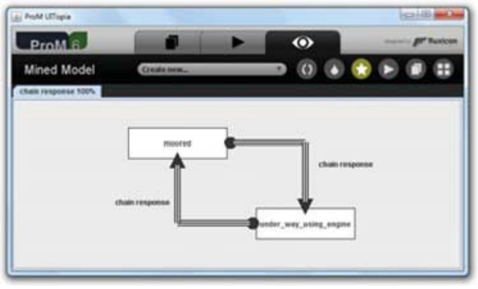


Fig. 3. Experiment 1: discovered chain response constraints

B. Discovery of Non-Vacuously Satisfied Constraints

1) Fishing Boats: In TABLE VI, the settings for an experiment to discover chain response constraints for a *fishing boat* is shown. TABLE VII shows the results for this experiment.

TABLE VI
EXPERIMENT 2: EXPERIMENTAL SETTINGS

experiment number	*template*	*PoI*	*PoE*
2	*chain response*	100%	100%

TABLE VII
EXPERIMENT 2: RESULTS

discovered constraint	*PoI*
chain response(not defined, not defined)	100%
chain response(not defined, under way using engine)	100%
chain response(not defined, under way sailing)	100%
chain response(not defined, moored)	100%
chain response(not defined, at anchor)	100%
chain response(under way sailing, under way sailing)	100%
chain response(under way sailing, under way using engine)	100%
chain response(under way sailing, not defined)	100%
chain response(under way sailing, moored)	100%
chain response(under way sailing, at anchor)	100%
chain response(at anchor, under way using engine)	100%
chain response(moored, under way using engine)	100%

In this case, we have a very unusual result because, for instance, if *chain response*(*not defined, not defined*) is satisfied in 100% of the instances, you would not expect that also *chain response*(*not defined, under way using engine*), *chain response*(*not defined, under way sailing*), *chain response*(*not defined, moored*), and *chain response*(*not defined, at anchor*) can be satisfied in 100% of the instances. Moreover, event *under way sailing* has the same behaviour.

If we look at the log, it contains, for this vessel type, a process instance consisting of only one event *not defined* and a process instance consisting of only one event *under way sailing*. In the remaining instances, these events never occur. For this reason, using the weak LTL semantics for the chain response template, events *not defined* and *under way sailing* can be associated to every other event.

As explained in subsection IV-B, the problem in this case is that a constraint is discovered also when it is *vacuously satisfied*. According to [22], for the LTL semantics of the chain response template

$$\varphi = \Box(A \Rightarrow \bigcirc B),$$
$$witness(\varphi) = \varphi \wedge \Diamond A.$$

Therefore, using the definition given in subsection IV-B, a (truncated) process instance is an interesting witnesses for *chain response*$(A,\ B)$ if at least once A is directly followed by B. In the following experiments, we always refer to this definition.

If we discover the chain response constraints for which interesting witnesses exist, we obtain the results shown in TABLE VIII. In this table, we specify the percentage of non-vacuously satisfied instances, i.e., interesting witnesses, and also the PoI parameter, i.e., the percentage of process instances where the constraint is (vacuously or non-vacuously) satisfied.

TABLE VIII
EXPERIMENT 2: INTERESTING WITNESSES

discovered constraint	*interesting witnesses*	*PoI*
chain response(under way using engine, moored)	14%	95%
chain response(moored, under way using engine)	10%	100%
chain response(at anchor, under way using engine)	5%	100%

For example, constraint *chain response*(*moored, under way using engine*) is satisfied for 100% of the process instances, but only 10% of them is an interesting witness. This percentage can be used to select the most interesting constraints.

C. User-Guided Discovery of DECLARE Constraints

In this subsection, we show how our approach allows users to guide the discovery process towards a wide range of properties.

1) Cargo/Hazard Pollutant C Vessels: TABLE IX shows the settings for an experiment aimed at discovering which events do not co-exist in the same process instance for vessel type *cargo/hazard pollutant C*.

TABLE IX
EXPERIMENT 3: EXPERIMENTAL SETTINGS

experiment number	*template*	*PoI*	*PoE*
3	*not co-existence*	100%	100%

In experiment 3, we discover all the constraints derived from the not co-existence template with a PoI of 100%. The results for experiment 3 are shown in Fig. 4.

For the LTL semantics of the not co-existence template

$$\varphi = \neg(\Diamond A \wedge \Diamond B),$$
$$witness(\varphi) = \varphi \wedge \neg\varphi$$

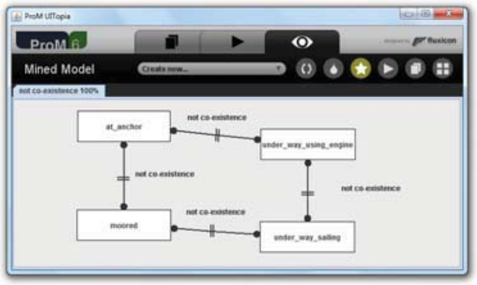


Fig. 4. Experiment 3: discovered not co-existence constraints

which is never satisfied. Therefore, not co-existence constraints are never non-vacuously satisfied.

The results of experiment 3 suggest the existence of two sub-types for vessel type *cargo/hazard pollutant C*: one composed of sailing boats which are never moored and another one composed of motorboats which never stop at anchor. This is confirmed by inspecting the log.

2) Pilot Boats: TABLE X shows the settings for an experiment aimed at discovering alternate response constraints using the sub-log associated to vessel type *pilot boat*. The results for experiment 4 are shown in Fig. 5

TABLE X
EXPERIMENT 4: EXPERIMENTAL SETTINGS

experiment number	*template*	*PoI*	*PoE*
4	*alternate response*	100	100%

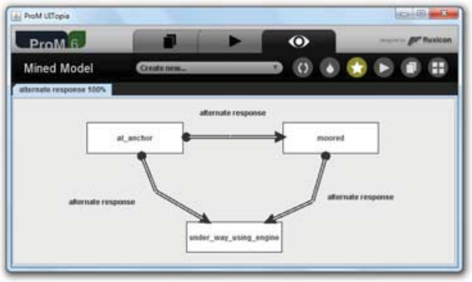


Fig. 5. Experiment 4: discovered alternate response constraints

For the LTL semantics of the alternate response template

$$\varphi = \Box(A \Rightarrow \bigcirc(\neg A\ U B)),$$

$$witness(\varphi) = \varphi \wedge \neg(\Box(A \Rightarrow \bigcirc B)).$$

The term $\neg(\Box(A \Rightarrow \bigcirc B))$ indicates that, to be non-vacuously satisfied, an alternate response constraint cannot be reduced to a chain response. This term is equivalent to $\Diamond(A \wedge \neg(\bigcirc B))$, and hence it can also be interpreted as that at least once an event A is not directly followed by an event B.

As shown in TABLE XI, 13% of the instances of the sub-log associated to vessel type *pilot boat* correspond to interesting witnesses for *alternate response*(*at anchor, moored*). Constraints *alternate response*(*at anchor, under way using engine*) and *alternate response*(*moored, under way using engine*) are trivially satisfied because events *at anchor* and *moored* are always (directly) followed by *under way using engine*.

TABLE XI
EXPERIMENT 4: INTERESTING WITNESSES

discovered constraint	*interesting witnesses*	*PoI*
alternate response(*at anchor, moored*)	13%	100%

The constraint *alternate response*(*at anchor, moored*) indicates that when a pilot boat stops at anchor it does not stop at anchor anymore before getting moored.

3) Cargo/Hazard Pollutant A: TABLE XII shows the settings for an experiment to discover not chain succession constraints for vessel type *cargo/hazard pollutant A*.

TABLE XII
EXPERIMENT 5: EXPERIMENTAL SETTINGS

experiment number	*template*	*PoI*	*PoE*	*NoC*
5	*not chain succession*	100%	100%	–

For the LTL semantics of the not chain succession template

$$\varphi = \Box(A \Rightarrow \bigcirc(\neg B)),$$

$$witness(\varphi) = \varphi \wedge \Diamond A.$$

The results for this experiment are shown in Fig. 6. These results show that when this type of ship is *not under command* the next navigational status cannot be *moored* or *at anchor*.

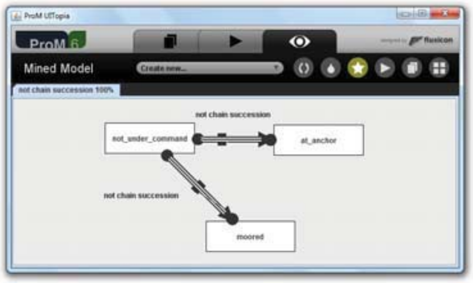


Fig. 6. Experiment 5: discovered not chain succession constraints

TABLE XIII shows the percentage of interesting witnesses for the discovered constraints. Since event *not under command* rarely happens in the sub-log, this percentage is extremely low.

TABLE XIII
EXPERIMENT 5: INTERESTING WITNESSES

discovered constraint	*interesting witnesses*	*PoI*
not chain succession(*not under command, moored*)	1%	100%
not chain succession(*not under command, at anchor*)	1%	100%

D. Discovery of Branched Constraints

1) Tanker: The sub-log associated to vessel type *tanker/unknown cargo type D* is composed of an alternation of event *under way using engine* followed by *moored* or *at anchor*. Therefore, this vessel type represents a good example where we can discover the chain response template where the second parameter is branched. In the LTL semantics of a branched template, the branched parameters are replaced by the disjunction of two or more parameters. In this case, the semantics of the chain response template becomes:

$$\varphi = \Box(A \Rightarrow \bigcirc(B \vee C)).$$

For this semantics we have

$$witness(\varphi) = \varphi \wedge \neg(\Box(A \Rightarrow \bigcirc B)) \wedge \neg(\Box(A \Rightarrow \bigcirc C)).$$

This means that this type of constraint is non-vacuously satisfied in all the process instances where at least once an event A is not directly followed by a B, and at least once an event A is not directly followed by a C.

One of the discovered constraints is a chain response constraint where the first parameter is *under way using engine* and the second one is the disjunction of *moored* and *at anchor*. A further experiment (experiment 6) has shown that this constraint can be discovered for different types of tankers. However, in some cases we need to remove noisy process instances by tuning the parameter PoI. In TABLE XIV, for each type of tanker, we list the PoI parameter, indicating the percentage of instances where the discovered constraint is (vacuously or non-vacuously) satisfied, and the percentage of interesting witnesses.

TABLE XIV
EXPERIMENT 6: RESULTS

vessel type	***PoI***	***interesting witnesses***
tanker/unknown cargo type A	94%	9%
tanker/hazard pollutant A	91%	11%
tanker/hazard pollutant B	91%	6%
tanker/hazard pollutant C	100%	13%
tanker/hazard pollutant D	100%	8%
tanker/unknown cargo type B	100%	13%
tanker/unknown cargo type C	100%	13%
tanker/unknown cargo type D	100%	13%
tanker/unknown cargo type E	100%	33%
tanker/unknown cargo type F	91%	12%

VI. CONCLUSION

In this paper, we have introduced a novel approach to discover declarative models from logs that allows users to guide the discovery process towards specific properties.

Moreover, we have shown how results on truncated semantics can be used to obtain significant results in the case that only partial logs are available. We have also applied results on vacuity detection to identify, for each discovered constraint, the percentage of interesting witnesses, i.e., process instances where the constraint is non-trivially valid.

In the near future, we want to extend our approach by providing users with the possibility to discover strongly, neutrally or weakly satisfied constraints depending on the level of reliability or flexibility they need in the discovery process. Given a constraint and a process instance where it is non-vacuously satisfied, it could also be useful to provide further information about how many times the constraint has been "activated" in the process instance (counting the number of violations for $\neg witness(\varphi)$). Also, given a constraint and a process instance where it is violated, the level of "healthiness" of the process instance can be evaluated based on the number of violations.

ACKNOWLEDGMENT

This research has been carried out as a part of the Poseidon project at Thales under the responsibilities of the Embedded Systems Institute (ESI). This project is partially supported by the Dutch Ministry of Economic Affairs under the BSIK program.

REFERENCES

[1] B. Van Dongen, A. K. de Medeiros, and L. Wen, "Process Mining: Overview and Outlook of Petri Net Discovery Algorithms," *T. Petri Nets and Other Models of Concurrency*, vol. 2, pp. 225–242, 2009.

[2] W. Van der Aalst, T. Weijters, and L. Maruster, "Workflow Mining: Discovering Process Models from Event Logs," *Knowledge and Data Engineering, IEEE Transactions on*, vol. 16, pp. 1128–1142, 2004.

[3] W. Van der Aalst, A. K. de Medeiros, and T. Weijters, "Genetic Process Mining," *Applications and Theory of Petri Nets*, LNCS, vol. 3536 pp. 48–69, 2005.

[4] C. Günther and W. Van der Aalst, "Fuzzy Mining - Adaptive Process Simplification Based on Multi-Perspective Metrics," in *BPM*, LNCS, vol. 4714, pp. 328–343, 2007.

[5] R. Bergenthum, J. Desel, R. Lorenz, and S. Mauser, "Process Mining Based on Regions of Languages," in *BPM*, LNCS, vol. 4714, pp. 375–383, 2007.

[6] A. Rozinat and W. Van der Aalst, "Conformance Checking of Processes Based on Monitoring Real Behavior," *Inf. Syst.*, vol. 33, pp. 64–95, 2008.

[7] J. C. Bose and W. M. P. Van der Aalst, "Abstractions in Process Mining: A Taxonomy of Patterns," in *BPM*, LNCS, vol. 5701, pp. 159–175, 2009.

[8] M. Pesic and W. Van der Aalst, "A Declarative Approach for Flexible Business Processes Management," *BPM Workshops*, vol. 4103, pp. 169–180, 2006.

[9] M. Pesic, H. Schonenberg, and W. Van der Aalst, "DECLARE: Full Support for Loosely-Structured Processes," in *EDOC*, pp. 287–300, IEEE, 2007.

[10] M. Pesic, "Constraint-Based Workflow Management Systems: Shifting Controls to users," Ph.D. dissertation, Beta Research School for Operations Management and Logistics, Eindhoven, 2008.

[11] F. Chesani, E. Lamma, P. Mello, M. Montali, F. Riguzzi, and S. Storari, "Exploiting Inductive Logic Programming Techniques for Declarative Process Mining," *ToPNoC*, vol. 5460, pp. 278–295, 2009.

[12] E. Bellodi, F. Riguzzi, and E. Lamma, "Probabilistic Declarative Process Mining," in *Knowledge Science, Engineering and Management*, LNCS, vol. 6291, pp. 292–303, 2010.

[13] R. Agrawal and R. Srikant, "Fast Algorithms for Mining Association Rules," in *VLDB*, vol. 1215, pp. 487–499, 1994.

[14] H. Mannila, H. Toivonen, and A. Inkeri Verkamo, "Discovery of Frequent Episodes in Event Sequences," *Data Mining and Knowledge Discovery*, vol. 1, no. 3, pp. 259–289, 1997.

[15] E. Lamma, P. Mello, F. Riguzzi, and S. Storari, "Applying Inductive Logic Programming to Process Mining," in *Inductive Logic Programming*, LNCS, vol. 4894, pp. 132–146, 2008.

[16] M. Leucker and C. Schallhart, "A Brief Account of Runtime Verification," *Journal of Logic and Algebraic Programming*, vol. 78, pp. 293–303, 2009.

[17] W. Van der Aalst, H. de Beer, and B. Van Dongen, "Process Mining and Verification of Properties: An Approach Based on Temporal Logic," in *On the Move to Meaningful Internet Systems 2005: CoopIS, DOA, and ODBASE*, vol. 3760, pp. 130–147, 2005.

[18] "Technical characteristics for a universal shipborne Automatic Identification System using time division multiple access in the VHF maritime mobile band," Recommendation ITU-R M.1371-1, 2001.

[19] C. Eisner, D. Fisman, J. Havlicek, A. Mcisaac, Y. Lustig, and D. V. Campenhout, "Reasoning with Temporal Logic on Truncated Paths," in *In CAV Proceedings, LNCS 2725*, pp. 27–40, 2003.

[20] J. Simmonds, J. Davies, A. Gurfinkel, and M. Chechik, "Exploiting Resolution Proofs to Speed up LTL Vacuity Detection for BMC," *International Journal STTT*, 2010.

[21] I. Beer and C. Eisner, "Efficient Detection of Vacuity in Temporal Model Checking," in *Formal Methods in System Design*, pp. 200–1, 2001.

[22] O. Kupferman and M. Y. Vardi, "Vacuity Detection in Temporal Model Checking," *International Journal STTT*, vol. 4, pp. 224–233, 2003.