Foundations and practice of binary process discovery

Slaats, Tijs; Debois, Søren; Back, Christoffer Olling; Christfort, Axel Kjeld Fjelrad

Published in:
Information Systems

DOI:
10.1016/j.is.2023.102339

Publication date:
2024

Document version
Publisher's PDF, also known as Version of record

Document license:
CC BY

Citation for published version (APA):
Foundations and practice of binary process discovery

Tijs Slaats a,∗, Søren Debois b,c, Christoffer Olling Back a,d, Axel Kjeld Fjelrad Christfort a

a Department of Computer Science, University of Copenhagen, 2100 København, Denmark
b DTU University of Copenhagen, Denmark
c DCR Solutions A/S, Denmark
d ServiceNow, Denmark

A R T I C L E   I N F O

Keywords:
Process mining
Binary classification
Negative examples
Labelled event logs
DisCoVer
Dynamic condition response graphs

A B S T R A C T

Most contemporary process discovery methods take as inputs only positive examples of process executions, and so they are one-class classification algorithms. However, we have found negative examples to also be available in industry, hence we build on earlier work that treats process discovery as a binary classification problem. This approach opens the door to many well-established methods and metrics from machine learning, in particular to improve the distinction between what should and should not be allowed by the output model. Concretely, we (1) present a verified formalisation of process discovery as a binary classification problem; (2) provide cases with negative examples from industry, including real-life logs; (3) propose the Rejection Miner binary classification procedure, applicable to any process notation that has a suitable syntactic composition operator; (4) implement two concrete binary miners, one outputting Declare patterns, the other Dynamic Condition Response (DCR) graphs; and (5) apply these miners to real world and synthetic logs obtained from our industry partners and the process discovery contest, showing increased output model quality in terms of accuracy and model size.

1. Introduction

From the perspective of machine learning, process discovery [1] sits uneasily in the gap between unary and binary classification problems [2,3]. Popular contemporary miners, e.g. [4–6], approach process discovery as unary classification: given only positive examples (the input log) they generate a classifier (the output model) which recognises traces (adhering to the output model) that resemble the training data. However, a process model is really a binary classifier: it classifies traces into those it accepts (desired executions of the process) and those it does not (undesired executions of the process).

Binary classification in machine learning relies on having access to examples of both classes. For process discovery, this means having not only positive examples of desired behaviour to be accepted by the output model, but also negative examples of undesired behaviour that should be rejected by the output model.

Negative examples also underpin a substantial part of the mechanics and theory of machine learning, in particular on model evaluation. Output models are evaluated on measures comparing ratios of true and false positives and negatives; however, without negative examples, it is impossible to apply such measures. Accordingly, in process discovery, we use measures based only on true positive answers, such as recall; we are deprived of more fine-grained measures involving true negative or false positive answers such as accuracy.

In practical process discovery, negative examples can help distinguish between incidental correlation and actual rules. For instance, suppose that in some log, whenever we see an activity B, that B is preceded by an activity A. Does that mean that we can infer the declarative rule A → B, that A is required before B may happen? In general, no: making this distinction requires domain knowledge. E.g., if A is “call taxi” and B is “file minutes from weekly status meeting”; by coincidence, we always call a taxi in the morning the day we file minutes, but clearly there is no rule that we must call a taxi before filing minutes. Conversely, if A is “approve payment” and B is “execute payment”, very likely it is a rule that B must be preceded by A.

A mining algorithm does not possess domain knowledge, and so must have help to make such distinctions, to decide whether to add a rule A → B to its output model. Negative examples potentially help here: If BA is in the set of negative examples, adding the rule A → B is justified, as it rejects this trace (we assume that the rule rejects no positive example). Conversely, if a rule rejects no trace from the negative examples, it is not necessary but discretionary for the miner to leave out or keep in as it pleases. In the case of our examples, we would expect to find ample evidence in our negative examples that
executing a payment before approving it is bad, whereas we would expect to find little to no evidence that filing minutes before calling a taxi is undesired.

Possibly one reason for the focus on unary classification in process discovery is that positive examples are oftentimes easier to come by than negative ones, e.g., system logs are readily available positive examples.

As shown by Ponce de Leon et al. [7] negative examples do exist in practice, some mining algorithms that include negative examples have been proposed, e.g., [7,8], and interestingly recent editions of the process discovery contest1 have moved towards using labelled test logs (but not training logs) to rank submissions. We add to this state of the art in binary process discovery with the following contributions:

1. We formalise process discovery as a binary classification problem, and show that not all process notations can express complete solutions to this problem (Section 4).
2. We propose the Rejection Miner, a notation-agnostic binary mining procedure applicable to any process notation with a syntactic composition operator (Section 5).
3. We have verified the formalisation of process discovery (contribution (1) and (2)) using the Isabelle proof assistant. All formal statements, algorithms and examples in Sections 4–5 have been so verified. The formal proof document is available online [9]. Formalised statements and proofs are marked by a black box at their end, like this: ■
4. We discuss how a Rejection Miner can be developed for process notations that do not offer a strict syntactic composition operator (we refer to these as non-additive process notations) and implement a prototype for Dynamic Condition Response (DCR) Graphs (Section 6).
5. We describe two cases where negative examples were encountered in industry and provide data sets [10] (Section 7).
6. We implement a concrete Rejection Miner and apply it to these data sets, comparing exploratively to contemporary unary miners (Section 8). The miner has been integrated in the commercial dcrgraphs.net modelling tool [11].
7. In addition we evaluate the miners on the classification logs from the 2021 Process Discovery Contest (Sections 7.3 and 8).

We note that in particular contributions (3), (4), and (7) add to our earlier published work in [12].

For the experiments, do note that the contemporary unary miners with which we compare do not take into account the negative examples. They must guess from the positive examples which traces to reject, whereas the rejection miners have the negative examples to further inform them on which traces should be rejected for certain. We find that both our binary miners achieve noticeably better accuracy, in particular on our largest real-world log, and, when using the same process notation, produce models that are orders-of-magnitude smaller than the unary miners.

Finally, we note that the experimental results, as reported in Section 8, have been significantly updated since the previous version of the paper [12]. First of all, we use newer versions of Minerful and the inductive miner, leading to some changes in the outcomes. In addition, the replication of the results through novel implementations of the rejection mining mechanism lead to the discovery of two bugs. The first of these was a bug in the implementation of the rejection miner prototype which caused some incorrect constraints to be found for logs of over 10 activities. This particularly influenced the results for the rejection miner on the Dreyers Foundation data set. With the corrected implementation, roughly twice the amount of constraints is found, with a slight increase in accuracy. The second was an error in the calculation of the true negative rate and balanced accuracy for all other miners in the out-of-sample experiment on the Dreyers Foundation data set. The new results are more in line with those for the in-sample experiment and in favour of the Rejection Miner.

2. Related work

There have been several earlier works framing process mining as a binary classification task. Lamma et al. [8] formulate constraints as Horn clauses and uses the ICL learning algorithm to successively find constraints which remove negative examples, stopping when there are no negative examples left. They translate these generated clauses to DECLARE. The Rejection Miner generalises this approach in that (a) it replaces the horn clauses with a generic notion of ”model” for notations with composition (or synchronous product of models), and thus applies directly to a plethora of languages such as DECLARE and DCR Graphs, (b) the Rejection Miner leaves the choice of which clauses to prune until a set of constraints ruling out all negative constraints is found, opening the door to non-greedy minimisation, and most importantly (c) we prove correctness for the Rejection Miner. Ponce de León et al. [7] propose an approach where traces are embedded in a n-dimensional space (n being the number of unique event classes of the log) using a Parikh representation. Finding a model is then reduced to the problem of finding a convex hull for the points such that positive points are included and negative points excluded. Goedertier et al. [13] and van den Broucke [14] artificially generate negative labels, but at the level of individual events rather than traces. The authors also defined process mining oriented metrics based on the resulting true positive/negative labels at the level of events. Slaats [15] identified the development of binary process discovery algorithms as a key open challenge for the field of declarative process discovery. Our work is also closely related to the work on vacuity detection in declarative process mining [16,17] which considers techniques for selecting the most relevant discovered constraints. However, they only consider logs with positive examples. The use of labelled input data is also well-accepted in the field of predictive process monitoring [18, 19]. Finally, our test-driven modelling use case presented in Section 7.1 is similar to the scenario-based modelling approach introduced by Fahland [20], where (potentially negative) scenarios are modelled as small Petri nets which can then be synthesised into a single larger model. Contrary to this approach we input positive and negative scenarios as traces and learn a declarative model from these.

Gold showed in [21] that even regular languages cannot be learned perfectly from positive examples alone, though parametrised subsets have been shown to be learnable [22], as have some monotonic formal languages of bounded length such as context-sensitive grammars and minimal models of linear Prologue programmes [23]. Relaxing the requirement of exactness, [24] shows that learning on positive examples alone is possible to within an arbitrarily small error using a Bayesian approach, while [25] demonstrates PAC(Probably Approximately Correct)-learnability when, in addition to positive examples, an oracle is provided that the learner can query with an instance and receive a positive or negative label.

Vapnik–Chervonenkis dimension analysis - a core concept in the theory of binary classification — in the context of grammatical inference has been addressed regarding finite sets of strings and regular automata in [26,27], and specifically models of words (traces) of equal length in [28].

3. Process notations and unary discovery

We recall the traditional definitions of event logs etc. [1].

---

1 https://icpmconference.org/2019/process-discovery-contest/
Definition 3.1 (Events, traces, logs). Assume a countably infinite universe $\mathcal{A}$ of all possible activities. As usual, an alphabet $\Sigma \subseteq \mathcal{A}$ is a set of activities, and the Kleene-star $\Sigma^*$ denotes the countably infinite set of finite strings or sequences over $\Sigma$; we call such a string a trace. A log $L$ is a multiset of occurrences of traces $L = \{t_1^m, \ldots, t_n^m\}$ where $m, n > 0$ is the multiplicity of the trace $t_i \in \Sigma^*$. We write $\mathcal{L}_\Sigma$ for the set of all event logs over alphabet $\Sigma$.

When convenient, in particular when stating and reasoning about correctness of miners, we treat an event log $L$ also as simply a set of traces by ignoring multiplicities. In the remainder of the paper, we also refer to the traces of a log as examples, as they are the examples provided to the miner of either good or bad behaviour.

When we discuss unary and binary process discovery in the abstract in later sections, we will be interested in applying discovery to a variety of process notations; and we shall propose a miner which can be instantiated to any notation with a suitable composition operator. To make such statements formally, we need a formal notion of process notation. We use $\mathcal{P}(S)$ for the power set of $S$.

Definition 3.2 (Process Notation). A process notation for an alphabet $\Sigma$ comprises a set of models $\mathcal{M}$ and an interpretation function $[\cdot] : \mathcal{M} \rightarrow \mathcal{P}(\Sigma^*)$ assigning to each individual model $m$ the set of traces $[m]$ accepted by that model. For a set $S \subseteq \Sigma^*$, we write $m \vDash S$ iff $S \subseteq [m]$.

This notion of “Process notation” is the straightforward expression of process mining in terms of functions on sets and multisets, and it is in line with similar foundational works, e.g., [29].

While a process notation comprises the three components $\Sigma$, $\mathcal{M}$, and $[\cdot]$, when no confusion is possible we shall allow ourselves to say “consider a process notation $\mathcal{M}$”, understanding the remaining two components to be implicit.

Note that in this view, a process model is a classifier: Given a trace, it classifies that trace as admissible or not: it accepts or rejects the trace.

Example 3.1. Here is a toy declarative formalism which allows exactly the condition constraint of DECLARE [30,31] or DCR [32,33] over a countably infinite alphabet $\Sigma = \{A, B, C, \ldots\}$. A “model” is any finite set of pairs $(x, y) \in \Sigma \times \Sigma$, and we interpret each such pair as a condition from $x$ to $y$. Formally:

$$\rightarrow = \{M \subseteq \Sigma \times \Sigma \mid M \text{ finite}\}$$

$$[M] = \{t \in \Sigma^* \mid \forall(x, y) \in M. \, \text{each } y \text{ in } t \text{ is preceded by } x\}$$

For instance, $[(A, B)]$ is a model consisting of a single condition from $A$ to $B$. In DECLARE or DCR, we would write this model “$A \rightarrow B$”.

Just as in DECLARE or DCR, this model admits all traces in which any occurrence of $B$ is preceded by an occurrence of $A$. That is, this model admits the trace $AB$, but not $B$ or $BABA$. Formally, we write

$$AB \in [(A, B)] \quad \text{or} \quad [(A, B)] \vDash \{AB\}$$

$$\{AB, B\} \notin [(A, B)] \quad \text{or} \quad [(A, B)] \vDash \{AB, B\}$$

$$\{B, BABA\} \notin [(A, B)] \quad \text{or} \quad [(A, B)] \vDash \{B, BABA\}$$

Any process modelling formalism with trace semantics is a process notation in the above sense; such formalism include DECLARE, DCR, and Workflow Nets [34] (see also [1]).

In many contemporary modelling formalisms, a concrete model $m$ references only a finite alphabet $\Sigma$, and the interpretation $[m] \subseteq \Sigma^*$ is confined to strings over this alphabet. Not all, however. E.g., in the “open world” interpretation of DECLARE, activities not specified explicitly in a model are still allowed in traces accepted by that model.

We conclude this Section by pinning down process discovery: a procedure which given an event log produces a process model which admits that log. Assume a fixed alphabet $\Sigma$, and write $\mathcal{L}_\Sigma$ for the set of all valid event logs over $\Sigma$.

Definition 3.3 (Unary Process Discovery). A unary process discovery algorithm $\gamma$ for a process notation $(\mathcal{M}, [\cdot])$ over $\Sigma$ is a function $\gamma : \mathcal{L}_\Sigma \rightarrow \mathcal{M}$. We say that $\gamma$ has perfect fitness iff for all $L \in \mathcal{L}_\Sigma$

$$\gamma(L) \vDash L \quad (1)$$

Anticipating our binary miners, we shall refer to “perfect fitness” also as positive soundness of the miner.

Note that a unary discovery algorithm gives rise to a function from logs to languages by composing with interpretation:

$$[\gamma(\cdot)] : \mathcal{L}_\Sigma \rightarrow \mathcal{P}(\Sigma^*)$$

That is, given a log, if we apply to it first a unary discovery mechanism and then the interpretation function, we get altogether a function from logs to languages: Given a log, this function yields the set of strings accepted by the model discovered for that log.

For process formalisms that have universal models – models admitting any trace – a unary miner always exists. This is the case for the major contemporary modelling notations, e.g., DECLARE, DCR, and Workflow Nets: these all have, for any finite alphabet, a so-called “flower-model”, whose interpretation is the entire space of possible traces.

4. Process discovery as binary classification

We proceed to consider process discovery a binary classification problem. This approach assumes that we have not only positive examples (the set $L$ in Definition 3.3), which the output model must accept, but also a set of negative examples, which the output model must reject.

Example 4.1. Consider again the condition models $\rightarrow$ of Example 3.1. Take as positive set of examples the singleton set $\{AB\}$, and take as negative examples the set $\{BA, B\}$. One model which accepts the positive example and rejects the negative ones is the singleton condition $[(A, B)]$. This model admits the positive example $AB$, because $B$ is preceded by $A$; and it rejects the negative examples, because in both of the traces $B$ and $BA$, the initial $B$ is not preceded by $A$.

The negative examples here help solve the relevancy problem that plagues unary miners for declarative formalisms: The positive example $AB$ clearly supports the constraint “$A$ is a condition for $B$”, however, as we saw in the introduction, with only positive examples and without domain knowledge, we cannot know whether this is a coincidence or a hard requirement. In the present example, the negative examples tell us that our model must somehow reject the trace $BA$, encouraging us to include the condition $A \rightarrow B$.

Unfortunately, a model accepting a given set $P$ of positive examples and rejecting a given set $N$ of negative ones does not necessarily exist: At the very least, we must have $P$ and $N$ disjoint (since otherwise we would have to find a model which both accepts and rejects the non-empty intersection of $P$ and $N$). To cater to such ambiguous inputs, we allow a binary miner to refuse to produce a model.

Definition 4.1 (Binary Process Discovery). Let $\mathcal{M}$ be a process notation for an alphabet $\Sigma$. A binary-classification process discovery algorithm (“binary miner”) is a partial function $\eta : \mathcal{L}_\Sigma \times \mathcal{L}_\Sigma \rightarrow \mathcal{M}$, taking sets of positive and negative examples $P, N$ to a model $\eta(P, N)$. We require that $\eta(P, N)$ is defined whenever $P, N$ are disjoint.

In the rest of this paper, unless otherwise stated, we shall implicitly assume that examples $P, N$ are disjoint. We proceed to generalise the notion of fitness from unary mining.

Definition 4.2 (Soundness, Perfection). Let $P, N \subseteq \mathcal{L}_\Sigma$ be positive and negative examples. We say that a binary miner $\eta$ is positively sound at $P, N$ if $\eta(P, N) \vDash P$. Similarly, we say that $\eta$ is negatively sound at $P, N$ if $\eta(N) \cap [\eta(P, N)] = \emptyset$. We say that $\eta$ is perfect iff for any disjoint $P, N$ it is defined and both positively and negatively sound.
In other words: A perfect binary miner produces an output whenever its positive and negative examples are not in direct conflict, and that output admits all positive examples and none of the negative examples provided as input.

**Over- and underfitting of out-of-sample data.** A perfect binary miner has no choice on how its output model treats the elements of \( P \) and \( N \): that model must admit its positive examples \( P \) and reject its negative examples \( N \). Where the perfect binary miner has a degree of freedom is in how its output model considers other trace, those trace neither in \( P \) nor \( N \). In the sequel, we call such traces "undecided". In the limits, we have the overfitting "maximally rejecting miner", whose output always accepts exactly \( P \) and nothing else; and the underfitting "maximally accepting miner", whose output rejects exactly \( N \) and nothing else. In the machine learning literature, controlling classification like this is known as **regularisation** [35].

However, unlike the unary case, where perfect fitness miners are generally quite easy to come by, perfect binary miners do not necessarily exist, and helpful ones may in practice be quite hard to come by.

First, let us try to use a unary miner as a binary one. We do so by simply ignoring the negative examples and applying our unary miner to the positive ones. In this case, it is easy to show that for any unary miner and any input log for that miner, either that miner returns exactly that log, or we can construct a negative example which will be accepted by the output model of that miner:

**Proposition 1.** Let \( \gamma \) be a unary miner for a notation \( M \) over alphabet \( \Sigma \). Then for all \( P \in L_\Sigma \) we have either \( \gamma(P) = P \), or there exists a \( N \in L_\Sigma \) s.t. \( N \) and \( P \) are disjoint, yet \( N \) is accepted by the output model \( \gamma(L) \).

**Proof.** By soundness of \( \gamma \) we have \( L \subseteq \gamma(L) \), if \( L = \gamma(L) \) we are done, so assume \( L \subseteq \gamma(\text{input}(L)) \). But then some \( t \in \gamma(L) \setminus L \) exists. Clearly \( t \) is disjoint from \( L \) and \( t \in \gamma(L) \); take \( N = \{t\} \).

So in this sense, non-trivial unary miners never generalise to binary ones. This is perhaps not entirely surprising. Much less obvious, and a core difference between binary and unary mining, we find that some notations cannot express distinctions fine enough to distinguish between positive and negative examples. This is in stark contrast to the unary case, where essentially all notations have a model accepting all traces (the "flower model"); moreover, all commonly accepted notations are able to express any finite language, and so for any input log (finite language), a perfectly fitting model must exist.

However, in the binary case, even though our example notation admits the "flower model", it is still too coarse to admit a perfect binary miner. We can see this by considering again the "condition-only" notation of Example 3.1.

**Lemma 1.** In the notation of Example 3.1, take positive examples \( P = \{ABC\} \), and negative examples \( N = \{AB\} \). Then no model \( m \) exists such that \( m \models P \) yet \( m \not\models N \).

**Proof.** Suppose \( m \) is a model with \( m \not\models \{AB\} \). By definition, \( m \) must contain a relation \( (x, y) \) prohibiting \( AB \). This relation must be on one of the forms \( (x, B) \) or \( (x, A) \), that is, some \( x \) is required to happen before \( A \) or required to happen before \( B \), but does not. In either case, clearly also \( m \not\models \{ABC\} \).

In fact, we prove now that no perfect binary miner can exist in any notation that has only finitely many possible models.

**Theorem 2.** No perfect binary miner exists for any process notation that has only finitely many possible models \( M \) over any non-empty finite alphabet \( \Sigma \).

**Proof.** We construct finite positive and negative examples \( P \) and \( N \) such that no model accepts \( P \) and rejects \( N \). First, we construct \( N \). Let \( I^+ \) as the subset of models that accepts infinitely many traces, i.e., \( I^+ = \{m \in M | \forall n \text{ infinite}\} \). Since there are only finitely many models, \( I^+ \) is finite, and without loss of generality write it \( I^+ = \{m_1, \ldots, m_k\} \). For each \( m_i \), choose a \( t_i \in \{m_i\} \) and define \( N = \{t_1, \ldots, t_k\} \). Next, we construct \( P \). Let \( \mathcal{C}(m) \) be the complement of the traces generated by a model \( m \) and \( I^- \) the subset of models which reject infinitely many traces, i.e., \( I^- = \{m \in M | \mathcal{C}(m) \text{ infinite}\} \). Again \( I^- \) is finite and we write it without loss of generality \( I^- = \{p_1, \ldots, p_k\} \). For each of \( p_j \), pick a trace \( s_j \) such that \( s_j \not\in \{p_j\} \) and \( s_j \not\in N \)-this is always possible because \( \mathcal{C}(m) \) is infinite and \( N \) finite. Then define \( P = \{s_1, \ldots, s_k\} \). Note that by construction \( P \) and \( N \) are disjoint. Finally, let \( m \in M \) be a model. At least one of \( \mathcal{C}(m) \) and \( \mathcal{C}(m) \) must be finite; we show that in neither case can \( m \) be the output of a perfect binary miner applied to \( P, N \). If \( \mathcal{C}(m) \) is infinite, then \( m \in I^+ \), say \( m = m_i \), and it follows that \( m \models t_i \in N \); hence \( m \) fails to reject all negative examples. If on the other hand \( \mathcal{C}(m) \) is infinite, then \( m \in I^- \), say \( m = p_j \) and it follows that \( s_j \not\in \{p_j\} \) and hence \( m \) cannot accept the positive example \( s_j \in P \).

To understand the ramifications of this Theorem, consider again DCR and DECLARE. For DCR or DECLARE models over a fixed finite alphabet (e.g., the set of tasks present in a given log), DCR has infinitely many such models (with distinct semantics), whereas DECLARE has only finitely many. To see this, note that in DCR, because labels and events are not one-one, we can keep adding events that do affect behaviour, while remaining within a finite set of observable tasks.2 In DECLARE, if there are \( n \) activities to choose from, you can populate only finitely many DECLARE templates with those finitely many tasks. Since the arity of DECLARE templates is bounded, and current DECLARE miners are bounded to a finite set of input templates, you are left with only finitely many models.

Note the following consequence for DECLARE: any binary miner for DECLARE has inputs \( P, N \) for which the output a model has either false positives or false negatives.

We note that the above proof could possibly be rephrased along the lines that there are infinitely many problems \( P, N \) with pairwise distinct solutions; the Theorem then follows from the Vapnik–Chervonenkis dimension [35] of the set of interpretations of the finite set of models being necessarily finite, and so unable to shatter this infinite set of distinct solutions.

In unary mining, we may construct a perfect fitness miner like this: As notation, pick simply finite sets of traces, and let the semantics of the notation be that a model (set of traces) \( T \) accepts a trace \( t \) iff \( t \in T \). Then the function \( \gamma(P) = P \) is a perfect fitness miner. This generalises to any notation strong enough to characterise exactly a given set of \( T \) of traces. Obviously, this unary miner has little practical relevance.

It is interesting to note that a similar perfect binary miner exists. Pick as our process notation pairs of sets of traces \( P, N \); that is, in this notation is a pair \( P, N \subseteq P(\Sigma) \times P(\Sigma) \). Take as semantics of this notation that a model \( (P, N) \subseteq P(\Sigma) \times P(\Sigma) \) accepts a trace \( t \) iff \( t \in P \) and \( t \not\in N \). Clearly the function \( \eta(P, N) = (P, N) \) is a perfect binary miner, although again, not a particularly helpful one. However, the construction shows that a perfect binary miner exists for any notation strong enough to exactly characterise membership resp. non-membership of finite sets of traces. Notable examples here are Petri-nets and BPMN (through an exclusive choice over the set of positive traces); so it follows that a (trivial) binary miner exists for these notations.

2 Note that labelled Petri nets and [20] similarly provides an infinity of possible models.
5. Rejection miners

We proceed to construct a family of binary miners we call “Rejection miners”, defined for any process notation which has a behaviour-preserving syntactic model composition. Rejection miners are parametric in a “pattern oracle” which selects a set of patterns for consideration; if the patterns selected allow it, the output of the Rejection Miner is perfect. When they do not, the miner does a greedy approximation to optimise for accuracy (i.e., maximising the ratio of true positives and negatives to all inputs).

5.1. Additive process notations

Declarative model notations typically comprise a collection of constraints, which can to some degree be considered in isolation. In such notations, we can think about a model as the sum of its parts, and the problem of mining can then be reduced to finding suitable pairs. We call such process notations “additive process notations”.

Definition 5.1 (Additive Process Notation). We say that a process formalism \( \mathcal{M} \) over \( \Sigma \) is additive if it comes equipped with a commutative monoid \( (\oplus, 1) \) on \( \mathcal{M} \) such that

\[
I = \Sigma^* \\
[m \oplus n] = [m] \cap [n]
\]

(2)

We lift the monoid operator to sequences and write \( \bigoplus_{i \in \mathbb{N}} m_i = m_1 \oplus \cdots \oplus m_{n-1} \).

That is, an additive formalism has a flower model \( I \) and a model combination operator \( \oplus \). This operator combines two models into a compound one, such that this compound model accepts exactly the traces accepted by both of the two original models.

Example 5.1. The condition-only notation of Example 3.1 is clearly additive: A model is literally a set of constraints, and so we may combine two models by taking their union. The required commutative union is then \( I \).

Example 5.2. DECLARE is an additive formalism: A DECLARE model is a finite set of constraints; the empty such set accepts all traces (1), and the union of two such sets is again such a set, with exactly the desired semantics (\( \oplus \)).

Example 5.3. DCR also has a model composition, where the composite model is the union of events, markings, and constraints [36,37]. However, this composition does not preserve semantics in the general case. To see this, consider two graphs, both over the alphabet \( \Sigma = \{A, B, C\} \), with constraints a single exclusion \( A \rightarrow\% B \) respectively a single condition, \( B \rightarrow C \). The latter graph will not accept the trace \( AC \), because that graph requires \( B \) to precede \( C \). But in the compound graph \( A \rightarrow\% B, B \rightarrow C \), once \( A \) is executed, \( B \) is excluded and \( C \) is released from its condition on \( B \); hence the compound graph \( \oplus \) accepts the trace \( AC \), violating Eq. (3). We will revisit this example, including a detailed description of the semantics of DCR Graphs in Section 6.

Note that in all three examples, the “models” of the process notation naturally falls in two classes: those which can be non-trivially decomposed into smaller models, and those which cannot be further decomposed. E.g., in the condition-only formalism, the model \( (A, B), (A, C) \) can be decomposed into

\[
(A, B, (A, C)) = [(A, B)] \cup [(A, C)] = [(A, B)] \oplus [(A, C)].
\]

3 An exclusion \( A \rightarrow\% B \) in DCR renders the target activity \( B \) and all of its constraints irrelevant once \( A \) executes; refer to the extended introduction to DCR in Section 6 for more details.

However, the individual parts cannot be further non-trivially decomposed: There are no \( X,Y \) both different from \( \emptyset \) s.t. \( \{(A, B)\} = X \oplus Y \).

This idea of decomposition seems to define what we think of as “a constraint” in a declarative notation: Constraints cannot be taken apart any further; they are the building blocks of other models. In the present formalisation, we shall not need to distinguish between constraints and compound models, so we leave the exploration of the connection between decomposition and constraints for future work.

In practice, any process notation can be considered additive by forming the synchronous product of models: To check whether a given trace \( t \) conforms to a composite model \( m \oplus n \), we simply check whether \( m \sqcup t \) and \( n \sqcup t \). Incidentally, this is a popular implementation mechanism for DECLARE constraints (see, e.g., [38,39]).

The key property of additive notations is that they are compositional: A miner can try to construct a model by iteratively adding to them, cutting down the space of accepted traces by each addition. The following Definition and Lemma encapsulates this intuition. For declarative modelling practitioners, it may be helpful for intuition to think “constraint” where the Definition says “model”.

Definition 5.2. Suppose \( \mathcal{M} \) is an additive formalism and \( L \in \Sigma \) a log, both over \( \Sigma \), and let \( \sigma = m_1, \ldots, m_{n-1} \in \mathcal{M}^* \) be a sequence of models. We say that \( \sigma \) fully rejects \( L \) iff for each \( i \in L \) for some \( j < n \) we have \( m_j \not\sqcup \{i\} \).

Assuming \( m \sqcup t \) for any single trace \( t \) is computable in time \( O(f(n)) \) where \( |t| = n \), then also \( "m \sqcup \text{fully rejects } L \)" for any finite log \( L \) is computable in time \( O(|L| \cdot f(n)) \) where \( |L| \) is the number of traces in the log (considered a set), and \( n \) is the length of the longest trace in the log.

If a set of models fully rejects a set \( L \), the composition of that set also rejects \( L \).

Lemma 2. Suppose \( \mathcal{M} \) is an additive formalism and \( L \in \Sigma \) a non-empty log, both over \( \Sigma \); and let \( \sigma \in \mathcal{M}^* \) be a non-empty sequence of models such that \( \sigma \) fully rejects \( L \). Then \( \bigoplus \sigma \not\sqcup L \).

However, the converse is not true: full rejection is stronger than the composition of models not modelling a language. For \( \bigoplus \sigma \not\sqcup L \) it is sufficient that some trace \( r \) in \( L \) is rejected by \( \bigoplus \sigma \). Full rejection on the other hand requires that all traces in \( L \) are rejected.

5.2. Rejection miners

With additive notations in place, we define Rejection Miners. A rejection miner is a binary miner parametric in three sub-components: A pattern oracle, which given positive and negative examples produces a finite set of (hopefully) relevant models (“patterns”); a model minimiser which given a sequence of models known to fully reject a set of negative examples selects a subset still fully rejecting those examples; and finally an approximation fallback to use in case the patterns suggested by the oracle are not powerful enough to fully reject the negative examples.

Definition 5.3 (Rejection Miner Components). Let \( \mathcal{M} \) be a process notation over an alphabet \( \Sigma \). A pattern oracle is a function patterns : \( \Sigma \times \Sigma \rightarrow \mathcal{M}^* \). A minimiser is a function minimise : \( \mathcal{M}^* \times \Sigma \rightarrow \mathcal{M} \) satisfying:

1. if \( \sigma \in \mathcal{M}^* \) fully rejects \( L \), then also minimise(\( \sigma, L \)) fully rejects \( L \); and
2. minimise(\( \sigma, L \)) contains only elements from the input sequence \( \sigma \).

Finally, an approximation is a partial function approximate : \( \mathcal{M}^* \times \Sigma \times \Sigma \rightarrow \mathcal{M} \), which is defined whenever its second and third inputs are disjoint, that is, \( \approximate(\sigma, P, N) \neq \perp \) iff \( P \cap N = \emptyset \).
An example pattern oracle for DECLARE would be the function that produces all possible instantiations of all templates with activities observed in either of its input logs, similar to the initial DECLARE miner proposed by Maggi et al. [40] An example minimiser is the greedy minimiser which, starting from the left of the list of models, removes those models which reject only traces in the negative examples \( N \) that are already rejected by preceding models. That is, model \( m_i \) is removed iff

\[
[m_i] \cap N \subseteq \left( \bigoplus_{j < i} m_j \right) \cap N
\]

With these components in place, and assuming that for a finite log \( L \) both the modelling relation \( m \not\in L \) and the addition operator \( \oplus \) are computable, we define the rejection mining algorithm in Algorithm 1 below.

Algorithm 1 The rejection miner

Let \( M \) be an additive notation over \( \Sigma \), let \( \text{patterns} \) be a pattern oracle and let \( \text{minimise} \) be a minimiser.

1: procedure \textsc{RejectionMiner}(\( P, N \))
2: \[ [m_1, \ldots, m_j] \leftarrow \text{patterns}(P, N) \] \( \triangleright \) remove \( m_j \) where \( m_j \not\in P \)
3: \( \sigma \leftarrow [m_1 | m_i \not\in P] \)
4: \( \sigma \leftarrow \text{minimise}(\sigma, N) \)
5: if \( \bigoplus \sigma \cap N = \emptyset \) then \( \triangleright \) are any negative examples not rejected?
6: \( \text{return} \bigoplus \sigma \)
7: else
8: \( \text{return} \) \text{approximate}(\sigma, P, N) \)
9: end if
10: end procedure

A brief explanation: On line 2, the pattern oracle is invoked to produce a finite list \([m_1, \ldots, m_j]\) of relevant models. On Line 3, those models not modelling the positive examples \( P \) are filtered out; only the models \( m_i \) which do model \( P \) are retained; we assign the resulting list to \( \sigma \). We then apply the minimiser in Line 4, which by Definition 5.3 at most removes models. On Line 5, we check whether all negative examples are rejected; if so, we have found a perfect model and return it on Line 6, otherwise we fallback to applying the approximation on Line 8.

Recall from the previous section the notions of maximally accepting or maximally rejecting perfect binary miners. The minimiser provides a handle for pushing the Rejection Miner towards either of these extremes. Using the identity function as the minimiser will retain all \( \sigma \) with \( \sigma(N) = \emptyset \) that eventually we should also see \( C \). The selection of constraints was a result of trial and error based on the dataset described in Section 7.1, our goal was to find a relatively simple set of constraints that allowed for the construction of a completely accurate model for each of the 215 logs in the dataset. A more complete approach would be to implement all standard DECLARE constraints.

The oracle outputs patterns sorted by how many negative examples they exclude. Ties are broken by sorting the disjunctive and conjunctive responses last, to de-emphasise these relatively more complex patterns. We then use the greedy minimiser sketched in Section 5.2 on this sorted list of patterns; in effect, the miner prefers fewer constraints without having to solve the NP-complete problem of finding a minimal subset.

We emphasise the flexibility of the oracle and minimiser selection: if one wants to include more patterns, one simply extends the oracle; if one wants to have a more restrictive model, one simply replaces the minimiser. The implementation also allows one to produce models that sacrifice true positives for a higher accuracy rate, by creating a minimiser that accepts constraints excluding some positive examples, but also excluding many negative examples.

6. Implementing a rejection miner for dynamic condition response graphs

In the previous sections, we have defined a general rejection miner for additive process notations. However, not all interesting declarative formalisms are additive. The most notable example is Dynamic Condition Response (DCR) graphs, which stand out in the declarative modelling sphere for being supported by commercial tools and have seen several successful industrial applications [42]. In this section we show how the rejection miner mechanism can be adapted to non-additive formalisms, with DCR Graphs as both our key motivation and prime example.

We begin by briefly recalling the DCR graphs formalism.

6.1. DCR graphs

First let us recall the definition and semantics of DCR Graphs. A DCR Graph consists of nodes called events and edges called relations. Normally DCR Graphs contain a labelling function that maps events to the activities that they represent and multiple events may map to the same activity. In this way the graph can capture that different rules apply to an activity depending on the context (for example if it is the first or second execution of that activity). However, our miner only mines models where each activity is mapped as exactly one DCR event.

\[ \text{http://www.dcrgraphs.net.} \]
Therefore we use a common simplification of DCR Graphs where we disregard the labelling function and assume it to be bijective [43]. Note that this does not prohibit activities from being repeated, as DCR events are repeatable unless their relations prohibit this. In addition, to avoid confusion with the notion of events in the process mining literature, we will refer to DCR events as activities.

Note that because of this limitation, our miner cannot be a perfect binary miner, as the amount of models it can generate is bounded by the number of unique activities in the log. Interestingly, a trivial perfect binary miner for labelled DCR Graphs does exist. Similar to labelled Petri nets and BPMN, one can model an exclusive choice over the set of positive traces.

Finally, as event logs contain by definition only finite traces, we will also only consider the semantics of DCR Graphs over finite traces. For a more comprehensive introduction, including a semantics over both finite and infinite traces, we refer to [33,44,45].

Definition 6.1. A DCR graph is a tuple \((E, M, \rightarrow, \leftarrow, \rightarrow+, \leftarrow+, \rightarrow\% )\), where

- \(E\) is the set of activities
- \(M = (\text{Ex}, \text{Re}, \text{In}) \in P(E) \times P(E) \times P(E)\) is the marking of the graph, consisting of the executed, pending, and included sets respectively.
- \(\phi \subseteq E \times E\) for \(\phi \in \{ \rightarrow, \leftarrow, \rightarrow+, \leftarrow+, \rightarrow\% \}\) are respectively the condition, response, inclusion and exclusion relations between activities.

Example 6.1. In Fig. 1 we show an example of a DCR Graph. It has 5 activities and demonstrates each relation. There are: (1) an exclusion relation from A to B, which means that A excludes B from the process, after which B cannot be executed and does not impose it is conditions on other activities, (2) a condition from B to C, which means that C cannot be executed unless B has been executed at least once before, (3) a response from D to B, which means that whenever D happens, B should eventually be executed afterwards, and (4) an inclusion from E to B, which means that if B was previously excluded from the process, E will add it back in. Currently, all activities are included in the graph, no activities have been previously executed, and no activities are pending. Activities can be pending initially, or become pending as the consequence of a response relation, meaning that they must be executed before the process can end, unless they are removed from the process.

Note that the relations between events A, B, and C are the same as we saw in Example 5.3

Notation. For a DCR graph \(G\) with activities \(E\) and an activity \(e \in E\), we write \((\rightarrow e)\) for the set \(\{ e' \in E \mid e' \rightarrow e\}\), write \((e\rightarrow+)\) for the set \(\{ e' \in E \mid e\rightarrow e'\}\) and similarly for \((e\rightarrow+)\) and \((e\rightarrow\% )\).

Definition 6.2 (Enabled Activities). Let \(G = (E, M, \rightarrow, \leftarrow, \rightarrow+, \leftarrow+, \rightarrow\% )\) be a DCR graph, with marking \(M = (\text{Ex}, \text{Re}, \text{In})\). An activity \(e \in E\) is enabled, written \(e \in \text{enabled}(G)\), iff \((a) e \in \text{In}\) and \((b) \text{In} \cap (\rightarrow e) \subseteq \text{Ex}\).

I.e. an activity of a DCR graph is enabled when \((a)\) it is included and \((b)\) all included conditions have been executed.

Example 6.2. In the example of Fig. 1 all activities are enabled except for activity C, which is blocked by the condition from activity B, which has not yet been executed.

If an activity is enabled then it can be executed. Executing an activity \(e\) updates the marking of the graph by \((a)\) adding it to the set of executed activities, \((b)\) removing it from the set of pending activities and adding its responses \((e\rightarrow+)\) to the set of pending activities, and \((c)\) respectively removing its exclusions \((e\rightarrow\% )\) from and adding its inclusions \((e\rightarrow+)\) to the set of included activities.

Definition 6.3 (Execution). Let \(G = (E, M, \rightarrow, \leftarrow, \rightarrow+, \leftarrow+, \rightarrow\% )\) be a DCR graph, with marking \(M = (\text{Ex}, \text{Re}, \text{In})\). When \(e \in \text{enabled}(G)\), the result of executing \(e\), written \(\text{execute}(G, e)\) is a new DCR graph \(G'\) with the same activities and relations, but a new marking \(M' = (\text{Ex}', \text{Re}', \text{In}')\), where \((a)\) \(\text{Ex}' = \text{Ex} \cup \{ e \}\) \((b)\) \(\text{Re}' = (\text{Re} \setminus \{ e \}) \cup (e\rightarrow')\), and \((c)\) \(\text{In}' = (\text{In} \setminus (e\rightarrow\% )) \cup (e\rightarrow+)\).

Definition 6.4 (Accepting). Let \(G\) be a DCR graph, with marking \(M = (\text{Ex}, \text{Re}, \text{In})\). We say that \(G\) is accepting, written \(\text{accepting}(G)\), iff \(\text{In} \cap \text{Re} = \{\}\).

Example 6.3. In the example of Fig. 1, consider the following trace: \((D, A, E, B, C)\). As a general note, the graph is initially accepting, and for each step in the trace, when an activity is executed (for the first time), it is added to the set of executed activities. Now let us look at each execution step in more detail: (1) When we execute the activity D, activity B will become pending and the graph is no longer accepting. (2) When we then execute activity A, it will exclude the activity B. Afterwards, activity B will no longer be enabled, but activity C will be, as the condition becomes irrelevant once B is excluded. The graph also becomes accepting again, as the only pending response has been excluded. (3) Executing activity E will include activity B once more, after which B will again be enabled, but not C. The graph is also no
longer accepting, as the pending response has been included back into the graph. (4) When we execute the activity B, it is removed from the pending set making the graph accepting again, and because B is added to the executed set, C now becomes enabled. (5) We can now execute the activity C.

Example 6.4. We see now why DCR Graphs are non-additive: between step (2) and (3) the activity C became enabled because of the interplay between the exclusion and condition relations, allowing for the trace (A, C). However, if we were to take these relations in isolation and considered two separate graphs, one A →% B and the other B → C, and define our language as the synonymous product of the languages of each individual graph, then the trace (A, C) should not be allowed, as it is not allowed in the second graph.

Finally we can define the execution sequences and accepting traces of a DCR graph.

Definition 6.5 (Execution Sequences and Accepting Traces). We define a labelled transition relation between DCR graphs \( G \rightarrow G' \) if \( e \in \text{enabled}(G) \) and \( G' = \text{execute}(G, e) \). A finite trace \( t = \langle e_1, \ldots, e_n \rangle \) is an execution sequence of a DCR graph \( G \) iff \( G \rightarrow^* e_1 \cdots \rightarrow^* G'' \). It is an accepting trace of \( G \) iff \( \text{accepting}(G'') \).

Example 6.5. The traces: \( \langle D_1 \rangle, \langle D.A \rangle, \langle D.A.E \rangle, \) and \( \langle D.A.E.B.C \rangle \) are all execution sequences of the example in Fig. 1. Of these, only \( \langle D.A \rangle \) and \( \langle D.A.E.B.C \rangle \) are accepting traces. The trace \( \langle C \rangle \) is neither an execution sequence or accepting trace.

6.2. Binary mining of DCR graphs

We proceed to consider binary mining for DCR graphs. As noted in the previous section, DCR Graphs are not an additive formalism. Several papers have been written on the topic of safe refinements for DCR Graphs, which show that under certain syntactic restrictions, patterns can be combined in an additive manner [33,43,46]. To avoid such syntactic restrictions (and therefore be able to mine more relations, leading to more accurate models), we suggest an algorithmic approach that allows us to apply the rejection miner algorithm to DCR Graphs (using single relations as patterns) while still guaranteeing that the set of positive traces is fully accepted by the resulting model.

We adopt the rejection miner algorithm as follows. The reason it does not apply directly is that even though the patterns (relations) \( \sigma \) selected in Line 3 of algorithm 1 in isolation admit the positive (line 2) and negative (line 3) traces, with respect to the established relations in \( G \). We proceed to formalise these concepts.

First, we say that a condition relation rejects a trace if in that trace, we attempt to execute the event corresponding to the target activity \( e \) while \( e \) is not included and the source activity \( e' \) was one of the activities to exclude it since it was last included.

Definition 6.7. Let \( G \) be a pre-existing DCR graph, \( \sigma \) a candidate DCR Graph, \( t \) a trace. Assume \( e \in t \) and we have already executed its prefix. An exclude-relation in \( \sigma (e' \rightarrow % e) \) rejects the trace \( t \) if \( e \) is not included in \( \sigma \), and \( e' \) has been executed since \( e \) was last included in \( G \).

Furthermore, after the full execution of each trace, we can determine that if there are pending responses \( e \) in the candidate graph, which are not also excluded in the pre-existing graph, then any response relation to \( e \) that fired since the last execution of \( e \) rejects that trace.

Definition 6.8. Let \( G \) be a pre-existing DCR graph, \( \sigma \) a candidate DCR Graph, \( t \) a trace, and \( G' \) and \( \sigma' \) be \( G \) and \( \sigma \) after executing \( t \) on each respectively. Assume \( e' \in t \). A response-relation in \( \sigma (e' \rightarrow e) \) rejects the trace if \( e \) is pending in \( \sigma' \) and included in \( G' \), and \( e' \) has been executed since \( e \) was last executed in \( t \).

Based on these definitions we define Algorithm 2 for computing the trace rejection for a set of relations. Here \( \text{exSiIn} \) is a mapping of an activity \( e \), to the set of activities executed since \( e \) was last included. Likewise \( \text{exSiEx} \) is a mapping of an activity \( e \), to the set of activities executed since \( e \) was last executed. Both of these can trivially be maintained at the end of each inner iteration. The algorithm produces \( r \), which is a mapping from a relation to the set of traces in \( L \), which the relation rejects. In particular in line 7, we add the current trace as a rejected trace to any condition relation that blocks the execution of activity \( e \) (in line with Definition 6.6). In line 9 we add the current trace as a rejected trace to any exclusion relation that excluded the activity \( e \) (in line with Definition 6.7). In line 12, for any remaining pending activity, we add the current trace as a rejected trace to any response relation that made that activity pending (in line with Definition 6.8).

Algorithm 2 TraceRejection(\( G, \sigma, L \))

1: \( r = \{ \} \)
2: for \( i \in L \) do
3: \( \text{exSiIn} = \{ \} \)
4: \( \text{exSiEx} = \{ \} \)
5: for \( e \in t \) do
6: \( G = \text{execute}(G, e); \sigma = \text{execute}(\sigma, e) \)
7: \( \langle \text{Ex}, \text{Re}, \text{In} \rangle = G[M] \)
8: \( \langle \text{Ex}', \text{Re}', \text{In}' \rangle = \sigma[M] \)
9: \( r[\langle e' \rightarrow e \rangle] = r[\langle e' \rightarrow e \rangle] \cup \{ t \} \quad \forall \langle e' \rightarrow e \rangle \in \sigma[\sigma[\langle e' \rightarrow e \rangle] \cap \text{In}] \)
10: if \( e \notin \text{In}' \) then
11: \( r[\langle e' \rightarrow % e \rangle] = r[\langle e' \rightarrow % e \rangle] \cup \{ t \} \quad \forall \langle e' \rightarrow % e \rangle \in \sigma[\sigma[\langle e' \rightarrow % e \rangle] \cap \text{exSiEx} \cap \text{In}] \)
12: end if
13: \( \text{exSiIn}, \text{exSiEx} = \text{updateMappings}(\sigma, G) \)
14: end for
15: \( r[\langle e' \rightarrow e \rangle] = r[\langle e' \rightarrow e \rangle] \cup \{ t \} \quad \forall \langle e' \rightarrow e \rangle \in \sigma[\sigma[\langle e' \rightarrow e \rangle] \cap \text{In}] \)
16: return \( r \)

Using Algorithm 2 we can now define a relation minimiser as seen in Algorithm 3. This greedy minimiser computes the trace rejection sets of the positive (line 2) and negative (line 3) traces, with respect to the relations in \( \sigma \), and adds the one that rejects the most negative traces and no positive traces (line 4). This is then repeated on the remaining traces until there are none, or no relation covers any of the remaining (line 7). Note that due to the fact that a DCR graph is not an additive formalism, the trace cover sets need to be recomputed for each relation added to the model G. Furthermore, for the same reasons, we cannot
assume that any relation in \( \sigma \) in isolation will still accept all positive traces, even if \( \sigma \) guarantee perfect fitness. It is therefore necessary to compute trace rejection for the positive traces as well, ensuring that only relations are picked that do not reject any positive traces.

Algorithm 3 Minimise(\( G, \sigma, P, N \))

1: repeat
2: \( \tau_P \leftarrow \text{TraceRejection}(G, \sigma, P) \)
3: \( \tau_N \leftarrow \text{TraceRejection}(G, \sigma, N) \)
4: \( \text{rel} \leftarrow \text{rel}' \mid \forall \text{rel}' \in \sigma.([\tau_N[\text{rel}']] \geq |\tau_N[\text{rel}']|) \land \tau_P[\text{rel}'] = \emptyset \)
5: \( G \leftarrow G \cup \text{rel} \)
6: \( N \leftarrow N \setminus \{\text{rel}\} \)
7: until \( N = \emptyset \lor |\tau[\text{rel}]| = 0 \) 

Using Algorithm 2 and 3 we can now define our implementation of a rejection miner as seen in Algorithm 4. We consider the pattern oracle producing the graph \( \sigma \) consisting of all conditions, responses and excludes. We then compute an adequate minimum of \( \sigma \), by greedily moving the relation covering the most negative traces into the initially empty model \( G \) until no more negative traces can be covered. If there are still uncovered negative traces, a greedy approach like in Algorithm 3 is now started, only now the difference in negative and positive traces covered is maximised. This continues until all negative traces are covered, or there is no more beneficial relations to add.

Algorithm 4 RejectionDCR(P,N)

Output: \( G \), the final model
1: \( \sigma \leftarrow \text{ProducePatterns}() \)
2: \( G \leftarrow \text{MakeFlowerModel}() \)
3: Minimise(\( G, \sigma, P, N \))
4: if \( N \neq \emptyset \) then
5: repeat
6: \( \tau_P \leftarrow \text{TraceRejection}(G, \sigma, P) \)
7: \( \tau_N \leftarrow \text{TraceRejection}(G, \sigma, N) \)
8: \( \text{rel} \leftarrow \text{rel}' \mid \forall \text{rel}' \in \sigma.([\tau_N[\text{rel}']] \geq |\tau_N[\text{rel}']|) \land \tau_P[\text{rel}'] = \emptyset \)
9: if \( |\tau_N[\text{rel}']| - |\tau_P[\text{rel}']| > 0 \) then
10: \( G \leftarrow G \cup \text{rel} \)
11: \( P \leftarrow P - \tau_P[\text{rel}] \)
12: \( N \leftarrow N \setminus \{\text{rel}\} \)
13: end if
14: until \( N = \emptyset \lor |\tau[\text{rel}]| \leq 0 \)
15: end if

One potential improvement to this algorithm, which we do not consider in the current paper, is the fact that based on earlier refinement results [43] we know that certain relations can be considered in parallel, as they are guaranteed to be independent and therefore not to affect each other. Applying these results would allow us to skip a significant number of trace rejection computations in the minimiser algorithm.

It should be noted that while this iterative process of computing \( \tau \) and selecting the relation \( \text{rel} \) rejecting most negative traces ensures that \( \text{rel} \) will actually cover the traces specified in \( \tau \) even when combining it with \( G \), there is no guarantee that adding \( \text{rel} \) will not cause a trace previously rejected by another relation in \( G \) to not be rejected anymore. Therefore the current algorithm is neither positively or negatively sound and is not guaranteed to provide the most accurate DCR Graphs for a given log. However, in our experiments this occurred relatively rarely, and the results in Section 8 show that the miner generally provides quite promising results.

We provide a JavaScript implementation of the Rejection DCR Miner, which is available at [47].

7. Cases with negative examples

The development of the Rejection Miner was not just motivated by academic, but also industrial interest. When pursuing process mining activities in practice we regularly see opportunities to label data and in some cases we have even been asked directly by commercial partners to include counter examples in the construction of models. In this section we discuss the two most developed cases we have encountered in industry where we both had the opportunity to extract labelled data and publish it in an anonymized format. The negative examples in these cases arise from test-driven development and as failures in process engineering. In addition we discuss the use of negative traces in the process discovery contest.

7.1. DCR solutions: Test-driven modelling

A Danish vendor of adaptive case-management systems, DCR Solutions, offers the on-line process modelling portal dcrgraphs.net. In this tool, modellers define required (positive) resp. forbidden (negative) test cases (traces), expected to be accepted resp. rejected by the model under development. The test cases are also used as input to the DiscoverR process discovery algorithm, which dynamically recommends new constraints to modellers [48]. However, the algorithm uses only the positive test-cases, ignoring the negative ones. The extension to consider also those negative ones has been repeatedly requested by the developers of the portal and was implemented as part of this paper. DCR Solutions has kindly allowed us to make the entire data set of test-cases produced in the portal available in an anonymized form [10].

7.2. Dreyers Foundation: Process engineering

The Danish Dreyers Foundation supports budding lawyers and architects, and has previously released an anonymised log of casework [49]. This log documents also testing and early stages of deployment of the system. In a number of cases, process instances that had gone astray were reset to their starting state and partially replayed. The log contains reset markers, and so provides clear negative examples: those prefixes that ended in a reset. We make available here also this partitioning into positive and negative examples [10]. Note that these reset markers are not comparable to the negative events of [13,14]: a reset marker indicates a point in the trace where it was observed that something went wrong, the actual error may have occurred much earlier in the process. In addition after the reset the entire prefix is discarded and the process starts fresh, therefore the points before and after the reset marker should be treated as separate traces.

7.3. Process discovery contest 2021 (PDC2021): Synthetic test logs

The latest editions of the process discovery contest compare miners based on a set of synthetic training logs and a corresponding set of labelled ground truth logs with both positive and negative traces. We decided to test our miners on the latest (2021) edition of the contest in order to measure their performance on a large set of logs, each of sufficient size to perform out-of-sample cross-fold validation. Note that we did not use the training logs provided as part of the contest. These only contained positive traces and would throw off the balance provided by the ground truth log. The original PDC2021 data set is available at [50]. We relabelled the ground truth logs to fit the format used in the DCR Solutions and Dreyers Foundation logs, so that they can be imported in our prototypes, these are available at [10].
8. Experimental results

We report on exploratory experiments applying two instantiations of the Rejection Miner to the data sets of Section 7, comparing results to current major unary miners. We make no claim to a comprehensive evaluation: The present experiments serve only to produce a preliminary account of how binary mining may differ from unary mining. We find that through the lens of negative examples, we get access to a host of metrics from binary classification.

8.1. Data sets

The DCR Solutions case (Section 7.1) comprises 215 logs, each containing at least one negative example, and each produced by users of the portal to codify what a single model should or should not do. The logs contain 7030 events, 1681 unique activities, 589 negative and 705 positive traces. Logs vary enormously in size: the largest log contains 1162 events, 19 activities, 98 negative and 14 positive traces; the smallest log contain but one negative trace of 3 events. Log size distribution is visualised in Fig. 2. The Dreyers case (Section 7.2) comprises a single log of 10177 events, 33 unique activities, 492 positive and 208 negative traces. The mean trace length is 15 (1–46), and the mean number of activities per trace is 12 (1–24). The PDC2021 data set contains 96 logs, based on an equal number of variations in the underlying model, and each log contains 125 positive and 125 negative traces. All data sets are available online [10].

Both industrial data sets were pre-processed to remove any conflicting traces (i.e. that were both marked as positive and negative for the same log). In addition the DCR Solutions data set had a notion of “optional” traces, but what this meant was not well-defined, therefore these were also removed.

8.2. Metrics

Binary classification mining allows us to rely on traditional machine learning metrics [51] of relative misclassification: true and false positives (TP and FP), true and false negatives (TN and FN). We use in particular the true positive rate (TPR), true negative rate (TNR), accuracy (ACC), balanced accuracy (BAC), positive predictive value (PPV), and F1-score (F1). We recall their definitions in Table 2. These particular measures demonstrate the difference between what can be measured in the unary and binary settings. In the setting of unary-classification miners, where we do not have negative examples, we can count only TP and FN. In that setting, we can only measure the true positive rate (TPR)-known as “fitness” in the process mining community—but none of the other measures. But in the setting of binary-classification miners, we can measure also how well the output model recognises negatives (TNR), how reliable a positive classification is (PPV), and generally how accurately both positive and negative traces are classified using accuracy (ACC) and balanced accuracy (BAC), which weight positive and negative traces equally, resp. proportionally.

We test for generalisation by performing cross-validation on out-of-sample data on the Dreyers Foundation and PDC2021 logs. Out-of-sample cross validation is not only the well-established standard in machine learning, it is also notation agnostic and therefore allows us to compare equally between the different miners. We excluded the DCR Solutions data set from out-of-sample testing because the size of most logs in the data set is too small to perform reliable k-fold validation.

Finally, one goal particular to process discovery is to produce output models that are understandable by humans: Output models are not mere devices for classification; they are vehicles for humans to understand the reasons and structure behind that classification. To this end, smaller models are more helpful, so we calculate also the size of the models, dependent on their notation. For the pattern-based notations such as DECLARE, we use the number of such patterns; for DCR models the number of relations; and for Workflow Nets, directly follows graphs, and transition system models: the raw number of edges in the graph. Of course, sizes for models in different notations are not directly comparable, but they give us an insight into the number of elements that need to be processed by the reader and give a rough indication of relative complexity.

We decided not to use more advanced measures for simplicity because they only apply to a subset of the notations, or are not comparable between notations. For example the notion of logical unit has a different meaning for imperative and declarative notations and the ways in which logical units interact with each other differ significantly across the paradigms. Therefore they affect the complexity of the diagrams in fundamentally different ways. We posit that model size is so obviously not comparable among notations that it should not be misleading for the average reader. In addition, together with the models and qualitative analysis in Section 8.5, it provides a rough estimation of what the resulting models look like.

8.3. Miners

We compare the Rejection Miner (RM1) and Rejection DCR Miner (RM2) to flagship miners for three major process notations, as well as two naive baseline miners. For DCR graphs [32,33], we use DisCoveR [53,54]. DisCoveR is used commercially for model recommendation by DCR solutions. We consider DisCoveR with two settings, the default one (intended to emphasise precision, denoted D), and a “light” version intended to emphasise simplicity (DL). For DECLARE [30,31],

---

Table 2

<table>
<thead>
<tr>
<th>Model class</th>
<th>Log classification</th>
<th>ACC =</th>
<th>TPR =</th>
<th>TNR =</th>
<th>F1 =</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pos.</td>
<td>TP</td>
<td>TN</td>
<td>FN</td>
<td>TP + TN</td>
<td></td>
</tr>
<tr>
<td>Neg.</td>
<td>FN</td>
<td>TN</td>
<td>FN</td>
<td>TP + FN</td>
<td></td>
</tr>
</tbody>
</table>

---

8.4. Discussion

...
we use Minerful [55] and consider three settings, (M1) the most restrictive setting where support = 1.0, confidence = 0.0, and interest factor = 0.0; (M2) a less restrictive setting (likely outputting smaller models) with support = 1.0, confidence = 0.5, and interest factor = 0.25; and (M3) with support = 1.0, confidence = 0.75, and interest factor = 0.5. For Workflow Nets [34], we use the Inductive Miner [1,5], with a noise threshold of 0.0 (IM1) and 0.2 (IM2) respectively. Finally, we include Directly Follows Model Miner (DF) with default noise threshold = 0.8, and Transition System Miner to serve as a baseline.

An important detail regarding Inductive Miner concerns the process of replaying traces on mined models. Since the miner sometimes returns models with many silent transitions, replay can be non-trivial. We used an alignment-based replay package, PNReplayer (7.062) from the ProM framework\(^7\) using the IterativeAStar alignment algorithm. To be considered accepted, a trace must have a trace fitness of 1.0. In the PDC2021 data set, a total of 698 alignments failed to complete within 10 minutes. These traces were simply omitted from metric calculations to avoid biasing results in either direction. Considering the large number of traces (250 per log across 96 logs), runs (10) and folds per run (10), the number of omitted timeouts is negligible, representing only 0.029% of the total 2,400,000 alignments.

### 8.4. Quantitative results

We performed both in-sample and out-of-sample testing, performing 10-fold validation over 10 runs with reshuffling [56]. The results are shown in Table 3. For the DCR Solutions and PDC 2021 data sets results were consolidated for the 215 and 96 logs, respectively. That is, mean values in Table 3 are computed as the mean across all logs, and boxplots in Figs. 3 and 5 are based on means for each log. Because of the limited size of most of the logs, we only tested on in-sample data for the DCR Solutions data set, however, since the primary goal for the company is to find models that accurately fit the training data, in-sample accuracy is highly relevant.

The remainder of this section is structured as follows. We discuss the results for each data set in isolation. For each dataset we (1) start with a discussion on the differences between the corresponding unary and binary miners (i.e. the pairing of the Rejection Miner and Minerful producing Declare models on the one hand, and the pairing of the Rejection DCR Miner and DisCoveR producing DCR graphs on the other), and (2) follow with a more general comparison of the binary miners to the unary miners, also including the results of the imperative miners. We do not provide an in-depth discussion on the influence of the imperative and declarative process modelling paradigms on the results, but instead refer to [57].

**DCR solutions.** First let us compare the Declare-based miners. The Rejection Miner mines perfectly accurate models on every log, using only 1.5 constraints on average. In practice this means that, given a mapping from the Declarative patterns to DCR Graphs, the Rejection Miner will allow the portal to recommend perfectly accurate models for all test cases that have been defined to-date. Minerful requires 123 constraints on average to come close to this result in terms of accuracy. Lowering the accuracy to 0.89 (or 0.85 for balanced accuracy), Minerful only requires an average of 39 constraints, still significantly many more than Rejection Miner. We conjecture that this result shows the importance in using negative examples for pruning the constraints generated by Declare miners.

Now let us compare the DCR-based miners. The Rejection DCR Miner achieves comparable accuracy to DisCoveR light, which is currently used for the task of recommending relations in the DCR portal. However, DisCoveR light suggests 20 relations on average to achieve this result, whereas the Rejection DCR Miner only suggests 1.3 on average. We posit that similar to the Declare-based miner, this shows the importance in using negative examples for pruning the relations

---

\(^7\) https://promtools.org/.
Fig. 4. Performance across 96 logs. Box plots are drawn using the overall mean for each log where mining and evaluation are performed on the same log. See Table 3 for miner abbreviations. Outlier visualisation omitted for readability.

Fig. 5. Performance across 10 runs of 10-fold cross-validation for all 96 logs. Box plots are drawn using the overall mean for each log, rather than every run. See Table 3 for miner abbreviations. Outlier visualisation omitted for readability.
generated by DCR-based miners, which may be an indication that this applies to declarative miners in general.

We also stress the importance of suggesting fewer relations for the business case behind this dataset, as when performing model recommendation, suggesting fewer relations will improve the user's experience of the task, a preference that has been voiced repeatedly by our industry partners. As a result, the Rejection Miner has already been integrated into the portal by the company.

A comparison to the imperative miners is more difficult: the Inductive Miner also manages to get high accuracy and the model size appears to lie between the binary and unary declarative miners. To what extent a model with 20 arcs and places is easier or more difficult to understand for end-users than a model with 1–2 constraints or DCR relations will depend on the exact constructs used in each model and the modeller's experience with each notation. We posit that a model with only 2 visual elements will provide a much clearer general overview than one with 20, although the smaller model may still
yield a higher mental load based on the complex semantics of the constraints involved. The Directly Follows Miner achieves a similar level of accuracy with significantly fewer visual elements, whereas the Transition System Miner performs rather poorly on true negative rate, bringing down its accuracy, but also produces a relatively small model.

**Dreyers foundation.** First we investigate the Declare-based miners. For each tested parameterisation Minerful achieves an accuracy of approximately 0.7, the smallest model achieving this result contains 110 constraints. The Rejection Miner on the other hand achieves an accuracy of 0.99 with only 12 constraints. When we look at the balanced accuracy and true negative rate we see an even stronger picture: whereas the Rejection Miner scores 0.96 on the true negative rate, Minerful fails at correctly classifying almost all negative examples. When we look at the out-of-sample experiment we see that these results generalise quite strongly: Minerful and the Rejection Miner achieve very similar results in terms of accuracy, balanced accuracy and size for both miners. This result paints a rather drastic picture of the performance of Minerful on the Dreyers Foundation log: even when introducing, in the worst case, 1141 constraints, it still fails to identify those constraints that truly matter in terms of disallowing negative behaviour. For the Rejection Miner it not only shows that it is able to achieve excellent results on this log, those results also generalise well towards unseen test data. In other words, providing the miner with some negative examples allows it to accurately predict what other negative examples may be seen in the future. In addition, we can see from the boxplots in Fig. 3 that there is very little variance in the results of the Rejection Miner, with model size and accuracy scores remaining close to the mean for each randomised run of the 10-fold validation.

Now we take a look at the DCR-based miners. The Rejection DCR miner scores somewhat weaker on this log, but notably has the highest TNR of all miners. It appears that the miner either needed to significantly sacrifice positive traces to reject the negative ones, or that lack of formal guarantees in retaining positive soundness hampered its performance. Interestingly it performs better on out-of-sample data than on the in-sample data. Further investigation into this phenomenon could be used to improve the miner, but we leave this for future work. In any case, we can conclude that also the Rejection DCR miner generalises well on the Dreyers Foundation log. When we compare these results to DisCoveR, we see a similar picture as for the Declare-based miners: the unary miner achieves an abyssal true negative rate, hampering accuracy, and produces significantly larger models (71 relations versus 10 relations for the binary miner).

We note here that it is remarkable that the binary miners manage to successfully classify the log with so few constraints or relations. We reflect on this observation in more detail in Section 8.5.

Finally, we relate to the results of the imperative miners. Here we see a more varied picture: the Inductive miner does not score well on accuracy for the in-sample data, but improves noticeably (scoring a true negative rate of 0.37) on the out-of-sample data. The directly follows and transition system miners do surprisingly well. The first scores between the Rejection Miner and Rejection DCR miner in terms of both accuracy and balanced accuracy in both the in-sample and out-of-sample test. The latter does not do as well in the in-sample test, scoring only a true negative rate of 0.1, but performs remarkably better in the out-of-sample test, scoring an accuracy of 0.97, second best of all miners and extremely close to the Rejection Miner. In terms of model size, the imperative miners generate large models containing in most cases over 100 semantic elements. We refer to Section 8.5 for a more in-depth look at what this means in practice. 

**Process discovery contest 2021.** Once again we first examine the Declare-based miners. Both the Rejection Miner and Minerful are able to attain very high accuracy scores, with the Rejection Miner achieving the highest scores of 0.994 accuracy and balanced accuracy, close to a perfect result. In terms of model size Minerful compares particularly poorly to the Rejection Miner on this log. The smallest average number of constraints is 261, and some level of accuracy needed to be sacrificed to reduce this amount. The Rejection Miner requires only 10.5 constraints on average. These results mostly generalise to the out-of-sample test, although both lose some accuracy (see Figs. 4 and 5).

We proceed to the DCR-based miners, where we see a similar picture. The Rejection DCR miner and DisCoveR in fact score almost exactly the same on accuracy and balanced accuracy, but there is an extremely difference in the number of relations required by each miner to achieve this result. Whereas DisCoveR finds on average 239 relations, the Rejection DCR miner produces models with an average number of relations of 6.7. In the out-of-sample test we see that these results generalise very closely. Together with the observations from the other logs, this supports our conjecture that having access to negative examples can help miners determine which constraints truly matter and should be mined.

We once again note here that it is remarkable that the binary miners manage to classify the log with so few constraints and relations in the in-sample test. In particular this means that the miners could find models that satisfied all test cases while the majority of the activities were determined to be completely unconstrained. We believe that this is an interesting topic for further investigation. In particular binary miners could perhaps be used as a tool for testing the coverage of the test sets of future contests. But we leave this for future work.

We also investigate the imperative miners. Here we see that the Inductive Miner and directly follows miner perform well on the in-sample test, and exceptionally well on the out-of-sample test, with the directly follows miner in fact scoring the highest accuracy and balanced accuracy of all miners. It is particularly surprising that the Inductive miner performs much better than in the contest itself, but this is likely caused by the fact that we only train on the (noise free) test data of the contest. This appears to allow it to find a better model than when trained on the full training log. The transition miner on the other hand performs more poorly, in particular because of a low true negative rate. We once again see a large difference in model size, and while we leave the reader to come to their own conclusions on what this means in practice, we would argue that for this log the difference in size between in particular the Rejection DCR miner and Inductive miner, rather clearly indicates a significantly more complex model.

Finally we believe that the discrepancy in terms of true negative rate between the Dreyers Foundation and PDC2021 log is worthy of further investigation, in particular when it comes to the declarative miners. It would appear that the Dreyers log manages to capture behaviour that is very difficult for miners to classify without counter examples, whereas the PDC log appears to capture behaviour that can be classified even without knowing negative examples. This could provide further insights into the inner workings of the mining algorithms, and possibly the Dreyers log could provide inspiration for new challenges in future contests. 

**Run-time performance.** Because the experiments ran across different environments and languages (e.g. JavaScript running directly in a browser versus JAVA Bytecode running on the JVM), we did not include an detailed overview of run-time performance in the results. On the Dreyers log the Rejection Miner took on average 11.6 s in the out-of-sample test, which was one order of magnitude slower than Minerful, and several orders of magnitude slower than DisCoveR and the Inductive Miner. The Rejection DCR Miner took on average 2.8 s. Since the initial publication of this paper, the rejection mining mechanism has been replicated as a constraint satisfaction problem using all standard Declare constraints by Chesani et al. [58], which significantly increases run-time performance.
8.5. Qualitative results: the Dreyers Foundation log

To also provide a more qualitative perspective on these results, we compare some of the models generated by the various miners. We chose to base these on the Dreyers Foundation log as it is significantly more complex than the individual logs in the DCR Solutions dataset, and clearly represents a real-life process, thereby giving more meaning to the models.

We start by comparing between the Rejection Miner (Fig. 6) and Minerful (Fig. 7), and the Rejection DCR Miner (Fig. 8) and DisCoveR (Fig. 9), as these form two pairs of a binary and unary miner that produce a model in the same process notation (respectively Declare and DCR Graphs). For both pairings we see similar results. The model mined by the unary miner is approximately 7–8 times larger than the model mined by the corresponding binary miner, and this has a strong effect on the visualisation: a good layout of the model appears to be impossible because of the high number of overlapping constraints and relations. This poor layout, combined with an overload of information, makes the models poorly suited to getting an understandable overview.
of the process. Their value lies primarily in inspecting individual activities and their immediate constraints or relations. The size of the models produced by the binary miners is more manageable and this makes the conveyed information much more straightforward to parse for the user.

It also becomes apparent however that the binary-mined models do not capture the entire process: less than half of the activities are involved in a constraint. We posit that this likely a result of how the log was generated: a process definition was given to a system, the system ran this process, and when something went wrong those traces were marked as undesired. As a result, there are only negative examples of scenarios that were not already captured successfully in the original model and thus the constraints proposed by the miners focus on fixing these mistakes, not re-discovering the constraints that already worked as intended. This appears to be supported by the constraints that have been found, for example the response between “Undo payment” and “Første udbetaling” (“first payment” in Danish), found by each binary
miner, likely shows that the initial model failed to account for the fact that when the first payment was undone for some reason (e.g. a change in bank account numbers making the payment fail), it should be repeated. Finally we note, as already shown in the quantitative analysis, that despite the models being smaller, and their constraints not being exhaustive, the binary models are still noticeably better at disallowing known bad behaviour (as shown by the high true negative rate of these models).

Based on these observations, we conclude that (1) for the rejection miner to find a complete model, it requires a sufficiently exhaustive set of negative examples, and (2) the rejection miner appears to be especially well-suited to model improvement tasks after the process has been running for some time and bad executions have been identified.

Finally, we have a look at the models created by the Inductive Miner (Fig. 10) and the Directly Follows Miner (Fig. 11). For the Inductive Miner, a concrete comparison to the binary miners is difficult to make: it contains many more graphical elements, but these may be seen as easier to parse for an average user than the constraints and relations shown in the declarative models. On the other hand, the high degree of looping and the many silent transitions in the Petri net are a clear detriment to its understandability. We will leave the reader to come to their own conclusion on which notation is preferred and focus on how much the model tells us about the actual process. Inspecting the Inductive Miner model more closely we can see that the overzealous use of loops and silent skip transitions effectively turns it into a flower model accepting almost any trace, which does not appear very useful for informing the correct semantics of the process. This is supported by the quantitative results for the model: the perfect true positive rate and true negative rate of 0.02 supports the notion that the model simply allows almost all behaviour and is not expressing any meaningful information about the control flow of the underlying process.

Inspecting the directly follows graph we can see that the lack of precision of the Inductive Miner model is likely caused by the high density of and many cycles among the directly follows relations. We see a similar picture to the Inductive Miner, where a concrete comparison to declarative models is difficult to make, but the graph certainly poses some challenges for the user to parse based on the high inter-connectivity of the elements.

We can see from these examples that imperative miners appear to have a particularly hard time capturing the Dreyers Foundation log, which coincides with earlier observations made on this particular dataset [49]. In addition, making a fair comparison between the declarative models generated by our binary miners and imperative models is difficult without employing actual users studies. However, the fact that the imperative miners often default towards modelling flower models does indicate that there may be a benefit in using negative examples to prune these flowers and make the models more precise. Future experiments on additional logs may yield better insights into how such pruning could be implemented in practice, especially when the miners find more meaningful models based on the positive examples.

9. Conclusion

In this paper we build on earlier work in binary process discovery [7,8,13,14] by providing an automatically verified formalisation of process discovery as a binary classification problem. We provided a formal account of when binary miners exist; proposed the generalised Rejection Miner algorithm; provide two implementations of the algorithm, one based on conjunctive normal form temporal logic patterns (specifically: a subset of Declare combined with disjunctive and conjunctive response constraints); introduced real-world cases of negative examples; and compared the Rejection Miner to contemporary unary miners for DCR Graphs, Declare, and Petri nets. Our experiments showed a general increase in accuracy, with a particularly large improvement for one of the real-world logs. Most noticeably, these increases in accuracy are achieved with significantly fewer model elements, showing that the inclusion of negative examples can help miners pin-point exactly which model constructs are relevant to capture the behaviour of the process and thereby improve model simplicity.

Future work. We expect to pursue the creation of additional labelled real-world logs, based on the uses cases from municipal government and additional industrial partners. We intend to further develop the Rejection DCR Miner by considering more constraint patterns and optimising the algorithm, for example by an encoding of the computations into bit vector operations, which forms the basis for the high performance of DisCoverR [48], an apriori sub-selection of relevant constraint templates, as seen in the Declare miner [59], the use of the refinement results of [43] and event independence results of [60] to avoid unnecessary computations, and generally improve the user experience and lift the miner from a prototype to a production tool. We also expect to pursue the mining of labelled DCR Graphs, which could lead to the development of a perfect binary DCR miner. Finally we will investigate other approaches towards binary mining, such as constraint solving and optimisation (e.g. genetic) algorithms.
Fig. 11. Model mined on the in-sample Dreyers Foundation log by the directly follows miner.

CRediT authorship contribution statement

**Tijs Slaats**: Conceptualization, Data curation, Formal analysis, Funding acquisition, Investigation, Methodology, Project administration, Software, Supervision, Validation, Visualization, Writing – original draft, Writing – review & editing. **Søren Debois**: Conceptualization, Data curation, Formal analysis, Investigation, Methodology, Software, Validation, Visualization, Writing – original draft, Writing – review & editing. **Christoffer Olling Back**: Conceptualization, Formal analysis, Investigation, Methodology, Software, Validation, Writing – original draft, Writing – review & editing. **Axel Kjeld Fjelrad Christfort**: Conceptualization, Formal analysis, Investigation, Methodology, Software, Validation, Visualization, Writing – original draft, Writing – review & editing.

Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: Søren Debois reports a relationship with DCR Solutions.
that includes: employment. Tijjs Sluats reports a relationship with DCR Solutions that includes: consulting or advisory. DCR Solutions provided data used in the experiments for the paper.

Data availability

Data will be made available on request.

Acknowledgements

This work is in part supported by Digital Research Centre Denmark (DIREC) through the AI and Blockchains for Complex Business Processes project. We wish to express our deep gratitude towards Sergio Tessaris, Chiara Di Francescomarino and Fabrizio Maria Maggi for their efforts in replicating our results and making us aware of the discrepancies they discovered in the output of the rejection miner, which led to the detection and correction of an error in the evaluation of a number of patterns. We also wish to thank DCR Solutions for providing data used in the experiments for the paper.

References


