Enhancing Process Mining Results using Domain Knowledge

P.M. Dixit[1,2], J.C.A.M. Buijs[2], W.M.P. van der Aalst[2], B.F.A. Hompes[1,2], and J. Buurman[1]

[1] *Philips Research, Eindhoven, The Netherlands*
[2] *Department of Mathematics and Computer Science*
Eindhoven University of Technology, Eindhoven, The Netherlands
{prabhakar.dixit,hans.buurman}@philips.com
{j.c.a.m.buijs,w.m.p.v.d.aalst,b.f.a.hompes}@tue.nl

Abstract. Process discovery algorithms typically aim at discovering process models from event logs. Most discovery algorithms discover the model based on an event log, without allowing the domain expert to influence the discovery approach in any way. However, the user may have certain domain expertise which should be exploited to create a better process model. In this paper, we address this issue of incorporating domain knowledge to improve the discovered process model. We firstly present a modification algorithm to modify a discovered process model. Furthermore, we present a verification algorithm to verify the presence of user specified constraints in the model. The outcome of our approach is a Pareto front of process models based on the constraints specified by the domain expert and common quality dimensions of process mining.

Keywords: user guided process discovery, declare templates, domain knowledge, algorithm post processing

1 Introduction

Process mining aims to bridge the gap between big data analytics and traditional business process management. This field can primarily be categorized into (1)process discovery, (2)conformance checking and (3)enhancement [1]. Process discovery techniques focus on using the event data in order to discover process models. Conformance checking techniques focus on aligning the event data on a process model to verify how well the model fits the data and vice versa [3]. Whereas enhancement techniques use event data and process models to repair or enrich the process model.

Most of the current process discovery approaches focus on discovering the process model entirely based on the event log. Enhancement techniques use end-to-end process models along with the event logs to repair and enrich the process models with information eg., to highlight bottlenecks or to annotate decisions with guards. In domains such as the healthcare sector, the underlying processes

are complex and case specific, hence the domain expert might only be aware of some conditions or constraints which should always hold in the process. However the domain expert might not be aware of the complete end-to-end process which is required as an input by the alignment based repair techniques. Nevertheless, the domain expert might be aware of certain sub-processes or *protocols* constituting to the end-to-end process. However, traditional process discovery techniques do not provide a way to incorporate the domain knowledge in order to discover and repair a more accurate process model, based on both domain knowledge and event logs. In this paper, we address the challenge of incorporating domain knowledge in traditional process model discovery to overcome challenges such as infrequent and/or incomplete data.

Domain knowledge can be introduced in the discovery process at multiple stages as shown in Figure 1. In our approach, we *post-process* an already discovered process model to incorporate the user specified domain knowledge. Process models can be represented by multiple modeling notations, for example BPMN, Petri nets, process trees etc. State of the art discovery algorithms such as the Inductive Miner [8] and the Evolutionary Tree Miner [4] discover block-structured process models represented by the notion of *process trees*. We use process trees in our approach as they are hierarchically structured and sound by construction. The hierarchical nature of process trees allows for a structured way to incorporate and validate the domain knowledge from the user. Our approach is generic and scalable as it is independent of the inherent discovery algorithm, and can be applied to any discovery approach which produces process trees.

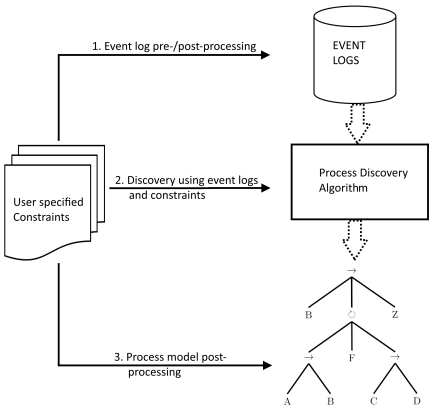


Fig. 1: Places where domain knowledge can be incorporated in the process discovery approach. Here we focus on 3.

In order to provide a very handy and effective way to gather user input, we make use of *Declare* templates [13]. *Declare* templates belong to the class of declarative languages, which are used to construct constraint based declarative process models. We do not use any aspect of declarative modeling in our approach. Only abstract *Declare* templates are used as a way for user to specify domain knowledge effectively in terms of constraints.

Figure 2 gives a high level overview of our approach. The ultimate goal is to provide a generic way to post-process the process tree such that the user is presented with a balanced set of optimal variants of process trees. Post-processing

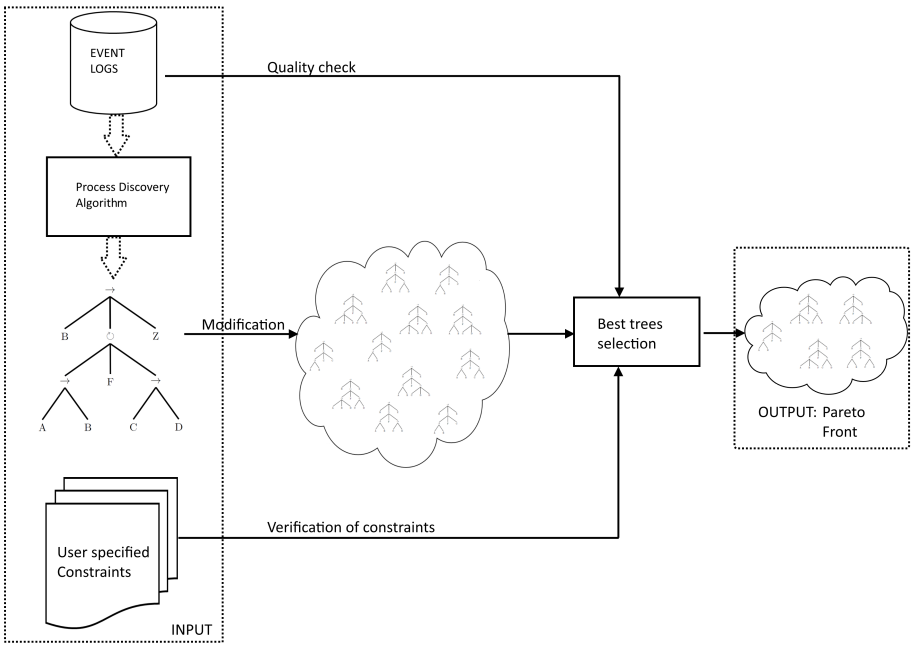


Fig. 2: The general overview combining traditional process discovery with domain knowledge specified using constraints. The original process tree is modified to generate probable candidate process trees. A competitive set of process trees is selected based on the verification of users constraints and quality dimensions.

the process tree with domain knowledge would enable us to overcome incompleteness and exceptional behaviour which could be wrongly represented in the originally discovered process tree. In order to achieve this goal, we first introduce a modification approach to generate a list of candidate process trees. We then introduce a novel verification algorithm to check whether a set of constraints is satisfied by a process tree. The candidate process trees are used in combination with the verification algorithm, number of edit operations and four quality dimensions to select a set of competitive process trees. The remainder of the paper is structured as follows. In Section 2 and Section 3, we provide a literature review of related work and the preliminaries respectively. In Section 4 and Section 5 we explain the modification and verification algorithms. In Section 6 we evaluate our approach based on synthetic and real life event logs. In Section 7 we conclude and discuss future research.

2 Related Work

Although the field of process discovery has matured in recent years, the aspect of applying user knowledge for discovering better process models is still in its nascent stages. Conformance techniques in process mining such as [2, 3, 5] replay event logs on the process model to check compliance, detect deviations and

bottlenecks in the model. These techniques focus on verifying the conformance of event logs with process model, but do not provide any way of incorporating domain knowledge to repair/improve the process model. In [11], the authors provide a way to mine declarative rules and models based on event logs, but do not allow users to introduce domain knowledge in rule discovery. The conformance based repair technique suggested by [6] takes a process model and an event log as input, and outputs a repaired process model based on the event log. However, the input required for this approach is an end-to-end process model and a noise free event log. Our approach requires only parts of process models or constraints described using declarative templates. Genetic algorithms [12] in process mining provide a possibility of using hand-made models as initial population. However this information may fade over time and may eventually get lost.

In [14], authors suggest an approach to discover a control flow model based on event logs and prior knowledge specified in terms of augmented Information Control Nets (ICN). Our approach mainly differs in the aspect of gathering domain knowledge. Although declarative templates can also be used to construct a network of related activities (similar to ICN), it can also be used to provide a set of independent pairwise constraints or unary constraints. The authors of [7] incorporate both positive and negative constraints during process discovery to discover C-net models. Compared to this, our approach differs mainly in two aspects. Firstly, we do not propose a new process discovery algorithm, but provide a generic approach to post process an already discovered process tree. Secondly, our approach provides the user with a balanced set of process models which maximally satisfy user constraints and score high on quality dimensions.

3 Preliminaries

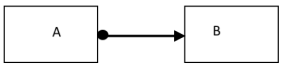
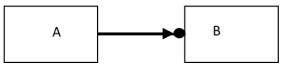
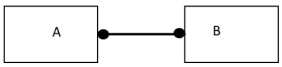
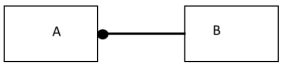
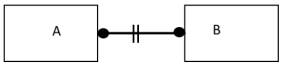
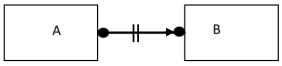
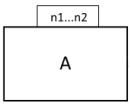
As mentioned in Section 1, we primarily use *Declare* templates as a means to incorporate the domain knowledge and process trees to represent the process models. This section provides a background and a brief description about process trees and *Declare* templates.

3.1 Declare Templates

A declarative model is defined by using constraints specified by a set of templates [13]. We use a subset of *Declare* templates as a way to input domain knowledge.

Table 1 provides an overview and interpretation of the Declare constraints that we consider [11, 13]. Binary templates provide ways to specify dependency (positive and negative) between two activities. For example, $response(A, B)$ specifies that activity A has to be eventually followed by activity B somewhere in the process. We use six binary constraints as shown in Table 1. We use one unary constraint $existence(n1, n2, A)$, as a way to specify the range of occurrence of an activity.

Table 1: Declare constraints and their graphical and textual interpretations

Template Name	**Graphical Representation**	**Interpretation**
$response(A, B)$		Activity B should (always) eventually occur after activity A
$precedence(A, B)$		Activity B can occur only after the occurrence of activity A
$coexistence(A, B)$		Activity A implies the presence of activity B (and vice versa)
$responded - existence(A, B)$		Activity B should (always) occur before or after the occurrence of activity A
$not - coexistence(A, B)$		Activity A implies the absence of activity B (and vice versa)
$not - succession(A, B)$		Activity A should never be followed by activity B
$existence(n1, n2, A)$		Activity A should occur: • n1..n2 times

3.2 Process Trees

Process trees provide a way to represent process models in a hierarchically structured way containing operators (*parent nodes*) and activities (*leaf nodes*). The operator nodes specify control flow constructs in the process tree. Figure 3 shows an example process tree. A process tree is traversed from left to right.

The order of child nodes is not important for *and* ($\wedge$), *exclusive-or* ($\times$) and *inclusive-or* ($\vee$) operators, unlike *sequence* ($\rightarrow$) and *Xor-loop* ($\circlearrowleft$) where the order is significant. In the process tree from Figure 3, activities A and Z are always the first and last activities respectively. For the $\circlearrowleft$ operator the left most node

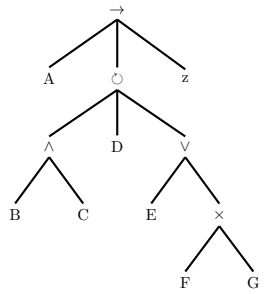


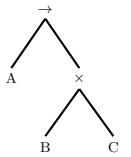
Fig. 3: Example Process tree showing *sequence* ($\rightarrow$), *and* ($\wedge$), *exclusive-or* ($\times$), *inclusive-or* ($\vee$) and *Xor-loop* ($\circlearrowleft$) operators

is the 'do' part of the loop and is executed at least once. In Figure 3, activity D is the optional 're-do' part of ↻, execution of which activates the loop again. Activities B and C occur in parallel and hence the order is not fixed. The right node of the loop is the escape node and it is executed exactly once. For the × operator, only one of either F or G is chosen. For the ∨ operator both × and activity C can occur, or only one of either two can occur.

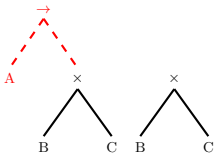
4 Modification

Following our methodology depicted in Figure 2, we start with the modification algorithm. As shown in Figure 2, the modification algorithm takes the discovered process tree and generates a list of candidate trees. This is accomplished using a "brute force" modification approach with the following steps:

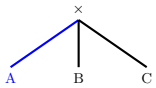
1. Starting with the original input process tree, variants are created based on three primary edit operations: *Add node*, *Remove node* and *Modify node*.
2. Every node in the process tree is subject to each edit operation, resulting in a new variant of process tree.
3. Each variant of process tree is further edited by iteratively calling all the edit operations exhaustively (in any order) using a "brute force" approach.
4. Every variant of the process tree is added to a pool of candidate process trees.
5. The process of creating process tree variants is repeated until a certain configurable threshold for *number of edit* operations w.r.t. each process tree is reached.



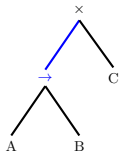
(a) Process tree discovered by the discovery algorithm. *(number of edits: 0)*



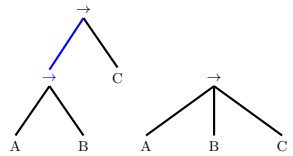
(b) Remove node: Removed activity A; also resulting in removal of → *(number of edits: 1)*



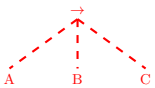
(c) Add node: Added activity A as a child of × *(number of edits: 2)*



(d) Add node: Added operator → as a child of ×; and parent of A and B *(number of edits: 3)*



(e) Modify node: Modified operator × to →; resulting in a process tree with only one parent operator *(number of edits: 4)*



(f) Remove node: Removed operator →; resulting in an *empty* process tree *(number of edits: 5)*

Fig. 4: Example modification operations on process tree.

It is important to carefully set the threshold for maximum number of edit operations, as a high threshold could result in many changes and a small threshold would only explore a few changes in the resultant process tree as compared to the original process tree. Hence there is no easy way to choose the threshold for selecting an optimal value for number of edit operations. It should be chosen by the user based on the original (starting) process tree, and the number of unverified constraints in the original process tree. A very high threshold value would result in more variants of process tree, however it would also be very compute intensive and inefficient. In order to improve the efficiency of the modification approach we can use techniques such as genetic algorithms or greedy algorithms to reduce the search space, discussed in Section 7.

Figure 4 shows different edit operations used in the modification algorithm. The *Modify node* operation exhaustively modifies every node in the process tree and can be classified into *Modify activity* and *Modify operator* depending on the type of node. Similarly, *Add node* iteratively adds either an activity node or an operator node (Figure 4c and Figure 4d). An operator can be added below the parent node (Figure 4d) and above the parent node (not shown in Figure 4) by exhaustively combining child nodes. Each edit operator results in a new process tree, which can be further edited by other edit operations exhaustively until the threshold for edit distance is reached. Every process tree arising after each edit operation is added to the pool of candidate process trees. By executing all edit operations in an iterative way, we can find an optimal sequence of operations to deduce *any* process tree with minimal edit operations. As shown in Figure 4f, we can reduce a process tree to an empty tree thereby ensuring the completeness of our modification approach.

Each process tree is evaluated against the four quality dimensions of process mining (replay fitness, precision, generalization and simplicity) [1], the number of user specified constraints verified, and the number of edit operations performed on the tree. This results in six quality dimensions. In order to evaluate the process trees based on these dimensions we use a Pareto front [4]. The general idea of a Pareto front is that all models are mutually non-dominating: A model is dominating with respect to other models, if for all measurement dimensions it is at least equal or better and for one strictly better. Using the six dimensions a Pareto front is presented to the user which contains the set of dominating process trees. For details about quality dimensions and Pareto front evaluation we refer to [4]. The verification of user constraints is covered in Section 5.

5 Verification

The verification algorithm takes a process tree and a set of constraints as input, and returns the set of constraints satisfied by the process tree as output. Each candidate process tree produced by the modification algorithm is verified. In this section, we present a novel verification approach to assist the selection of the best candidate process trees. The major advantage of our approach is that since we utilize a posteriori approach, it is independent of underlying process discovery

Algorithm 1: Declare constraints verification in a process tree

```
Input: process tree, set of constraints
Output: constraints categorized as verified or unverified
1  begin
2      foreach constraint do
3          if not existence constraint then
4              compute collection of common sub-trees
5          else
6              consider full tree
7          foreach sub-tree do
8              if not existence constraint then
9                  verify common parent
10                 verify position of activties
11                 if common parent or position verification fails then
12                     set constraint verification unsuccessful
13             if relations constraint & occurs(ST_A,B) is always then
14                 set constraint verification successful
15             if negative relations constraint & occurs(ST_A,B) is (never) then
16                 set constraint verification successful
17             if existence constraint then
18                 check range from occurs(PT,A) to occurs_multiple_times(PT,A)
19     return set of constraints - verified vs unverified
```

algorithm used to discover the initial process tree, and hence extremely generic and scalable. In algorithm 1, we show the main sequence of steps used by the verification approach. In the following sub-sections, we detail the algorithm.

5.1 Sub-tree Computation & Position Verification

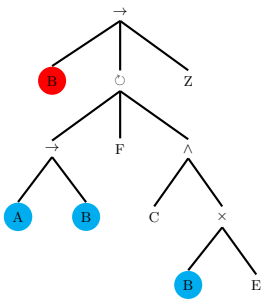
Sub-trees are the sub-blocks containing the first common ancestor between the two activities of the binary (relation or negative relation) constraints. The same activity can be present at multiple locations in a process tree which could result in multiple sub-trees for a single constraint, with the total number of sub-trees equal to the number of occurrences of the activity in the process tree. Formally, the computation of the collection of sub-trees for a binary constraint defined on activities A,B can be given as:

$S_B = \{n' \mid n' \in N \ \wedge \ l(n) = B\}$

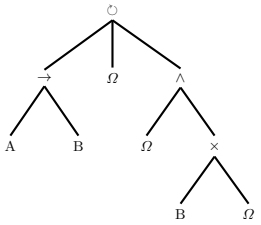
$STcollection_A = \{ST_n(S_B, PT) \mid n \in N \ \wedge \ l(n) = A \ \wedge \ ST_n(S_B, PT) \neq null\}$

where $STcollection_A$ is the collection of sub-trees w.r.t each node A, N is the collection of all nodes in process tree PT, S_B is the set of nodes labelled B, and $ST_n(S_B, PT)$ is the sub-tree computed w.r.t. node n explained in detail below.

Consider the constraint $response(A, B)$ that should be verified for the process tree from Figure 5a. As described in Table 1, a response constraint states



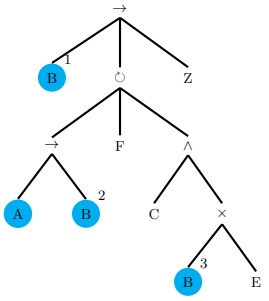
(a) Activities with cyan background are valid activities in process tree for constraint *response*(A, B)



(b) Sub-tree for *response*(A, B) with all the irrelevant activities marked as Ω

Fig. 5: Sub-tree computation for the constraint *response*(A, B)

that every occurrence of activity A should eventually be followed by activity B. In order to verify that such constraint holds true in the process tree, we first gather all the locations within the process tree where activity A occurs. For each occurrence of A in the process tree, we find the *first common ancestor* containing A and *all* the B's which can be reached after executing activity A. As a process tree is generally navigated from left to right, all the B's eventually occurring after A would naturally be on the *right* side of A. One caveat is that the order of children for the operators $\vee$, $\times$,and $\wedge$ is not fixed and the child nodes can be executed in any order. Hence there is an additional check required to verify the common parent, addressed in subsection 5.3. Figure 5b shows the sub-tree for constraint *response*(A, B). Since there is only one occurrence of activity A in the process tree, there is only one sub-tree. The first occurrence of B from the

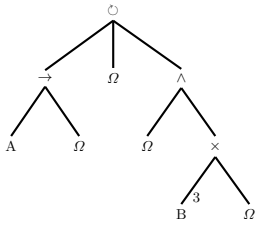


(a) *Precedence*(A, B) - Active nodes for this constraint are highlighted in cyan

null

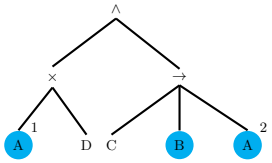
(b) *Precedence*(A, B) - Null sub-tree with respect to activity B_1

(c) *Precedence*(A, B) - Sub-tree with respect to activity B_2

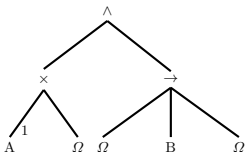


(d) *Precedence*(A, B) - Sub-tree with respect to activity B_3

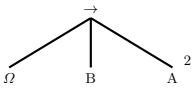
Fig. 6: Sub-trees computation for the constraint *precedence*(A, B). As Figure 6b results in a **null** sub-tree, the constraint verification for the constraint *precedence*(A, B) fails w.r.t. the entire process tree.



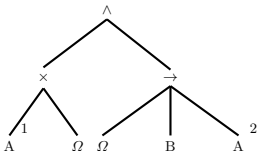
(a) *Responded* − *existence*(A, B) and *coexistence*(A, B) - Active nodes for these constraints are highlighted in cyan



(b) Sub-tree for constraints *coexistence*(A, B) and *responded* − *existence*(A, B) with respect to activity A_1



(c) Sub-tree for constraints *coexistence*(A, B) and *responded* − *existence*(A, B) with respect to activity A_2



(d) Sub-tree for constraint *coexistence*(A, B) with respect to activity B

Fig. 7: Sub-trees computation for constraints *responded* − *existence*(A, B) and *coexistence*(A, B)

original process tree is ignored as it is on the *left* side of A, and hence this B cannot be guaranteed to be executed *after* executing activity A.

For the *precedence*(A, B) constraint; we are interested in finding all the common sub-trees with respect to B, containing all A's on the left side of (*executed before*) B. There are a total of 3 sub-trees corresponding to each B in the process tree from Figure 6. The sub-trees for B_2 and B_3 are shown in Figure 6c and Figure 6d respectively. However, for B_1 there is no sub-tree containing activity A prior to *(i.e. on the left side of)* B. This results in a *null* sub-tree as shown in Figure 6b, and therefore the verification fails.

Relation constraints such as coexistence and responded-existence are independent of the position of the other activity in the process tree. Figure 7 shows the sub-trees for constraints *responded* − *existence*(A, B) and *coexistence*(A, B). The sub-tree from Figure 7d is calculated with respect to activity B and is only valid for the constraint *coexistence*(A, B). The sub-trees for negative relations constraint are calculated in a similar way to their respective relations constraints counterpart. However, unlike relation constraints, for negative relations constraints the absence of a sub-tree (*null* sub-tree) for each activity from constraint implies satisfaction of the constraint in the process tree. Sub-tree calculation is not necessary for unary constraints such as existence, wherein we consider the entire process tree. The next step is to determine whether an activity will occur, as discussed in subsection 5.2.

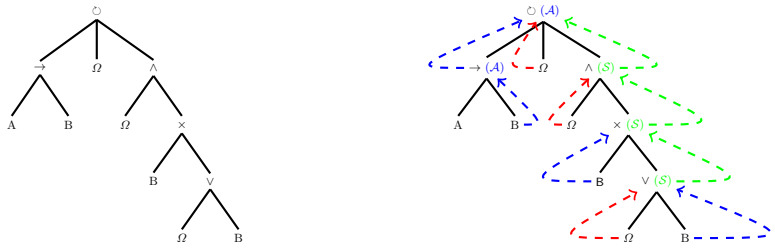
5.2 Activity Occurrence Verification

For binary constraints the next step after calculating the sub-trees is checking the occurrence of the activity in the sub-tree. In order to achieve this, we use the predicate $occurs(ST_A, B)$, where A is the node with respect to which sub-tree ST is computed and B is the second activity of the binary constraint. For every ancestor of node A, we check the occurrence of activity B which can have the following values: *always*, *sometimes* or *never*. Formally, this step can be defined as follows:

$\forall_{ST_n \in STcollection_A}\ \exists_{ancestor(n)}\ occurs(ST_A, B)$
where acceptable values for occurs(ST_A,B) are always and never for relation and negative relation constraints resp.

Figure 8b shows the occurrence of activity B, for the sub-tree from Figure 8a which is computed with respect to activity A. For choice operators such as $\times$ and $\wedge$, if activity B is present in *all* the child nodes, then activity B occurs *always* w.r.t. the operator node. If only few or none of the children of the choice operator have occurrence of activity B, then activity B occurs *sometimes* or *never* resp. Similarly, if at least one child of $\rightarrow$ and $\wedge$ is activity B, then activity B occurs *always* w.r.t. this node. In case of $\circlearrowleft$ if activity B is present only in the re-do part of the loop (which may or may not be executed), then activity B occurs *sometimes*. If activity B is present in the loop or exit child of the $\circlearrowleft$ operator, then activity B is guaranteed to occur *always* w.r.t. this node. We check the occurrence of activity B, at every ancestor of activity A. For binary relations constraint, if none of the ancestor(s) of activity A have the occurrence of B as *always*, then the constraint is not satisfied. On the contrary for negative relations constraints, if any of the ancestor(s) of activity A have the occurrence of B as *always* or *sometimes*, then the constraint is not satisfied. For every parent satisfying the constraint, we move on to validating the corresponding parent verification discussed in subsection 5.3.

In case of an unary constraint, the predicate *occurs_multiple_times(PT,A)* is calculated with possible values *yes* or *no*, where PT is the entire process tree and A is the activity from the unary constraint. If any of the ancestor(s) of activity A are children of the loop part or the re-do of $\circlearrowleft$ operator, then the multiple occurrence of activity A is set to *yes*. Otherwise, the multiple occurrence part of



(a) Sub-tree for constraint $response(A, B)$

(b) Blue, red and green colors indicate the occurrence always($\mathcal{A}$), never($\mathcal{N}$) and sometimes($\mathcal{S}$) respectively.

Fig. 8: *Occurrence(ST_A,B)* verification for constraint $response(A, B)$

Table 2: Overview of possible ranges for existence constraint

occurs(PT,A) at the root of PT	*occurs_multiple_times(PT,A)* at the root of PT	range of occurrence
sometimes	*no*	0..1
sometimes	*yes*	0..n
always	*yes*	1..n
always	*no*	exactly 1
never	n.a.	exactly 0

activity A is set to *no*. *occurs_multiple_times(PT,A)* gives us the upper bound of the range, and we combine this with *occurs(PT,A)* to calculate the lower bound of the range. We evaluate the unary constraints at the root of the tree depending on the values of *occurs(PT,A)* and *occurs_multiple_times(PT,A)*, as shown in Table 2.

5.3 Parent Verification

If occurrence verification for each activity from the binary constraint is successful, then the next step is to verify the common parent. There are a set of *allowable* common parent operators for each type of constraint. For example, if we have to verify the *coexistence*(E, B) constraint on the process tree Figure 5a, then one of the sub-trees computed is Figure 9. As the common parent for this sub-tree is the choice operator $\times$, both E and B will never occur together. Hence the common parent verification for this particular sub-tree fails for constraint *coexistence*(E, B). Table 3 summarizes the invalid common parents for all the constraints from Table 1.

Fig. 9: Sub-tree violating constraint *coexistence*(E, B)

For binary constraints, if either the sub-tree computation, position verification, common parent verification or activity occurrence verification fails, then that constraint is marked unsatisfied. If all these steps are successful for all the corresponding sub-trees, then the constraint is marked satisfied. For unary constraints, if activity occurrence verification is successful (within the input range) then the constraint is marked satisfied, otherwise, it is marked unsatisfied.

Table 3: Invalid common parents for each of the declare constraints

constraint	invalid common parent operator
response(A, B)	$\times$, $\vee$, $\circlearrowleft^1$, $\wedge$
precedence(A, B)	$\times$, $\vee$, $\circlearrowleft^1$, $\wedge$
coexistence(A, B)	$\times$, $\vee$, $\circlearrowleft^1$
responded $-$ *existence*(A, B)	$\times$, $\vee$, $\circlearrowleft^1$
not $-$ *succession*(A, B)	$\rightarrow$, $\vee$, $\circlearrowleft$, $\wedge$
not $-$ *coexistence*(A, B)	$\rightarrow$, $\vee$, $\circlearrowleft$, $\wedge$
	$\circlearrowleft^1$ is invalid only if node B (or A) is a child of the middle (redo) part

6 Evaluation

Evaluation of the candidate process trees can be done in multiple ways. One method could be to present the domain expert with a list of candidate process trees (or process models) to choose from. However this approach is highly subjective and would depend entirely on the preference of the domain expert, and hence would be difficult to quantify. Another approach for evaluation is to discover an *expected* model based on user specified constraints. In this approach there is a certain expected model, which isn't discovered by the traditional process discovery techniques due to reasons such as data inconsistencies, discovery algorithm biases etc. We use the latter approach for evaluation as it provides a quantifiable and controlled way to evaluate the results without depending on the high subjectivity of domain expert. We evaluate our approach based on both a synthetic log and a real life log.

6.1 Synthetic Event Log

We use a synthetic event log to demonstrate how our approach could improve an incorrect model discovered due to algorithm bias and noisy event log. For the event log $L = [\ \langle A,B,C,D\rangle^{90}, \langle A,C,B,D\rangle^{90}, \langle A,C,D,B\rangle^{90}, \langle C,A,D,B\rangle^{90}, \langle C,A,B,D\rangle^{90}, \langle C,D,A,B\rangle^{90}, \langle C,D,B,A\rangle^{6}, \langle C,B,A,D\rangle^{6}, \langle D,A,C,B\rangle^{6}]$, the Inductive Miner infrequent (IMi) [9] generates the process tree with all four activities in parallel as shown in Figure 10a.

From the high frequent traces of the log we can deduce simple rules such as activity A is always eventually followed by activity B; and activity B is always preceded by activity A. Similar relationship holds for activities C and D. We use this information and input the process tree discovered by IMi [9], event log (L) and the following four constraints in our algorithm: *response(A,B), precedence(A,B), response(C,D),* and *precedence(C,D).* Upon setting the maximum edit distance to 3, the modification algorithm creates 554 unique process trees resulting in a Pareto front of 7 process trees.

Figure 10 shows the original process tree discovered by Inductive Miner (Figure 10a) and a modified process tree (Figure 10b) with highest replay fitness and precision score from the Pareto front. Table 4 summarizes the dimension scores of the process trees from Figure 10. The modified process tree from Figure 10b satisfies all the four constraints. The number of edit operations required in order to discover the modified process trees is 2. Figure 10b also has a higher precision value of 1, and considerably high replay fitness score of almost 1. This process tree is highly precise, thereby explaining the high frequent traces of the event log

Fig. 10: Original and modified process trees for event log L.

Table 4: Quality dimensions of the Pareto front for process trees from Figure 10

tree	constraints satisfied	replay fitness	precision	generalization	simplicity	number of edits
Figure 10a	0	1	0.833	0.957	1	0
Figure 10b	4	0.997	1	0.957	1	2

much better and ignoring the infrequent noisy traces. From this we can conclude that by adding knowledge inferred from the event log to the discovered model it becomes possible to improve it considerably. This way, it is possible to overcome noise in the event log.

6.2 Real Life Event Log

Exceptional cases may dominate the normal cases, thereby leading to a process model that is over-fitting the data or that is too general to be of any value. This process model could however be improved by incorporating domain knowledge. In order to evaluate such a scenario, we use the following steps on a real-life log containing the road traffic fine management process with 11 events and 150,370 cases available at [10]:

- Use the complete event log to mine a process tree using IMi resulting in a structured process tree. Figure 11a shows the Petri net representation of this process tree. Learn *domain rules* based on this tree.
- Filter the event log to select 10% of the cases having *exceptionally deviating* behavior from the process model of Figure 11a.
- Create a process tree based on the filtered log using IMi. We assume that this process tree is our starting point, and input it to the modification algorithm. The Petri net representation of this process tree is shown in Figure 11b.
- Use the rules learnt from the original process model, in combination with the entire event log and modified trees to generate a Pareto front.

We deduce 2 coexistence, 2 responded-existence, 4 response and 1 not-succession rules (9 in total) from the original process model. We use an edit distance of 3 in the modification algorithm and stop creating variants of process trees after creating 500,000 unique process trees which results in a Pareto front of 54 process trees. In Table 5 we compare the original process tree, filtered process tree and the 5 modified process trees; out of which; trees 1,2, and 3 have the combined highest values for replay fitness and precision in the Pareto front, and trees 4 and 5 have highest individual values in the Pareto front for replay fitness and precision respectively. As we use the process model containing only 10%

(a) Process model mined with complete event log.

(b) Process model with filtered log containing infrequent traces only.

Fig. 11: Petri net models to show *structural dissimilarities* between models for complete and filtered event logs.

Table 5: Dimensions statistics for process trees based on real life event log

tree	constraints satisfied	replay fitness	precision	generalization	simplicity	number of edits
Complete log	9	0.970	0.872	0.983	1	0
Filtered log	2	0.957	0.740	0.845	1	0
Pareto Front 1	8	0.882	0.785	0.861	1	3
Pareto Front 2	8	0.817	0.812	0.839	1	3
Pareto Front 3	8	0.816	0.825	0.862	1	3
Pareto Front 4	8	1	0.576	0.009	1	3
Pareto Front 5	8	0.544	0.943	0.929	1	3

of the exceptional cases from the original log as our starting point, most of the modified process trees score lower than the original process tree on replay fitness and precision dimensions. Also, trees 4 and 5 from Table 5 demonstrate that post modification higher replay fitness could result in lower precision and vice versa. Although the filtered process tree has a higher score in terms of fitness, it has the lowest precision score among all the other trees (except tree 4). The process tree discovered from the complete event log scores the highest in all dimensions. However, the modified trees 1 to 3 from Pareto front (in Table 5) have a nicely balanced score of all dimensions and in general, explain the complete event log much better than the process tree from the incomplete filtered log. From a users perspective, depending on the preference, the user can select any process tree from the Pareto front. For example, if the user is looking for a process tree satisfying maximum constraints as well as describing the log very well, then then tree 4 from Table 5 seems to be the viable option. However, the user can also see the tradeoff in the Pareto front and the fact that although tree 4 satisfies users requirements, it scores very badly as compared to other trees in some other dimensions (precision and generalization). Hence, while choosing the models from Pareto front, the user can make an informed decision while considering the requirements as well as evaluating different dimensions.

7 Conclusions and Future Work

In this paper we introduced two algorithms in order to incorporate and verify domain knowledge in a discovered process model. The proposed verification algorithm provides a comprehensive way of validating whether the constraints are satisfied by the process tree. In the current approach we consider a subset of *Declare* templates. In the future this could be extended to include all the *Declare* templates. The current modification algorithm uses a brute force approach and exhaustively generates multiple process trees. However, currently the modification algorithm does not consider the user constraints during the modification process. In the future, we would like to improve upon the modification algorithm by modifying the process tree in a smarter way (for eg. using genetic or greedy algorithms), to optimise the modification approach and/or ensure certain guarantees in the modified process trees. Another future direction could be

to incorporate domain knowledge at different stages, for example when logging event data or during the discovery phase.

References

[1] van der Aalst, W.M.P.: Process Mining: Discovery, Conformance and Enhancement of Business Processes. Springer, Berlin (2011)

[2] Adriansyah, A., van Dongen, B.F., van der Aalst, W.M.P.: Conformance checking using cost-based fitness analysis. In: Enterprise Distributed Object Computing Conference (EDOC), 2011 15th IEEE International. pp. 55–64. IEEE (2011)

[3] Adriansyah, A., van Dongen, B.F., van der Aalst, W.M.P.: Towards robust conformance checking. In: Business Process Management Workshops, Lecture Notes in Business Information Processing, vol. 66, pp. 122–133. Springer Berlin Heidelberg (2011)

[4] Buijs, J.C.A.M., van Dongen, B.F., van der Aalst, W.M.P.: Quality dimensions in process discovery: The importance of fitness, precision, generalization and simplicity. Int. J. Cooperative Inf. Syst. 23(1) (2014)

[5] De Leoni, M., Maggi, F.M., van der Aalst, W.M.P.: Aligning event logs and declarative process models for conformance checking. In: Business Process Management, pp. 82–97. Springer (2012)

[6] Fahland, D., van der Aalst, W.M.P.: Repairing process models to reflect reality. In: Business process management, pp. 229–245. Springer (2012)

[7] Greco, G., Guzzo, A., Lupa, F., Luigi, P.: Process discovery under precedence constraints

[8] Leemans, S.J.J., Fahland, D., van der Aalst, W.M.P.: Discovering block-structured process models from event logs containing infrequent behaviour. In: Business Process Management Workshops. pp. 66–78. Springer (2014)

[9] Leemans, S.J.J., Fahland, D., van der Aalst, W.M.P.: Discovering block-structured process models from event logs containing infrequent behaviour. In: Business Process Management Workshops. pp. 66–78. Springer (2014)

[10] de Leoni, M., Mannhardt, F.: Road traffic fine management process, `http://doi:10.4121/uuid:270fd440-1057-4fb9-89a9-b699b47990f5`

[11] Maggi, F.M., Mooij, A.J., van der Aalst, W.M.P.: User-guided discovery of declarative process models. In: Computational Intelligence and Data Mining (CIDM), 2011 IEEE Symposium on. pp. 192–199. IEEE (2011)

[12] de Medeiros, A.K.A., Weijters, A.J.M.M., van der Aalst, W.M.P.: Genetic process mining: an experimental evaluation. Data Mining and Knowledge Discovery 14(2), 245–304 (2007)

[13] Pesic, M., Schonenberg, H., van der Aalst, W.M.P.: Declare: Full support for loosely-structured processes. In: Enterprise Distributed Object Computing Conference, 2007. EDOC 2007. 11th IEEE International. pp. 287–287. IEEE (2007)

[14] Rembert, A.J., Omokpo, A., Mazzoleni, P., Goodwin, R.T.: Process discovery using prior knowledge. In: Service-Oriented Computing, pp. 328–342. Springer (2013)