UnconstrainedMiner: Efficient Discovery of Generalized Declarative Process Models

Michael Westergaard[1,2⋆], Christian Stahl[1], and Hajo A. Reijers[1,3]

[1] Eindhoven University of Technology, Den Dolech 2, 5600 MB, The Netherlands
[2] National Research University Higher School of Economics, Moscow, 101000, Russia
[3] Perceptive Software, Piet Joubertstraat 4, 7315 AV Apeldoorn, The Netherlands
m.westergaard@tue.nl, c.stahl@tue.nl, h.a.reijers@tue.nl

Abstract. Process discovery techniques derive a process model from observed behavior (e.g., event logs). In case of less structured processes, *declarative models* have notable advantages over procedural models. A declarative model consists of a set of *temporal constraints* over the activities in the event log. In this paper, we address three limitations of current discovery techniques: their *unclear semantics* of declarative constraints for business processes, their *non-performative discovery* of constraints, and their *potential identification of vacuous constraints*. We implemented our contributions as a declarative discovery algorithm for the Declare language. Our evaluations on a real-life event log indicate that it outperforms state of the art techniques by several orders of magnitude.

1 Introduction

For purposes of quality management and performance improvement, organizations are interested in the way their internal business processes are executed. Such an understanding can be gained by *observing* process behavior as recorded in the form of an *event log*. Event logs may be extracted from databases, message logs, or audit trails. Given an event log, *process discovery* techniques can be used to produce a process model [3].

Less structured processes such as health-care processes can often easier be captured using a *declarative* rather than a *procedural* approach. While a procedural model describes how the process has to work exactly, a declarative model describes only the essential characteristics of the process. To this end, *constraints* are specified that restrict the possible execution of activities. Over the last years, declarative languages, such as Declare [4,17] (formerly known as DecSerFlow), DCR Graphs [10] and Montali's logic based framework [15], have been developed and integrated in academic and industrial modeling tools [20]. There also exist tools to discover a declarative model from an event log. Examples are the ProM Declare miner [12] and the MINERful++ tool [7].

Despite advances in discovering declarative process models over the last years, current discovery techniques are subject to limitations. We will make these limitations explicit, describe how related approaches have attempted to deal with these, and describe our contributions against that background.

⋆ Support from the Basic Research Program of the National Research University Higher School of Economics is gratefully acknowledged.

Limitation 1: Unclear semantics Our interest is in the application of process discovery for business processes. An important implication is that the executions of interest, as recorded in event logs, are *finite*. For this reason, too, the semantics of the Declare miner is based on finite linear temporal logic (LTL). The limitation that exists is that for *infinite* traces, the LTL semantics [18] is well-understood—a constraint (i.e., an LTL formula) is translated into a Büchi automaton that can then be used for verification [19]—for finite traces, there is *currently no agreement on how the LTL semantics should be defined* [5, 8, 9, 16]. Existing approaches deal with this limitation in various ways. The current Declare miner uses a slight modification of the algorithm proposed in [9] to construct for a constraint a finite automaton instead of a Büchi automaton. By exploiting the structure of Declare constraints, Westergaard [21] improves the algorithm in [9] dramatically, making declarative specifications scale to realistic settings. Bauer et al. [5] show that a classical 2-valued semantics does not suffice and propose 3-valued and 4-valued semantics instead. Still, these semantics on finite traces do not converge with the semantics on infinite traces [16]. Therefore, Morgenstern et al. [16] propose to consider four classes of LTL properties and define a semantics for each class. However, dealing with (several) 3-valued or 4-valued semantics as proposed in [5,16] overly complicates the usage of declarative techniques and the development of tools.

Therefore, we decided to follow a different approach. The first contribution of this paper is a *redefinition of the original Declare semantics* [4] using *regular expressions*. Regular expressions provide sufficient expressiveness, and are easier to explain to practitioners than the semantics in [5, 16]. More importantly, as every regular expression describes a regular language, which can be represented by a finite automaton, we have a mapping from declarative constraints to finite automata. The semantics of finite automata is well-defined, and operations such as product and complement are decidable. The MINERful++ tool [7] also uses regular expressions. The semantics of several constraints described in [7] is wrong, but has been recently corrected. MINERful++ does not support choice constraints; that is, it only supports a subset of all Declare constraints. Instead, we handle all constraints of Declare.

Limitation 2: Non-performative or incomplete discovery The Declare miner in ProM [12] systematically checks one constraint after another (for each possible instantiation of parameters) and returns a model comprising all constraints. Because of this brute-force approach, this miner has severe difficulties with event logs of realistic business processes. Alternative approaches try to alleviate the drawbacks of handling all constraints associated with realistic business processes. These include forcing users to pick among interesting constraints [13], employing a priori reduction to avoid considering rarely occurring events [13], and considering the relationships between constraints to avoid mining some constraints [14]. The limitation, here, is that the mining of declarative models is either non-performative or too selective.

Our contribution is that we stick to mining *all* constraints, but employ post-processing to weed out uninteresting constraints. As such, we potentially obtain better resulting models than making filtering beforehand out of necessity. Any post-processing (and complexity-reducing filtering) possible with existing miners is also possible with our approach. Our approach just allows more intelligent post-processing due to having more

information at its disposal. To make our technique performative, we present four powerful *reduction techniques*, including symmetry reduction and parallelization.

The most related approach is provided by the MINERful++ tool [7], which computes statistics about event relationships and uses that to infer constraints. The approach is fast, but computation of constraints from statistics makes it difficult to add new constraints. The reason is that it requires to develop and prove rules for doing so. For example, it is far from obvious how to extend this approach to also mine choices. In contrast, our approach supports all Declare constraints and it can cope with *any constraint that can be specified as a regular expression*.

Limitation 3: Irrelevant discovery A discovered constraint may hold although it is irrelevant. This phenomenon has been coined as *vacuity* in model checking. Vacuity detection in model checking aims to discover meaningless satisfaction of the specification. Usually, mutations of subformulae of the specification are used to discover vacuity. Much research has been conducted (see [11] for a survey). Vacuity is also relevant for process discovery. In contrast to an LTL formula, a constraint does not contain subformulae. Consequently, vacuity refers to discover events that are prevented.

Vacuity has been incorporated in the *support* metric of a constraint which is the fraction of traces in which the constraint is nonvacuously satisfied [7, 13]. There are, however, Declare constraints for which the confidence value cannot be expressed. Examples are the choice constraints.

In this paper, we *generalize support to all Declare constraints and even to any regular expression*. We define positive and negative support. Positive support coincides with support in [7,13], whereas negative support occurs for all constraints that could not be considered by [7, 13]. Moreover, our support notions are *independent of the syntax of a constraint* and can be used to compute the confidence of a constraint.

To demonstrate the feasibility and performance of our approach, we implemented the *UnconstrainedMiner* as a plug-in in the tool ProM [1]. The UnconstrainedMiner uses the Declare language from [4], but the semantics is defined on regular expressions. On top of the language, we implemented the novel discovery algorithm and the notion of positive and negative support of constraints. The proposed reduction techniques enable us to discover all constraints for event logs obtained from processes of realistic size within seconds. The tool thereby outperforms the state of the art tools by several orders of magnitude. As an additional feature, the UnconstrainedMiner is also *extensible*: It allows users to add any constraint that can be defined as a regular expression. As our notion of support is generic, it is automatically computed for added constraints. A first version of the UnconstraintMiner has been presented at the BPM Demos [22]. In this paper, we present an extended version of the tool and the theory, including the novel notion of positive and negative support.

We continue with a motivating example and then provide background information in Sect. 3. We define the Declare language using regular expressions in Sect. 4. We present our discovery algorithm in Sect. 5 and define the relevance of discovered constraints and vacuity in Sect. 6. Sect. 7 presents experimental results and Sect. 8 concludes the paper.

2 Motivating Example

Figure 1 shows (part of) a process and an event log consisting of 100 traces. In the process, either the upper or the lower block is executed. The average probability for each block is 10% and 90%, respectively. The idea is that a high payment should be preceded by a double check. Furthermore, a payment should not be made if rejected. Often, we do not have a model like (a) available, but only the event log in (b). This is where process discovery comes in. Example constraints are precedence(Double Check, Pay High) (i.e., Pay High cannot occur before Double Check), response(Double Check, Pay High) (i.e., Double Check is eventually followed by Pay High), and exclusive choice1of3(Pay High, Pay Low, Reject) (i.e., exactly one of the three events occurs).

A discovered constraint may hold although it is irrelevant. As an example, consider the constraint response(Pay High, Pay Low). Obviously, both events are unrelated (they are in exclusive branches) but the constraint holds for 94 traces. We will pick up on the example in subsequent sections to illustrate our approach.

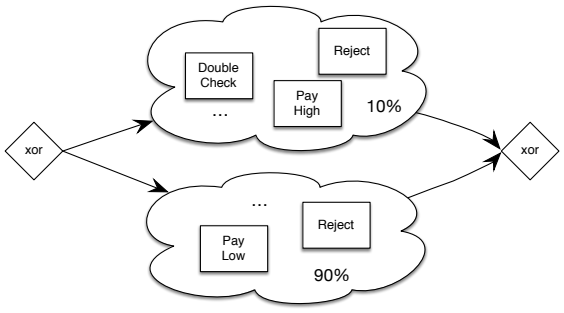
3 Preliminaries

We define (finite) automata and regular expression, which we shall use to define the semantics of the Declare language.

Definition 1 (finite automaton). *A* finite automaton $A = (S, \Sigma, \delta, s_0, F)$ *consists of a finite set* s *of* states*, a finite* alphabet Σ*, a* transition relation $\delta \subseteq S \times \Sigma \times S$ *on states, an* initial state $s_0 \in S$*, and a set* $F \subseteq S$ *of* accepting states*. If for all* $s, s_1, s_2 \in S$ *and* $a \in \Sigma$*,* $(s, a, s_1), (s, a, s_2)$ *implies* $s_1 = s_2$*, then* A *is* deterministic*.*

As usual, $\mathcal{L}(A)$ denotes the language of A; ε denotes the empty trace; and for two finite traces $\sigma, \sigma' \in \Sigma^*$, $\sigma\sigma'$ denotes their concatenation.

The Declare language allows anything not explicitly constrained, so we always encode a Declare constraint as a finite deterministic automaton with a special symbol (typically -) indicating any action not explicitly specified.



(a) Abstract process

#	trace
1	Pay High
4	Double Check, Reject
5	Double Check, Pay High
20	Reject
70	Pay Low
100	

(b) Event log L

Fig. 1: Running example

Given two deterministic finite automata A and B, we define their *synchronous product* AB and the *complement* A^c of A in the standard way. Furthermore, recall that for every deterministic finite automaton A with language $\mathcal{L}(A)$, there exists a deterministic finite automaton A' that has the minimal number of states and $\mathcal{L}(A) = \mathcal{L}(A')$. All these operations are computable and efficient.

A regular expression describes a regular language which can be represented by a finite automaton.

Definition 2 (regular expression). *A* regular expression *over an alphabet Σ and the language it represents are inductively defined by*

- *$\emptyset, \varepsilon$ and . are regular expressions, and $\mathcal{L}(\emptyset) = \emptyset$, $\mathcal{L}(\varepsilon) = \{\varepsilon\}$ and $\mathcal{L}(.) = \Sigma$.*
- *Each $a \in \Sigma$ is a regular expression and $\mathcal{L}(a) = \{a\}$.*
- *If α, β are regular expressions, so are the alternation $\alpha \mid \beta$ with $\mathcal{L}(\alpha \mid \beta) = \mathcal{L}(\alpha) \cup \mathcal{L}(\beta)$ and the concatenation $\alpha\beta$ with $\mathcal{L}(\alpha\beta) = \{ab \mid a \in \mathcal{L}(\alpha), b \in \mathcal{L}(\beta)\}$.*
- *If α is a regular expression, so is α^* and it is the fixpoint of $\alpha* = \varepsilon \mid \alpha\alpha^*$.*

Let α be a regular expression and $a_1, \ldots, a_n \in \Sigma$. For a more compact notation, we introduce the quantifier ? defined by $\alpha? = \alpha \mid \varepsilon$; $[a_1 \ldots a_n]$ is a short hand for $a_1 \mid \cdots \mid a_n$ and $[\hat{}a_1 \ldots a_n]$ denotes the complement of $[a_1 \ldots a_n]$ (i.e., none of the characters $a_1 \ldots a_n$ matches); finally $\alpha\{n\}$ denotes that α is matched n times.

A regular expression α can be transformed to a finite automaton $A(\alpha)$ such that $\mathcal{L}(\alpha) = \mathcal{L}(A(\alpha))$.

4 The Declare Language

As indicated, using finite LTL for the Declare semantics is problematic. Already in [7], Di Ciccio and Mecella use regular expressions instead. However, the semantics in [7] describes some constraints incorrectly and some constraints not at all. In this section, we provide a full semantics for *all constraints of Declare* using regular expressions.

Table 1 contains the semantics of all Declare constraints. Due to space limitations, we shall not provide details for each of them. For the informal semantics, we refer to [17], and for the intuition about some of the regular expression semantics, we refer to [7]. Our semantics differs for a significant portion from [7] as to fix subtle errors. Also, the semantics for the choice constraints is new. The semantics is intended to capture our intuition of the constraints and, as a result, some expressions seem more complex than necessary. For example, precedence(a,b) says that b cannot occur before an a; there are alternative expressions possible, but in this way it generalizes nicely to the response and succession constraints, and to the alternate and chain variants. A constraint like exclusive choice1of3 is complex and details the three possible situations—any one of the three parameters occur and the other two do not—and illustrates that the declarative approach can capture a complex procedural situation nicely.

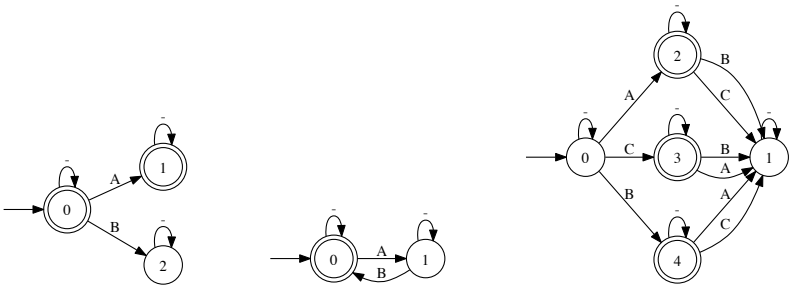
For each constraint C, the automaton of the constraint, $A(C)$, is $A(C) = A(sem(C))$. Figure 2 shows the automata for precedence, response, and exclusive choice1of3. To instantiate a constraint, the parameters $(a, b, c, \ldots)$ are replaced with concrete event classes. The special symbol - indicates that any action not explicitly listed may be chosen (so all three automata accept the string "C"). The automata representation for all constraints can be found in the appendix.

Table 1: Declare constraints and their formal semantics.

	Constraint (C)	Formal semantics ($sem(C)$)
position	init(a)	(a.*)?
	strong init(a)	a.*
	last(a)	.*a
count	existence(a,n)	.*(a.*){n}
	absence(a,n)	[^a]*(a?[^a]*){n-1}
	exactly(a,n)	[^a]*(a[^a]*){n}
ordered	precedence(a,b)	[^b]*(a.*b)*[^b]*
	response(a,b)	[^a]*(a.*b)*[^a]*
	succession(a,b)	[^ab]*(a.*b)*[^ab]*
	alternate(a,b)	[^a]*(a[^a]*b[^a]*)*a?[^a]*
	alternate precedence(a,b)	[^b]*(a[^b]*b[^b]*)*
	alternate response(a,b)	[^a]*(a[^a]*b[^a]*)*
	alternate succession(a,b)	[^ab]*(a[^ab]*b[^ab]*)*
	chain precedence(a,b)	[^b]*(ab[^b]*)*
	chain response(a,b)	[^a]*(ab[^a]*)*
	chain succession(a,b)	[^ab]*(ab[^ab]*)*
unord.	responded existence(a,b)	[^a]*((a.*b.*)|(b.*a.*))?
	co-existence(a,b)	[^ab]*((a.*b.*)|(b.*a.*))?
choice	choice1of2(a,b)	.*[ab].*
	choice1of3(a,b,c)	.*[abc].*
	choice1of4(a,b,c,d)	.*[abcd].*
	choice1of5(a,b,c,d,e)	.*[abcde].*
	choice2of3(a,b,c)	.*((a.*[bc])|(b.*[ac])|(c.*[ab])).*
	exclusive choice1of2(a,b)	([^b]*a[^b]*) | ([^a]*b[^a]*)
	exclusive choice1of3(a,b,c)	([^bc]*a[^bc]*) | ([^ac]*b[^ac]*) | ([^ab]*c[^ab]*)
	exclusive choice2of3(a,b,c)	([^c]*((a[^c]*b) | (b[^c]*a))[^c]*) | ([^b]*((a[^b]*c) | (c[^b]*a))[^b]*) | ([^a]*((b[^a]*c) | (c[^a]*b))[^a]*)
negative	not co-existence(a,b)	[^ab]*((a[^b]*) | (b[^a]*))?
	not succession(a,b)	[^a]*(a[^b]*)*
	not chain succession(a,b)	[^a]*(a+[^ab][^a]*)*a*

5 Discovering Regular Expressions from Event Logs

The ProM Declare miner [12] mines constraints by translating the finite LTL semantics into a finite automaton, replaying a log on the automaton, and accumulating statistics to accept constraints according to a given threshold. The MINERful++ miner [7] instead computes statistics about event relationships and uses that to infer constraints. While the MINERful++ approach is faster, it is less general and cannot handle all Declare constraints. We, therefore, use the ProM Declare miner as a starting point. We use various optimization techniques to make our miner perform better than previous miners. Aside from an efficient base implementation, we propose four reduction techniques for improvement: symmetry reduction, prefix sharing, parallelization, and super-scalarity.



(a) (b) (c)

Fig. 2: Automata for precedence (a), response (b), and exclusive choice1of3 (c).

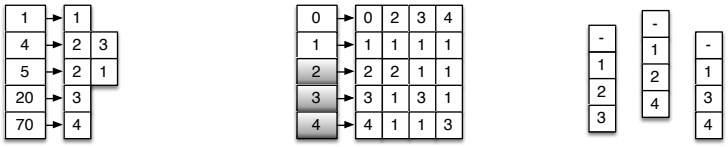
5.1 Efficient Base Implementation

Given a log file consisting of a set of traces, the base idea is to gather all different events of the log (yielding a set of event classes), instantiate all Declare constraints using all possible combinations of event classes, and check whether each trace belongs to the language of the resulting automaton (by following transitions and checking whether we end in an accepting state). For the example in Fig. 1, the event classes are $\{$Double Check, Pay High, Pay Low, Reject $\ldots\}$. Examples for constraint instances are precedence(Double Check, Pay High), precedence(Double Check, Pay Low), and response(Double Check, Pay High).

To make this simple approach more efficient, we first enumerate all encountered event classes. This allows us to refer to events using an integer instead of text string or even a structured entry, which makes comparison faster. We get a representation of the log from Fig. 1 like in Fig. 3(a). Then, we realize that we need not instantiate constraints repeatedly, but only maintain a translation table. We, therefore, enumerate all labels of the automaton obtained from a constraint. We can now represent logs as sets of lists of integers (Fig. 3(a)), the automaton of a constraint as a look-up table mapping states and label numbers to new states (e.g., Fig. 3(b) shows the automaton for precedence), and an instantiation of a constraint as a map from event identifiers to label identifiers (Fig. 3(c)). In practise, we represent these as arrays of arrays of integers and arrays of integers, making the representation compact and allowing for fast operations.

5.2 Symmetry Reduction

While the base algorithm is intriguingly simple and with its efficient base implementation already outperforms all previous miners, it does perform superfluous work. Basically, we still replay every trace on every instantiation of every constraint, leading to $tn(n-1)\cdots(n-p+1)$ replays for a $O(ltn(n-1)\cdots(n-p+1)$ performance, where l is the length of the longest trace, t is the number of traces, n the number of event classes, and p the number of parameters of the constraint. If $t = 10,000$, $n = 20$, and $p = 5$ for the choice1of5 constraint, we have to perform a total of more than 18.6 billion replays.



(a) Log from Fig. 1b using 1=Pay High, 2=Double Check, 3=Reject, and 4=Pay Low

(b) Automaton from Fig. 2c using index 0=-, 1=A, 2=B, and 3=C

(c) Some/all (depending on symmetry) mappings with A $\mapsto$Pay High

Fig. 3: Representations of logs (a), automata (b), and instantiations of constraints (c).

Some constraints display a notion of symmetry, however. This means that swapping some parameters does not change the validity of the constraint. For example, the constraint choice1of2(Pay High, Pay Low) is the same as choice1of2(Pay Low, Pay High). In general, we can have multiple such symmetry subgroups. We only need to check for one member of a symmetry subgroup to check for all of them, which means that each symmetry group G reduces the number of comparisons by the number of possible permutations of the elements of the groups or $|G|!$. Often, a constraint is symmetric in all parameters, leading to a reduction of a factor of $p!$. For example, the choice1of5 constraint is symmetric in all parameters, and forcing us to make "only" 155 million replays as opposed to 18.6 billion. If a constraint is symmetric in all parameters, we reduce the complexity from $O(ltn(n{-}1)\cdots(n{-}p{+}1))$ to $O(ltn(n{-}1)\cdots(n{-}p{+}1)/p!) = O(lt\binom{n}{p})$.

We can automatically check if a constraint is symmetrical in two parameters by checking if the regular expression obtained by swapping two parameters has the same language as the original regular expression. This is easily done by checking that the intersection of one and the complement of the other is empty, and vice versa—that is, whether $\mathcal{L}(A(C)) \cap \mathcal{L}(A(C[p_i \mapsto p_j, p_j \mapsto p_i])^c) = \mathcal{L}(A(C[p_i \mapsto p_j, p_j \mapsto p_i])) \cap \mathcal{L}(A(C)^c) = \emptyset$, for all $p_i, p_j \in G$.

5.3 Prefix Sharing

When a log stems from a single process, the individual traces typically share a lot of characteristics. We can exploit this to achieve improved performance. The idea is to organize the log in a prefix-tree or *trie*. This is a tree where each node corresponds to an event and each edge to one event following another in an execution. Figure 4 shows a log and an example trie representation.

We can replay such a trie on an automaton in a single run, allowing us to not replay any shared prefix more than once. Formally, this is done by constructing the synchronous product of the automaton and the trie, but in practise we can do it a bit simpler. We annotate the trie with the number of traces terminating in each node. Now, as we replay the log on the automaton, we annotate each node of the trie with the state in the automaton corresponding to it.

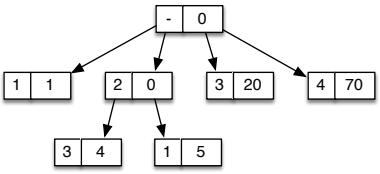


Fig. 4: Trie representation of the log from Fig. 3a. The number to the left of each node indicates the event, and the number to the right how many traces end in this node.

5.4 Parallelization

Our algorithm does not make any global assumptions and does not depend on global computations. This means that it should be easy to parallelize. We can parallelize either in constraints or traces. Parallelization requires that we can decompose the problem and later compose the solution. The ProM Declare miner [12] depends on global computations to do its a priori reduction, meaning it can only parallelize on constraints. MINERful++ could in principle perform its global computations in parallel, but does not do so. We make no assumptions on global computations and all values computed are just the number of traces which satisfy various properties (whether a trace is accepted by a constraint and the various vacuity properties detailed in the next section), we can compose on both traces and constraints. Experiments show that, in practise, the most complex constraint dominates the computation; so unless we have many constraints that are as difficult as the most complex one, decomposing on constraints will yield little reduction. It is possible to split up checking of instances of a single constraint, but it is complicated. In contrast, if we can split up checking by traces, this is much simpler. We, therefore, parallelize on traces.

5.5 Super-scalarity

Previous miners already try to use relationships between constraints to reduce computation efforts. Unfortunately, when used as a reduction, this does not improve the worst-case computation time and additionally imposes a check for each instance of each constraint. Moreover, it will either force users to select threshold levels before mining or yield imprecise threshold levels as accurate thresholds cannot be inferred. Therefore, we propose to check multiple constraints at a time instead. We have already demonstrated how to check multiple constraints at the same time [21] (by having acceptance states for each constraint) and extend this approach to mining. If care is taken, replaying a log on an automaton recognizing multiple constraints is not more time-consuming when replaying. Furthermore, such an automaton can be constructed once for each set of constraints, and instantiated the same way as our base algorithm, effectively allowing us to mine multiple constraints at a time, much how like super-scalar processors execute different parts of multiple instructions at a time.

When we combine constraints, we have to consider two things: symmetry subgroups and memory use. Large symmetry subgroups reduce execution time, so it is bad to split up a subgroup. When we combine automata, the size of the product is the product of

the individual sizes in the worst case, so we should only combine related constraints. To avoid breaking symmetry subgroups, we require that to allow combining two constraints, C_1 and C_2, any symmetry subgroups of C_1 is fully contained in exactly one symmetry subgroup of C_2.

Definition 3 (combining constraints). *We* combine *two constraints C_1 and C_2 if the following three statements hold: (1) $\mathcal{L}(C_1) \supseteq \mathcal{L}(C_2)$; (2) for all symmetry subgroups G_1 of C_1, there exists a symmetry subgroup G_2 of C_2 such that $G_1 \subseteq G_2$; and (3) any for symmetry subgroup G of C_1 with $G \cap G_2 \neq \emptyset$, we have $G = G_1$.*

The first condition in Definition 3 thereby ensures that we only combine related constraints. We can easily check the symmetry requirements using simple set operations and the implication requirement using $\overline{\mathcal{L}(C_1)} \cap \mathcal{L}(C_2) = \emptyset$.

Our requirements for combining are not symmetrical, but capture that we may merge something simpler to something more complex; that is, we do allow adding a smaller symmetry group to a larger. While this means we may check a simple constraint more times than necessary, we do so for free, as the cost is already incurred by the more complex constraint. This is reflected in Definition 3, which is asymmetric and basically allows adding something simple to something complex as long as neither is made more complex.

6 Vacuity Detection

The idea of vacuity checking in model checking is to check that all parts of a formula matter to the truth of the full formula. This is done by detecting subformulae that can be replaced by any stronger variant without affecting the truth value. In practise, a subformula is replaced by either true or false depending on whether it occurs positively or negatively. As formulas are often translated to negative normal form, where all negations are moved all the way in front of atomic propositions, all subformulae except atomic propositions occur positively and can be replaced by false. A similar method exists for regular languages [6]. This approach tries to mimic the ideas from LTL by quantifying over all sequences. This neglects two facts of Declare: Declare does not use complex event relationships in single constraints, and some constraints cannot be vacuously satisfied. This allows us to do faster and better vacuity tests.

Declare does not have complex relationships between atomic propositions (tasks) because no constraint has more than two symmetry subgroups. We use this to look at constraints in isolation. Conceptually, we add absence constraints prohibiting subsets of tasks, much like how support is computed for LTL. If the outcome allows any sequence of events (except those including the prohibited ones), the constraint is not interesting unless at least one of the tasks occur.

Definition 4 (positive support). *Let C be a constraint with parameters $p_1, \ldots, p_p$ and $\mathcal{L}(C) \cap \mathcal{L}([\hat{}p'_1 \cdots p'_k]^*) = \mathcal{L}([\hat{}p'_1 \cdots p'_k]^*)$ with $p'_i \in \{p_1, \ldots, p_p\}$, for $1 \leq i \leq k$. Then, C is* vacuously satisfied *for a trace σ if $\sigma \in \mathcal{L}([\hat{}p'_1 \cdots p'_k]^*)$. A trace σ has* positive support *for C if σ is a member of the language of the* positive support automaton $Pos(C) = \prod_{\mathcal{L}(C) \cap \mathcal{L}([\hat{}p'_1 \cdots p'_k]^*) = \mathcal{L}([\hat{}p'_1 \cdots p'_k]^*)} A^c([\hat{}p'_1 \cdots p'_k]^*)$.

A constraint C can have multiple sets $p'_1 \cdots p'_k$ (which we represent with $Pos(C)$) and it has positive support for a trace σ if σ is a member of any such set. We use $Pos(C)$ to represent all those sets $p'_1 \cdots p'_k$. Thus, having positive support intuitively means the constraint was satisfied but not by chance (by avoiding triggering anything that could make it unsatisfied). In general, $Pos(C)$ looks like Fig. 5, where the set of labels is the union of all labels that may trigger the constraint C. As an example, we have the label set $\{\mathsf{B}\}$ for precedence(A,B) and $\{\mathsf{A}\}$ for response(A,B).

Having a notion of positive support, it makes sense to also look at *negative support*. The intuition is to look at what tasks are necessary to satisfy a constraint. We again look at what happens if we exclude a subset of parameters. Instead of considering the case when we accept everything else, we look at what happens if we accept nothing at all.

Definition 5 (negative support). *Let C be a constraint over parameters $p_1, \ldots, p_p$ and $\mathcal{L}(C) \cap \mathcal{L}([\hat{}p'_1 \cdots p'_l]^*) = \emptyset$ with $p'_i \in \{p_1, \ldots, p_p\}$, for $1 \leq i \leq l$. A trace σ has* negative support *for C if σ is not a member of the language of the* negative support automaton $Neg(C) = A(|_{\mathcal{L}(C) \cap \mathcal{L}([\hat{}p'_1 \cdots p'_l]^*) = \emptyset}[\hat{}p'_1 \cdots p'_l]^*)$.

Intuitively, $\mathcal{L}(C) \cap \mathcal{L}([\hat{}p'_1 \cdots p'_l]^*) = \emptyset$ means that if we avoid all $p'_1 \cdots p'_l$, the constraint has no chance of being satisfied (so if it is not, we do not particularly care). In general, $Neg(C)$ looks like Fig. 5, where the labels is the intersection of labels that can change the truth value of the constraint. For example, we have $\{\mathsf{A}, \mathsf{B}, \mathsf{C}\}$ for exclusive choice1of3(A,B,C).

Interestingly, positive support checks for presence of some parameters whereas negative support checks for absence, but we can compute them using a single computation. Table 2 shows for each Declare constraint the corresponding label set for the support automaton in Fig. 5. Note that the constraint init(a) has neither positive nor negative support.

Intuitively, it may seem that it would be possible for a constraint to have both a nontrivial positive and nontrivial negative support automaton. This is not the case, however.

Theorem 1. *No regular expression has both a nontrivial positive and nontrivial negative support automaton.*

Proof. Assume a constraint C with parameters $p_1, \ldots, p_p$ and two subsets $p'_1, \ldots, p'_k$, $q'_1, \ldots, q'_l \in \{p_1, \ldots, p_p\}$ with $\mathcal{L}(C) \cap \mathcal{L}([\hat{}p'_1 \cdots p'_k]^*) = \mathcal{L}([\hat{}p'_1 \cdots p'_k]^*)$ and $\mathcal{L}(C) \cap \mathcal{L}([\hat{}q'_1 \cdots q'_l]^*) = \emptyset$. We then get that $\mathcal{L}([\hat{}p'_1 \cdots p'_k]^*) \cap \mathcal{L}(C) \cap \mathcal{L}([\hat{}q'_1 \cdots q'_l]^*) = \mathcal{L}([\hat{}p'_1 \cdots p'_k]^*) \cap \emptyset = \emptyset$ (adding intersection with $\mathcal{L}([\hat{}p'_1 \cdots p'_k]^*)$ on both sides). Using the first equality, we get $\mathcal{L}([\hat{}p'_1 \cdots p'_k]^*) \cap \mathcal{L}([\hat{}q'_1 \cdots q'_l]^*) = \emptyset$. However, ε would lie in both $\mathcal{L}([\hat{}p'_1 \cdots p'_k]^*)$ and $\mathcal{L}([\hat{}q'_1 \cdots q'_l]^*)$ contradicting that the intersection is the empty set, contradicting that a constraint can have nontrivial both positive and negative support. □

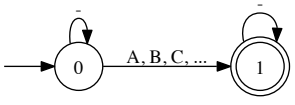


Fig. 5: Automaton recognizing positive and negative support.

Table 2: Declare constraints and their support: The second column gives the label set for the arc connecting state 0 and 1 in Fig. 5.

	Constraint (C)	Label set of the support automaton
position	init(a)	–
	strong init(a)	a
	last(a)	a
count	existence(a,n)	a
	absence(a,n)	a
	exactly(a,n)	a
ordered	precedence(a,b)	b
	response(a,b)	a
	succession(a,b)	a,b
	alternate(a,b)	a
	alternate precedence(a,b)	b
	alternate response(a,b)	a
	alternate succession(a,b)	a,b
	chain precedence(a,b)	b
	chain response(a,b)	a
	chain succession(a,b)	a,b
unord.	responded existence(a,b)	a
	co-existence(a,b)	a,b
choice	choice1of2(a,b)	a,b
	choice1of3(a,b,c)	a,b,c
	choice1of4(a,b,c,d)	a,b,c,d
	choice1of5(a,b,c,d,e)	a,b,c,d,e
	choice2of3(a,b,c)	a,b,c
	exclusive choice1of2(a,b)	a,b
	exclusive choice1of3(a,b,c)	a,b,c
	exclusive choice2of3(a,b,c)	a,b,c
negative	not co-existence(a,b)	a,b
	not succession(a,b)	a,b
	not chain succession(a,b)	a,b

For a better understanding of when constraints have positive or negative support, we define prefix-closure of a language.

Definition 6 (prefix-closure). *Let C be a constraint and Par be the set of parameters of C. If $a \in \mathcal{L}(C) \implies \forall v \notin Par(C) : av \in \mathcal{L}(C)$, then $\mathcal{L}(C)$ is* prefix-closed.

The next lemma proves some facts positive and negative support. With this lemma, we can illustrate how negative support, positive support, and prefix closure are related using Fig. 6.

Lemma 1. *Let C be a constraint. Then the following holds*

1. $Neg(C) \neq \emptyset \implies \varepsilon \notin \mathcal{L}(C)$.
2. $Pos(C) \neq \emptyset \implies \varepsilon \in \mathcal{L}(C)$.
3. *If* $\mathcal{L}(C)$ *is prefix-closed and* $\varepsilon \in \mathcal{L}(C)$ *then* $Pos(C) \neq \emptyset$.

Proof. (1) If $Neg(C) \neq \emptyset$, then $\mathcal{L}(C) \cap \mathcal{L}([\hat{}p'_1 \cdots p'_l]^*) = \emptyset$, for all parameter sets $p'_1 \cdots p'_l$. Because $\varepsilon \in \mathcal{L}([\hat{}p'_1 \cdots p'_l]^*$, we conclude $\varepsilon \notin \mathcal{L}(C)$.

(2) There exist $p'_1 \cdots p'_k$ such that $\mathcal{L}(C) \cap \mathcal{L}([\hat{}p'_1 \cdots p'_k]^*) = \mathcal{L}([\hat{}p'_1 \cdots p'_k]^*)$ by assumption. Because $\varepsilon \in \mathcal{L}([\hat{}p'_1 \cdots p'_k]^*$, we conclude $\varepsilon \in \mathcal{L}(C)$.

(3) Let Par be the set of parameters of C. Because $\mathcal{L}(C)$ is prefix-closed and $\varepsilon \in \mathcal{L}(C)$, we have $\sigma \in \mathcal{L}(C)$, for all $\sigma \in (\Sigma \setminus Par(C))^*$. Then, we obtain $\mathcal{L}(C) \cap \mathcal{L}([\hat{}p'_1 \cdots p'_k]^*) = \mathcal{L}([\hat{}p'_1 \cdots p'_k]^*)$, for $\{p'_1, \cdots, p'_k\} = Par(C)$. From this, we conclude $Pos(C) \neq \emptyset$. □

It is possible for a trace to have positive support without satisfying a constraint (i.e., it is triggered but not satisfied), and for a constraint to not be satisfied despite not having negative support. These are, in fact, really the interesting cases as a constraint was triggered and we are interested in whether it is satisfied in that case. For this reason, we define the notion of *dependent support*, which captures this.

Definition 7 (dependent support). *A trace* σ *has* dependent support *if* $\sigma \in \mathcal{L}(Pos(C)) \cap \mathcal{L}(A(C))$ *or* $\sigma \in \mathcal{L}(Neg(C))^c \cap \mathcal{L}(A(C))$.

We note that as $\mathcal{L}(Neg(C)) \supseteq \mathcal{L}(A(C))$, we have that $\mathcal{L}(Neg(C)) \cap \mathcal{L}(A(C)) = \mathcal{L}(A(C))$ if it is nontrivial. Figure 7 illustrates dependent support (the highlighted area) as a partitioning of all traces. As an example, for exclusive choice1of3(A,B,C) all accepting traces of the automaton in Fig. 2(c) have dependent support whereas for precedence(A,B) and response(A,B) only the accepting traces that leave the initial state of Fig. 2(b) and (c), respectively, have dependent support.

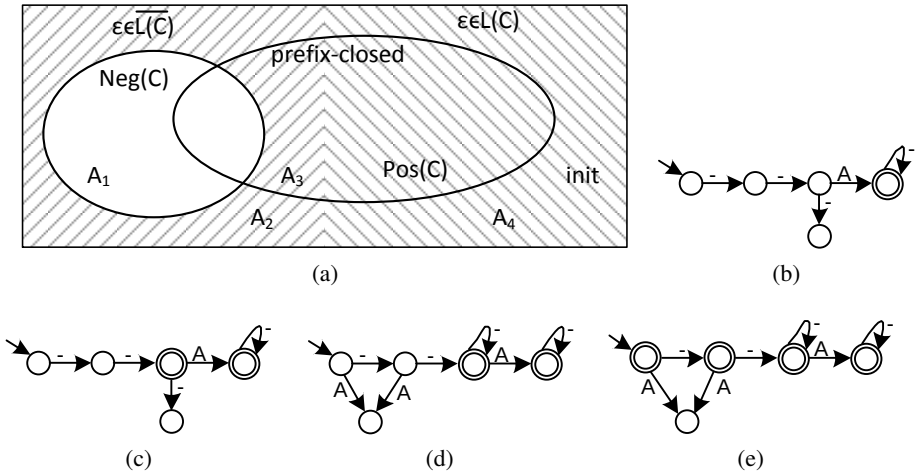


Fig. 6: Illustration of how negative support, positive support, and prefix closure are related (a) and four automata A_1–A_4 (b)–(e) to show the differences.

(a) (b)

Fig. 7: Dependent support for constraints with nontrivial positive (a) and negative (b) support.

As positive and negative support cannot happen at the same time by Theorem 1, this means the logical disjunction always is exclusive. For a set $\mathcal{L}$ of traces, we say that the *confidence* of the log is the number of traces with dependent support divided by sum of the number of traces with positive or negative support. As traces with dependent support is a subset of traces with positive or negative support, this number is between 0 and 1. We use the convention that 0 divided by 0 is 0 here. This measures how large a percentage of triggered constraints were satisfied.

7 Experimental Results

Everything presented hitherto has been implemented in the process mining framework ProM [1][4]. In this section, we show that for our running example, we get meaningful values allowing us to recognize a constraint which was previously impossible. We also illustrate that our algorithm and implementation is not only more expressive but also faster on a real-life data set. Figure 8 shows a screen-shot from the configuration of the miner.

To illustrate the usefulness of the vacuity measurements defined previously, we summarize the support measurements for seven constraint instances in Table 3. We see that with a threshold of 80 for confidence, we get sensible results: precedence(Double Check, Pay High) and exclusive choice1of3(Reject, Pay High, Pay Low) hold. We also see that we reject an obviously meaningless constraint, response(Pay High, Pay Low). We would reject response(Double Check, Pay High), but see we pay out in 55% of cases, indicating that making the extra check is worth the effort. We also reject the exclusive choice1of3(Double Check, Reject, Pay High) constraint, which would previously not be possible. The remaining exclusive choices are both mostly correct, but not completely precise. For reference, for the log in Fig. 1b, all instances of precedence, response, and exclusive choice1of3 with confidence over 50% are shown in Table 3.

To assess the speed of various implementations, we tested the ProM Declare miner, MINERful++, and the UnconstrainedMiner on the BPI challenge log from 2012 [2]. The reason for using this in place of the newer 2013 log is that the 2013 log is trivial for Declare mining due to a low number of event classes. We perform four tests: just one constraint (succession), all the easy constraints (the constraints from Table 1 except for the choice group), nearly all constraints (all constraints save for choice1of4 and choice1of5), and all constraints. Table 4 summarizes the results. We see that the

[4] To run the tool, add the UnconstrainedMiner package using the ProM package manager.

Fig. 8: Configuration of the UnconstrainedMiner in ProM.

Table 3: Support measurements for the running example from Fig. 1.

Constraint	Measurement (%)				
	match	pos	neg	dep	conf
precedence(Double Check, Pay High)	99	6	0	5	83
response(Double Check, Pay High)	96	9	0	5	55
response(Pay High, Pay Low)	94	6	0	0	0
exclusive choice1of3(Reject, Pay High, Pay Low)	100	0	100	100	100
exclusive choice1of3(Double Check, Reject, Pay Low)	95	0	99	95	96
exclusive choice1of3(Double Check, Pay High, Pay Low)	75	0	80	75	94
exclusive choice1of3(Double Check, Reject, Pay High)	21	0	30	21	70

ProM Declare miner is by far the slowest. MINERful++ is quite good for the easy constraints. Neither of these can handle all the choices (the ProM Declare miner can handle choices with two parameters). Our base implementation is already fastest, but spends significant time for the difficult choices (note the unit is minutes here). Symmetry significantly improves speed for the hard constraints, but less so for the easy ones, which get less reduction. Adding prefix sharing is surprisingly enough slower than plain symmetry reduction despite a sharing ration of around a factor four. The reason is that the

base algorithm is extremely efficient, even though the trie implementation is efficient, it is still not enough to be better. For a sharing factor of over 15 we expect this to perform better. Running the base algorithm with symmetry reduction in parallel (here just for two cores) reduces time by approximately 33%. The reason we do not get a full 50% reduction is partly a bit of overhead, but also that modern CPUs use Turbo Boost, which makes a single core run faster if the other is idle. Super-scalar mining yields most if there are many constraints with similar difficulty, which is the case for the easy constraints and to some extent for the hard ones, but for all constraints choice1of5 dominates the computation and is the single constraint of this difficulty. Mining all constraints using symmetry reduction, super-scalarity and eight computation cores (that are slower than the ones used in the table), the UnconstrainedMiner ties with MINERful++ even though it mines more and more complex constraints.

8 Conclusion

In this paper, we provided a complete semantics for Declare using regular expressions. We suggested various improvements to basic mining algorithms, including an outline for a basic algorithm, using symmetry reduction, sharing prefixes, parallelization, and super-scalarity. All were implemented and evaluated. Indeed, our base algorithm is faster than the state of the art, and all reductions improve speed, save for prefix sharing, which is only better if prefix sharing is above a factor 15.

We proposed a generalized notion of support, which coincides with previous definitions where they are both defined. Our definition can be automatically computed, and generalizes to cases where previous attempts did not provide an answer. We demonstrated that for a toy example this notion leads to a meaningful model.

The two contributions in unison means (1) Declare mining is now so fast, we can use it as a stepping stone for other algorithms, even for real-life examples, and (2) we can automatically compute all measures; we can even add arbitrary constraints to mine for, even if they are not originally part of Declare. Future work includes investigating this, in particular by using Declare to mine models with block structure.

References

1. www.promtools.org

Table 4: Experimental results using the BPI challenge log from 2012.

Test set		Execution Time for Tool						
Name	p	ProM	MINERful++	UnconstrainedMiner using Algorithm				
				Base	Symmetry	Prefix	Parallel	Super-scalar
One	2	$129m$	–	–	–	–	–	–
Easy	≤ 2	$550m$	$26s$	$19s$	$16s$	$43s$	$12s$	$2s$
Hard	≤ 3	–	–	$117s$	$35s$	$112s$	$27s$	$7s$
All	≤ 5	–	–	$215m$	$154s$	$437s$	$108s$	$56s$

2. DOI: 10.4121/uuid:3926db30-f712-4394-aebc-75976070e91f
3. Aalst, W.M.P.v.d.: Process Mining: Discovery, Conformance and Enhancement of Business Processes. Springer (2011)
4. Aalst, W.M.P.v.d., Pesic, M., Schonenberg, H.: Declarative workflows: Balancing between flexibility and support. Computer Science - R&D 23(2), 99–113 (2009)
5. Bauer, A., Leucker, M., Schallhart, C.: Comparing ltl semantics for runtime verification. J. Log. Comput. 20(3), 651–674 (2010)
6. Bustan, D., Flaisher, A., Grumberg, O., Kupferman, O., Vardi, M.Y.: Regular vacuity. In: CHARME 2005. LNCS, vol. 3725, pp. 191–206. Springer (2005)
7. Di Ciccio, C., Mecella, M.: A two-step fast algorithm for the automated discovery of declarative workflows. In: CIDM 2013. IEEE (2013)
8. Eisner, C., Fisman, D., Havlicek, J., Lustig, Y., McIsaac, A., Campenhout, D.V.: Reasoning with temporal logic on truncated paths. In: CAV 2003. LNCS, vol. 2725, pp. 27–39. Springer (2003)
9. Giannakopoulou, D., Havelund, K.: Automata-Based Verification of Temporal Properties on Running Programs. In: ASE 2001. pp. 412–416. IEEE Computer Society (2001)
10. Hildebrandt, T., Mukkamala, R.R.: Declarative event-based workflow as distributed dynamic condition response graphs. In: Post-proc.of PLACES 2010 (2010)
11. Kupferman, O.: Sanity checks in formal verification. In: CONCUR 2006. LNCS, vol. 4137, pp. 37–51. Springer (2006)
12. Maggi, F.M.: Declarative process mining with the declare component of prom. In: BPM (Demos) 2013. CEUR Workshop Proceedings, vol. 1021. CEUR-WS.org (2013)
13. Maggi, F.M., Bose, R.P.J.C., Aalst, W.M.P.v.d.: Efficient discovery of understandable declarative process models from event logs. In: CAiSE 2012. LNCS, vol. 7328, pp. 270–285. Springer (2012)
14. Maggi, F.M., Bose, R.P.J.C., Aalst, W.M.P.v.d.: A knowledge-based integrated approach for discovering and repairing declare maps. In: CAiSE 2013. LNCS, vol. 7908, pp. 433–448. Springer (2013)
15. Montali, M.: Specification and Verification of Declarative Open Interaction Models - A Logic-Based Approach, LNBIP, vol. 56. Springer (2010)
16. Morgenstern, A., Gesell, M., Schneider, K.: An asymptotically correct finite path semantics for ltl. In: LPAR 2012. LNCS, vol. 7180, pp. 304–319. Springer (2012)
17. Pesic, M.: Constraint-Based Workflow Management Systems: Shifting Controls to Users. Ph.D. thesis, Eindhoven University of Technology (2008)
18. Pnueli, A.: The temporal logic of programs. In: FOCS. pp. 46–57. IEEE Computer Society (1977)
19. Vardi, M.Y., Wolper, P.: An automata-theoretic approach to automatic program verification (preliminary report). In: LICS. pp. 332–344. IEEE Computer Society (1986)
20. Westergaard, M.: CPN Tools 4: Multi-formalism and Extensibility. In: Application and Theory of Petri Nets and Concurrency. LNCS, vol. 7927, pp. 400–409. Springer-Verlag (2013)
21. Westergaard, M.: Better Algorithms for Analyzing and Enacting Declarative Workflow Languages Using LTL. In: BPM 2011. LNCS, vol. 6896, pp. 83–98. Springer (2011)
22. Westergaard, M., Stahl, C.: Leveraging super-scalarity and parallelism to provide fast declare mining without restrictions. In: BPM (Demos). CEUR Workshop Proceedings, vol. 1021. CEUR-WS.org (2013)

A Automata Representation of Declare Constraints

A.1 Positioned

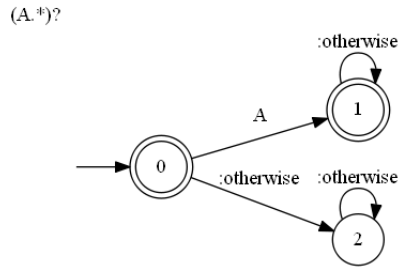


Fig. 9: Automaton modeling the constraint init(a).

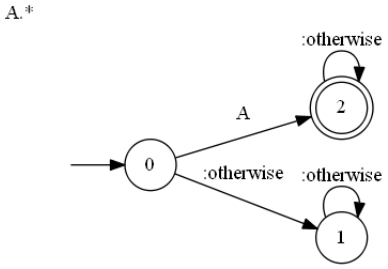


Fig. 10: Automaton modeling the constraint strong init(a).

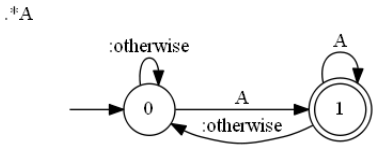


Fig. 11: Automaton modeling the constraint last(a).

A.2 Count

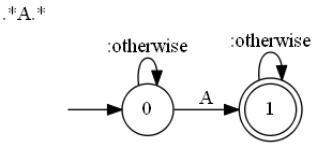


Fig. 12: Automaton modeling the constraint existence(a,1).

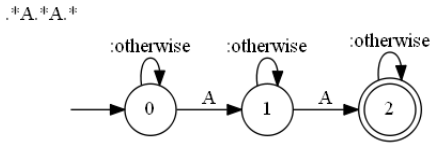


Fig. 13: Automaton modeling the constraint existence(a,2).

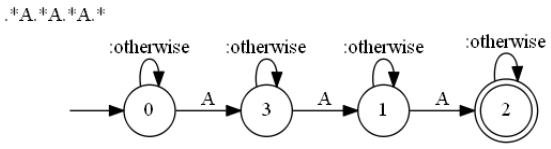


Fig. 14: Automaton modeling the constraint existence(a,3).

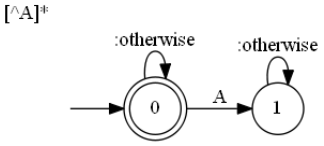


Fig. 15: Automaton modeling the constraint absence(a,1).

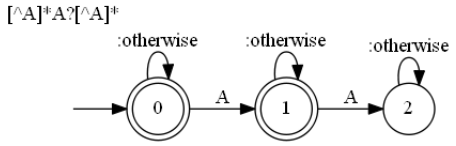


Fig. 16: Automaton modeling the constraint absence(a,2).

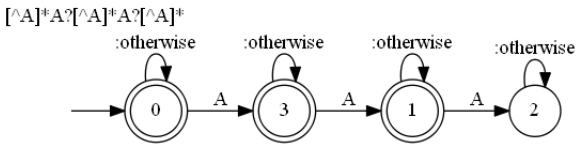


Fig. 17: Automaton modeling the constraint absence(a,3).

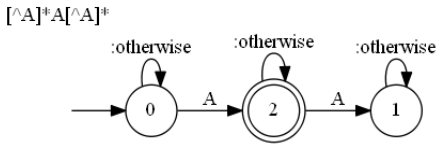


Fig. 18: Automaton modeling the constraint exactly(a,1).

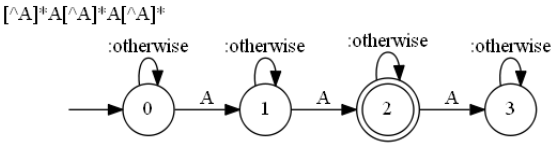


Fig. 19: Automaton modeling the constraint exactly(a,2).

A.3 Ordered

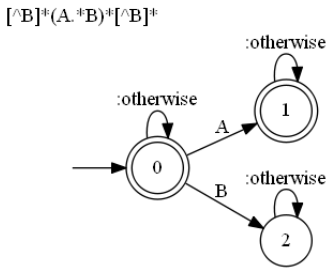


Fig. 20: Automaton modeling the constraint precedence(a,b).

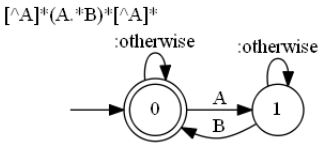


Fig. 21: Automaton modeling the constraint response(a,b).

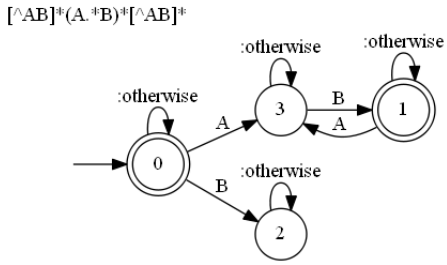


Fig. 22: Automaton modeling the constraint succession(a,b).

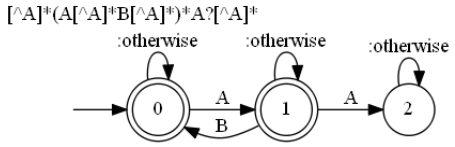


Fig. 23: Automaton modeling the constraint alternate(a,b).

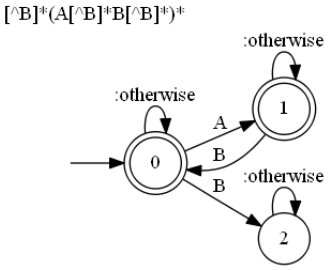


Fig. 24: Automaton modeling the constraint alternate precedence(a,b).

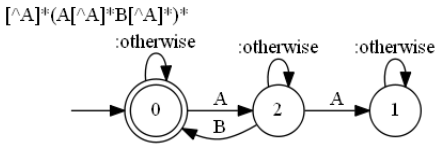


Fig. 25: Automaton modeling the constraint alternate response(a,b).

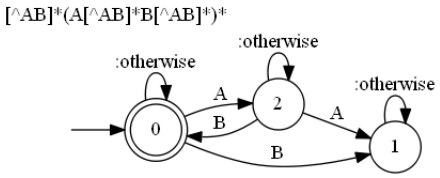


Fig. 26: Automaton modeling the constraint alternate succession(a,b).

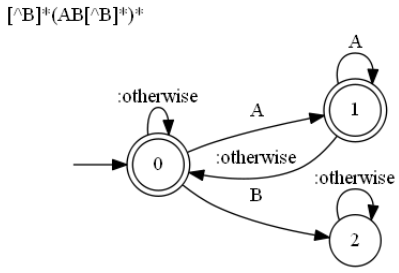


Fig. 27: Automaton modeling the constraint chain precedence(a,b).

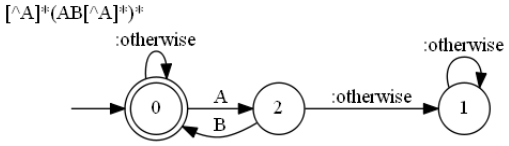


Fig. 28: Automaton modeling the constraint chain response(a,b).

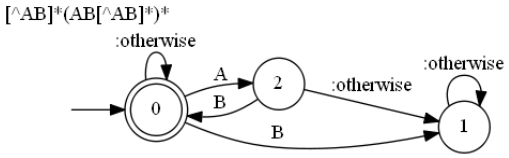


Fig. 29: Automaton modeling the constraint chain succession(a,b).

A.4 Unordered

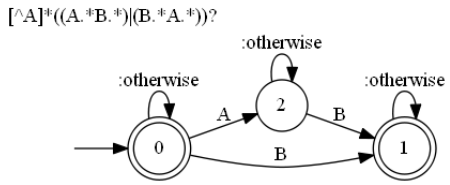


Fig. 30: Automaton modeling the constraint responded existence(a,b).

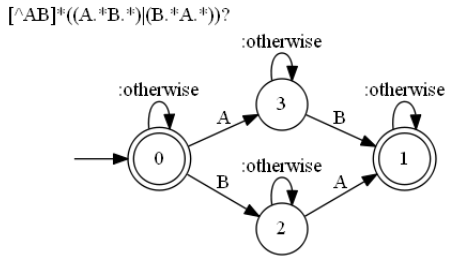


Fig. 31: Automaton modeling the constraint co-existence(a,b).

A.5 Choice

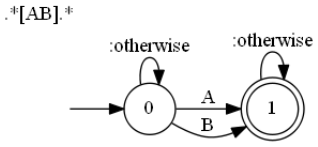


Fig. 32: Automaton modeling the constraint choice1of2(a,b).

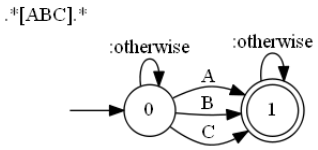


Fig. 33: Automaton modeling the constraint choice1of3(a,b,c).

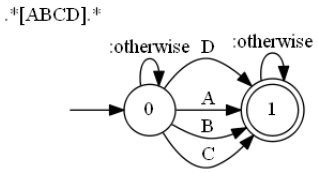


Fig. 34: Automaton modeling the constraint choice1of4(a,b,c,d).

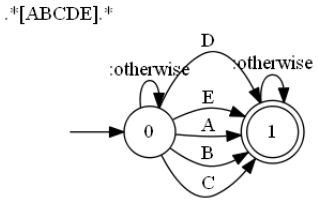


Fig. 35: Automaton modeling the constraint choice1of5(a,b,c,d,e).

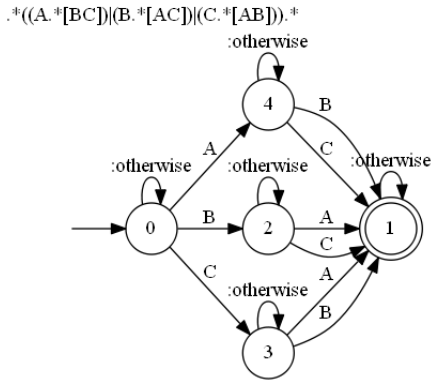


Fig. 36: Automaton modeling the constraint choice2of3(a,b,c).

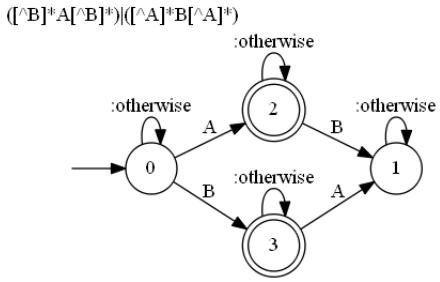


Fig. 37: Automaton modeling the constraint exclusive choice1of2(a,b).

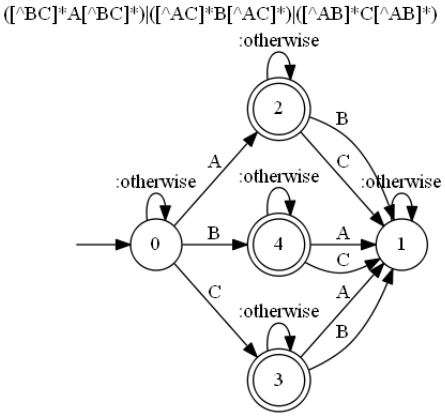


Fig. 38: Automaton modeling the constraint exclusive choice1of3(a,b,c).

([^C]*((A[^C]*B)|(B[^C]*A))[^C]*)|([^B]*((A[^B]*C)|(C[^B]*A))[^B]*)|([^A]*((B[^A]*C)|(C[^A]*B))[^A]*)

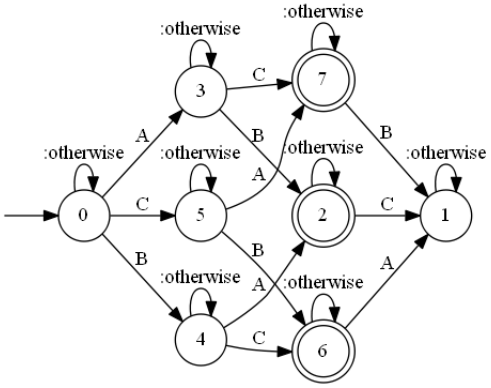


Fig. 39: Automaton modeling the constraint exclusive choice2of3(a,b,c).

A.6 Negative

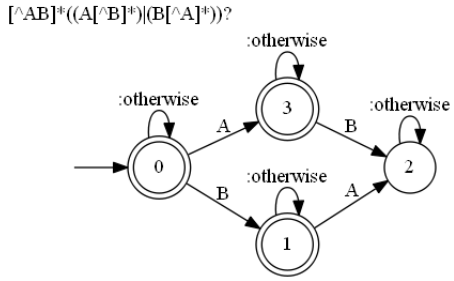


Fig. 40: Automaton modeling the constraint not co-existence(a,b).

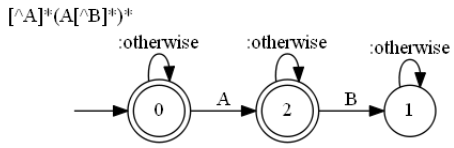


Fig. 41: Automaton modeling the constraint not succession(a,b).

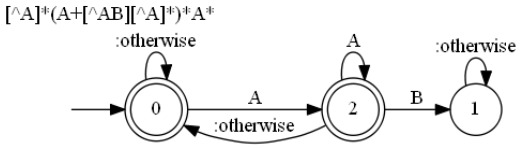


Fig. 42: Automaton modeling the constraint not chain succession(a,b).